IBM Cúram Social Program Management
8.0.2

Cúram Server Developer's Guide

IBM

Note

Before using this information and the product it supports, read the information in "Notices" on page 199

Edition

This edition applies to IBM® Cúram Social Program Management 8.0.0, 8.0.1, and 8.0.2.

Licensed Materials - Property of IBM.

Contents

Figures

Tables

Chapter 1. Cúram server developer

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

Model-driven generation facilitates client and server development by taking a Unified Modeling Language (UML) model and producing the following generated artifacts:

- 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

Building and configuring a Cúram application

Use this information as a starting point for building and configuring a Cúram application, to find out what is supported in the build, and see how to configure the infrastructure. You can use the Data Manager tool to create a database. Use the SQL Checker to check the correctness of handcrafted SQL.

- – "Directory Structure" on page 1 provides an introduction to the layout of the application.
  - "Build files and their targets" on page 9 details the build support provided.
  - "Cúram Configuration Settings" on page 31 enumerates the various configuration settings supported by the infrastructure.
  - "Data Manager" on page 37 details the Data Manager tool that can be used to create a database to support the Cúram application.
  - "SQL Checker" on page 61 details the SQL Checker tool that can be used to ensure the semantic and syntactic correctness of SQL which has been handcrafted by an Application Developer.

Directory Structure

Use this information to learn about the directory structure for the server side IBM Cúram Social Program Management application, and the underlying Server Development Environment.

Application components

The Cúram server application is organized into collections of artifacts called *components*. Use this information to learn about the component folders and the *component order*, which is fundamental to the manner in which server artifacts are customized.

Component folders

Each component has its own folder in the `<EJBServer>/components` folder. The `core` component is always present. This contains all of the artifacts needed for the core functionality of the IBM Cúram Social Program Management Platform. The name of the component folder is used as the name of the component.

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.

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 implicitly is 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 MyComponentTwo 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.

Application directory structure

Use this information to understand two aspects of the Cúram application directory structure.

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

Source artifacts of the Cúram application

Use this information to understand the directory structure for the source artifacts of a Cúram application project.

Source artifacts of the Cúram application shows the directory structure for the source artifacts of a Cúram application project; for instance, the structure before it runs a build. The details that follow describe in more detail each directory within the directory 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 that is included with Cúram.
- The `CodePackage` is based on a model setting that 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 might 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
```

Figure 1. Cúram application structure

Table 1. Cúram application installation structure	
Name	**Contents**
project	A top-level directory that contains 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 a place holder for those components.
components/core	A pre-defined component that is used by all other components.
components/core/codetable	Code table XML (ctx) files that are created by the developer are kept here. These files are used to define code tables for a Cúram application. The outputs that are produced from a code table file consist of an SQL script to populate the code table in the database, and a Java file that provides the necessary constants to the application. For more information, see "Code table files" on page 88.
components/core/data	The Data Manager for this component.
components/core/doc	The JavaDoc for this component.

Name	Contents
Table 1. Cúram application installation structure (continued)	
Name	**Contents**
components/core/events	Event XML (evx) files that are created by the developer are kept here. These files are used to define event classes and event types for a Cúram application. The outputs that are 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 that provides the necessary constants to the application. For more information, see "Events and event handlers" on page 136.
components/core/lib	Contains the built component code packaged in a compressed file: such as core.jar. Additionally, any third-party Java Archive (JAR) files that are specified here automatically are included in the class path used during compilation or a Batch Launcher run. Files that are listed here also are added to any Enterprise ARchive (EAR) file that is created and an entry added to the manifest file to reference this file.
components/core/message	Message (.xml) files that are created by the developer are stored here. The Java artifacts that are produced from a message file are a Java file and a properties file. For more information, see "Message files" on page 82.
components/core/model	The elements of a Cúram application Unified Modeling Language (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 that are created by the developer are stored here. These files might be hand-crafted or created by way of an online client (Rules Editor).
components/core/sample	An optional directory that contains a compressed file of a set of sample java source files that match the component built code within the lib directory. Used for debugging or reference,
components/core/webservices	An optional directory that contains the .xsd schema files that are referenced by web services in this component.
components/core/workflow	Workflow process definition (.xml) files that are created by the developer are stored here. These files might be hand-crafted or created by way of an online client (Process Definition Tool). The *Cúram Workflow Reference Guide* describes these files in some detail.
components/core/wsdl	An optional directory that contains the .wsdl Web Service Description Language (WSDL) files that are started 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 when they are relative

Table 1. Cúram application installation structure (continued)	
Name	**Contents**
components/custom	Any number of new components might 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 command file wraps the build.xml file (an Apache Ant build file) that is contained within the EJBServer. The build structure and use of this file is described in "Build files and their targets" on page 9.
build.xml	An Ant build file that extends the Cúram Server Development Environment (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 command file wraps the build.xml file. The use of this file is described in "Build files and their targets" on page 9.

Cúram application build structure

Use this information to understand the directory structure that is created when a Cúram application is built.

The directory structure that is created when a Cúram application is built is described in the example and table that follow. The"Cúram application build structure" on page 5 presents the new directories that are created during the build process while "Cúram application build structure" on page 5 gives more details on the contents of each directory.

```
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
```

Figure 2. The Cúram application build structure

Table 2. Build directory structure	
Name	**Contents**
build/datamanager	Contains intermediate files produced by the Data Manager and the resulting merge Data Mining Extensions (DMX) files from the initial demonstration and test directories. The Data Manager creates the intermediate files when it is converting the database independent files into a format that can be loaded onto the database. Five database-dependent `.sql` files are produced in addition to one database-independent `.xml` file that 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 that are created by the command line project build.
build/sqlcheck	A database-dependent `sqlj` file that contains a subset of the dynamic SQL statements from the model and the inserts from the Data Manager collated together.
build/svr	All build artifacts for the server side.
build/svr/cls	All of the compiled class files for the application.
build/svr/gen	Generated server-side sources.

Table 2. Build directory structure (continued)

Name	Contents
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 artifacts.
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 files can be used as constants in the Cúram application.
build/svr/events/gen	The generated events file artifacts that include the `.java` files that contain the event class, 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 artifacts.
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-generated client stubs for each registered outbound web service connector.
<app.xml>	Extracted Unified Modeling Language (UML) model contents, which are named according to model.

Table 2. Build directory structure (continued)	
Name	**Contents**
buildlogs	A log file is created each time that a build is run and is stored here. This log file can be used to investigate any problems with the build process.

Artifacts of the SDEJ

Use this information to understand the directory structure of the SDEJ after installation is complete.

Artifacts of the SDEJ shows the directory structure of the Cúram Server Development Environment (SDEJ) when installation is complete, while Artifacts of the SDEJ gives more details on the contents of each directory. The CURAMSDEJ is the root of the directory structure - the name that is given to wherever the SDEJ is set up or installed.

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

Figure 3. SDEJ structure

Table 3. SDEJ structure at installation	
Name	**Contents**
bin	This directory contains all Ant build scripts necessary to build, verify, and configure a Cúram application. The `build.bat` script file that is delivered with the Cúram application hooks into this directory to start the `build.xml` file contained here. Use of this file is described in "Build files and their targets" on page 9
codetable	This directory contains the set of codetable files included with the SDEJ. These files use the file extension `.itx`. Each of these files can be customized, see "Localizing SDEJ code table files" on page 107 for more details.
doc	This directory contains the Javadoc information included with the SDEJ.
drivers	This directory contains the drivers that are 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 functions.

Table 3. SDEJ structure at installation (continued)

Name	Contents
message	This directory contains the set of message files included 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 "Localizing SDEJ Message Files" on page 87 for more details.
rsa	This directory contains the Eclipse plug-in artifacts that are used to provide Cúram functions in IBM Rational® Software Architect Designer. For more information, see the *Working with the Cúram Model in Rational Software Architect Designer*.
scripts	This directory contains the database independent XML files necessary to create the database required by the SDEJ.
util	This directory contains useful utilities included with the SDEJ.
xmlserver	This directory contains the artifacts and build scripts necessary to run the xmlserver. For more information, see *Cúram XML Infrastructure Guide*.

Build files and their targets

Use this information to understand how the Cúram Server Development Environment (SDEJ) uses Ant to process its build files and how to build a Cúram application after it is installed. The optional parameters that can be specified when you perform a build also are described.

The Ant build files are in the `/bin` directory of the SDEJ. The build files are started through `build.bat` and `buildhelp.bat`.

How to initiate the build

Starting **buildhelp** at the command line (in SERVER_DIR) shows all available targets. A single build target is required to build the Cúram application for development from its initial configuration. The user needs to initiate these actions:

- 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 (for example, *JAVA_HOME*, *J2EE_JAR*, and *ANT_HOME*). This process is described in the *Cúram Third-Party Tools Installation Guide*.
- 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 start this build target.

Overriding default JUNIT.JAR

The `junit.jar` file is set by default relative to the *JUNIT_HOME* environment variable; for example, `${sysenv.JUNIT_HOME}/junit.jar`. To override the location or 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 setting takes precedence over the default. An example of its usage (for example, Microsoft Windows): set JUNIT_JAR = c:\junit-4.8.jar

How to configure the build

Optional parameters that can be provided when you build the Cúram application are explained.

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:

Table 4. Build Configuration Settings		
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 to ensure, you catch errors before entering codetables onto the database. This does not override the maximum length on the database "Cúram Build Settings" on page 10.
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 "Cúram Build Settings" on page 10.

Table 4. Build Configuration Settings (continued)

Parameter	Values	Description
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 "Cúram Build Settings" on page 10.
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 "What is happening under the hood" on page 15 for more details. Note: The errors are still reported.
prp.noninterned strings	true false	Indicates whether code table artefacts should be generated with strings which will not be interned. This is described in more detail in "ctgen" on page 18. 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.

Table 4. Build Configuration Settings (continued)		
Parameter	**Values**	**Description**
extra.generator .options	String	Specifies additional command line parameters for the server code generator. These settings are described in "Generator Settings" on page 14.
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.
prp.genschema validation	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`.
appserver.failonerror	true false	Indicates whether the application server command will trigger an error if the start/stop command fails. The default is `true`.

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 `datamanager_config.xml` file before loading the code tables into the database. Please refer to "Data Manager" on page 37 for further information on using the Data Manager.

Java Compiler Settings
These parameters can be passed when performing the build and they control the behavior of the Java compiler.

The parameters are passed in the following way: **build server -Dcmp.debug=on**. The settings are:

Table 5. Java Compiler Settings		
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.

Parameter	Values	Description
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`.
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 compilation errors will fail the build. The default is `true`.
cmp.listfiles	yes no	Indicates whether the source files to be compiled will be listed. The default is no.
cmp.gc	-J-XX:+UseG1GC -J-XX:+UseParallelGC -J-XX:+UseSerialGC -J-XX:+UseConcMarkSweepGC	Controls the type of garbage collection used by the compiler. The default is -J-XX:+UseG1GC.
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.

Table 5. Java Compiler Settings (continued)

Table 5. Java Compiler Settings (continued)		
Parameter	**Values**	**Description**
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.

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:

Table 6. Java Task Settings		
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`.

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:

Table 7. Generator Settings	
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.

Table 7. Generator Settings (continued)	
Option	**Meaning**
-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. 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

Other Environment Settings

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

```
<errortext>unmappable character for encoding UTF8</errortext>
```

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

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

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:

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

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

Figure 4. 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** target transforms the intermediate XML representation into a set of data definition XML files, the final Java code, deployment support artifacts, and web service support artifacts.

Data definition XML files

The data definition XML files are placed in the `build/svr/gen/ddl` directory and are made up of the following 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 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 is captured in the UML model. `<SERVER_MODEL_NAME>_fid.xml` describes the persistent data that

is necessary to support the security-related information that is 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 target**checksql**. More information on the contents of these files is provided in "Data Manager" on page 37.

The final output of the **modelgen** target is a merged version of the data definition XML files, which are placed in the `build/ddl`. These merged files are named as follows:

- `Merged_Tables.xml`
- `Merged_Indices.xml`
- `Merged_PrimaryKeys.xml`
- `Merged_UniqueConstraints.xml`
- `Merged_ForeignKeys.xml`
- `Merged_Batch.xml`
- `Merged_Fids.xml`
- `Merged_FieldsReturned.xml`
- `Merged_SQLJ.xml`

Foreign Keys: IBM Cúram Social Program Management enforces referential integrity and foreign keys are generated to test integrity. The use of declarative referential integrity (foreign keys) in a production system impacts the performance of that system and is therefore not supported.

Java code

Many Java code artifacts 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>` directories). A CodePackage might be empty or there might be a number of CodePackage elements within each other (for example, `<ProjectPackage>/intf` and `<ProjectPackage>/<CodePackageA>/<CodePackageB>/intf` may both be generated depending on the options that are chosen).

- `intf` - Defines the interface for the objects that are modeled.
- `fact` - Provides factory wrappers for the objects that are identified in bizinterface.
- `base` - Ensures that the developer provides implementations for those methods, which must be hand-crafted.
- `remote` - Provides remote interfaces for the objects that can be exposed to the client.
- `struct` - Defines the classes which model parameters between the objects.
- `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. This directory also contains a `rdoindex.properties` file that contains a listing of all the generated objects.

Deployment artifacts

A number of deployment artifacts are also generated by the model build. This section does not attempt to detail the meaning of these files but introduces the files and their locations. These artifacts are used when you build an application `.ear` file, they are generated according to the following categories:

- IBM-specific metadata:

  Supports deployment on IBM WebSphere® Application Server. These artifacts are generated into the `/build/ear/WAS` directory and contain the necessary `.xml` files.
- Oracle-specific metadata:

  Supports deployment on WebLogic. These artifacts are generated into the `/build/ear/WLS` directory and contain the necessary `.xml` files.

Kubernetes WebSphere Application Server Liberty ignores user-defined bindings, so a Liberty-specific descriptor is not needed.

Web service artifacts

A number of web service artifacts are generated. This section does not attempt to detail the meaning of these files but introduces the files and their locations. These artifacts are used when you build an `.ear` file that supports web service invocation. The artifacts consist of special `structs` that contain web service conversion routines and are generated into the `/build/svr/gen/` 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 IBM Cúram 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 IBM Cúram 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, irrespective of how many references to that string value exist.

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.

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
```

Utility Targets

A number of utility Ant targets are provided which are not necessary to build a server. Useful targets are as follows:

Database targets

database
    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 "Rules Targets" on page 24).

prepare.application.data
    Run this target after the database target is run and before starting the application server for the first time.

    Note: Failing to run in this sequence results in transaction timeouts during first login and a failure to initialize and access the application.

    Whenever you rerun the database target is rerun (for example, in a development environment) you must also rerun **prepare.application.data**.

extractdata
    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 "Data Manager" on page 37.

reloadextracteddata
    Reloads data that was extracted using the **extractdata** command back onto the database. This target depends on the **insertextracteddata** and the **extracteddata** targets existing in the

datamanager_config.xml file. If these targets do not exist in your datamanager_config.xml file, the default datamanager_config.xml file should be used as a reference for adding them.

checksql
Validates the hand-crafted SQL and test data against the actual database. If you do not run **checksql**, syntactical (and semantic) mistakes in hand-crafted SQL are not 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 you must run the **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 "SQL Checker" on page 61.

foreignkeycheck
Enabling foreign keys on the database could degrade performance. Enabling foreign keys could also violate referential integrity as a result of program bugs or manual intervention by a Database Administrator. **foreignkeycheck** validates that the Referential Integrity has not been violated 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.

Password targets

encrypt
Encrypts a plain-text password (e.g. for curam.db.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. See the *Cúram Security Handbook* for more information regarding cryptographic settings for encrypted passwords. The mandatory argument password that denotes the plain-text password has to be specified when invoking the target.

For example:

```
encrypt -Dpassword=passw0rd
```

in this example, the output of this execution is displaying the encrypted password in the console.

The **encrypt** target can be also used to encrypt a plain-text password for a property in a properties file using two optional arguments, property and properties.file.path respectively.

For example:

```
encrypt -Dpassword=passw0rd -Dproperty=curam.db.password
-Dproperties.file.path=c:\bootstrap.properties
```

In this example, the output of this execution is the property value is updated with the encrypted password in the specified properties file. If the property does not exist, then the specified property will be added to the properties file along with the encrypted password as the property value.

digest
Digests a plain-text user password. When you change cryptographic digest settings, for internal and external Cúram users, you may need to update digested password values in DMX (e.g. USERS.DMX) and SQL files for passwords to be stored on the database. To make these updates you will need the new digest password values, which you can obtain via this target. Care should be used in creating these passwords and should only be done for test users. See the *Cúram Security Handbook* for more information regarding cryptographic settings for digested passwords.

Test targets

test
Executes the tests associated with the application.

---

[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.

- If Clover is available a code coverage report can also be generated. More details on the usage of Clover are available in "Clover Targets" on page 23.
- 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.
- You can exclude or include sets of tests while running the `test` target. To Exclude or include tests, copy the `ExcludeTests.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
Examines 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:

- 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. for DB2: buffer pools, tablespaces, etc.; and for Oracle: privileges for the Cúram user. If DB2 is remote the configuration check for LOCKSIZETIMEOUT will fail. This check can be bypassed by setting the `db2.is.remote` property.
- 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.
- `Kubernetes` WLP_HOME system environment variable with the value set to the WebSphere Liberty installation directory.
- Ant variables ANT_HOME and ANT_OPTS
- server and client environment variables

configreport
Creates a `config_report.zip` file, which contains information to assist with diagnostics gathering, and can be used if remote support is required. The file is created in the <CURAM_DIR>/EJBServer directory, and contains a copy of:

- Environment settings for Cúram specific and system environment variables, and software versions on the machine.
- The installer logs, which also provide the version of Cúram being used.
- Properties files - all properties files that are located in the <CURAM_DIR>/EJBServer/project/ `properties` directory.

- A copy of the `deployment_packaging.xml` file.
- The output from the **configtest** build target, as detailed above.
- The log files from the application server being used. Note that these log files will only be copied if:
  - the application server is installed on the same machine as Cúram.
  - the application is running on a standalone server.
  - the default location where the log files are written to has not been changed.

Properties targets

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 "Application Properties" on page 31 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 in <SERVER_DIR>/build/propertiesextractor/

mergeuserpreferenceproperties
Merges the user preference properties files.

Javadoc targets

javadoc
Generates the Java Documentation (JavaDoc) from the application. To produce useful JavaDoc, your developers must place comments in the model as well as in the code.

apijavadoc
Generates the javadoc for black/grey box components based on the javadoc.properties files.

Other targets

clean
Deletes all the generated and compiled files to ensure all generated and compiled artifacts 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.

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

deprecationreport
The command-line Java compiler deprecation warnings are extended to apply to certain Cúram builds and validations. This helps to 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 "Deprecation" on page 111 for more information.

release
Gathers all the files that are needed to run IBM Cúram Social Program Management on another machine in the <SERVER_DIR>/release directory. The `release` target is used to build the application for a target platform, for example, to build the application in Windows for deployment on IBM z/OS®, or to move the application between machines. If you move the `release` directory to another machine, you must place the `Bootstrap.properties` file and the `AppServer.properties` file in a `release/project/properties` directory. Also, before you use any of the scripts, you must set the following environment variables:

- Set **SERVER_DIR** to the release directory.
- Set **SERVER_MODEL_NAME** to the name of the application model.

- Set **CURAMSDEJ** to the location of the SDEJ.
- Set **SERVER_COMPONENT_ORDER** in your target environment where you plan to work with the resulting `release` directory. The value must be the same as the value that is used in your source environment.

The appropriate `SetEnvironment.bat` and `SetEnvironment.sh` files are also generated into the release deliverable for a specified environment. The files are in the root directory of the release delivery and contain commands to set the standard environment variables, including the environment variables for the development environment, the locale list, and component orders. The files that are copied are:

- Ant build files
- Project JAR files
- DDL files
- SQL files
- Code tables files
- Batch launcher
- Data manager
- Application EAR files
- XML server files

Run the following command to generate a compressed zip file for the release:

```
build release -Dcreate.zip=true
```

model
Extracts 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 "Statistics" on page 59.

supplement
Compiles and jars all the Java files contained within any supplementary directory specified by the `-Dsupplement=<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.

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.

Rules Targets

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:

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 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 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 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 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 'ruleset.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.

IEG Targets

The **validateieg2scripts** command validates the intelligent evidence gathering (IEG) scripts in the specified directory.

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` directory 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**

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 and Cúram events (See "Events and event handlers" on page 136) 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.

Deployment Targets

There are a number of deployment targets that 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.

Note: ☐⊘Generic Search Server is deprecated. For more information, see Deprecated features

Application server configuration targets that apply to all application servers:
  `configure` Automatically configures the application server.

  `configurewebserverplugin` Automatically configures the application server with a web server plug-in.

EAR file builds for each application server:
  `websphereEAR` produces an `.ear` file that can be deployed on WebSphere Application Server.

  `websphereWebServices` produces an `.ear` file that can be deployed on WebSphere Application Server that to support Web Services invocation.

  `weblogicEAR` produces an `.ear` file that can be deployed on WebLogic Server.

  `weblogicWebServices` produces an `.ear` file that can be deployed on WebLogic Server to support Web Services invocation.

---

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

Kubernetes **libertyEAR** produces an .ear file that can be deployed on WebSphere Liberty.

Kubernetes **libertyWebServices** produces an .ear file that can be deployed on WebSphere Liberty to support Web Services invocation.

precompilejsp precompiles all JSPs in the specified .ear file (applies to WebSphere Application Server and WebLogic Server only).

EAR file install and uninstall (applies to all application servers):
installapp installs and starts a specified EJB application.

Note: The EAR file (Curam.ear) that contains the server module must be deployed before you install any other (client-only) EAR files.

uninstallapp stops and uninstalls the specified application.

Application preparation (applies to all application servers):
prepare.application.data Run this target after the database target is run and before starting the application server for the first time.

Note: Failing to run in this sequence results in transaction timeouts during first login and a failure to initialize and access the application.

Whenever you rerun the database target is rerun (for example, in a development environment) you must also rerun**prepare.application.data**.

Application server control (applies to all application servers):
startserver starts an application server.

restartserver restarts an application server.

stopserver stops an application server.

Deprecated targets
weblogicEARGSS builds the GSS .ear file for WebLogic Server.

websphereEARGSS builds the GSS .ear file for WebSphere Application Server.

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".

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 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>
```

Figure 5. Before/After Target usage

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.

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 **websphereEAR** instead of **serverbuild.websphereEAR**.

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**
- Kubernetes **serverbuild.libertyEAR**
- Kubernetes **serverbuild.libertyWebServices**

Application Targets

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

BI App

The **biapp.configure.birtviewer** target configures the Business Intelligence and Reporting Tools (BIRT) Viewer application for IBM WebSphere Application Server after the BIRT EAR file is installed. Use the following Ant property specifications when you run the target:

-Dserver.name=server.name - The name of the server to deploy the application into.

-Dapplication.name=application.name - The name of the BIRT Viewer application.

Kubernetes The biapp.configure.birtviewer target does not apply to Kubernetes deployments. For Kubernetes deployments, BIRT deployment is managed within a Helm chart. You can find example Helm charts in the following GitHub repo: https://github.com/IBM/spm-kubernetes/tree/main/helm-charts/apps. For more information about preparing Helm charts, see Preparing Helm charts.

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.

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** - INITIALAPPRESOURCE.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.

Cúram Configuration Settings

Configure environment variables in your environment to configure support for multiple time zones, and to configure dates and date/time behavior.

IBM Cúram Social Program Management application properties are set in `application.prx` and `bootstrap.properties`. You can configure support for multiple time zones, and configure dates and date/time behavior.

Application Properties

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

Application prx

Use this information to understand the properties that are used in the `Application.prx` file when you run a Cúram application.

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

The properties that are defined in `Application.prx` can be "dynamic" or "static". Dynamic properties take effect immediately if changed and published by using the administration interface during run time. Modifying static properties has 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>
```

Figure 6. PRX entry

The file is organized as follows:

Property Element
A property element is used for each property.

> **Name Attribute**
> Attribute specifying the name of the property.

> **Dynamic Attribute**
> Indicator whether a change to the property value requires 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 that is 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.

How to merge an application `prx` file
Use this information to understand how to merge `prx` files into your Cúram application.

Prx files are in the `/properties` directory of a component and the root `/project/properties` directory. The IBM Cúram Social Program Management Platform includes a set of `prx` files. These files might be overridden by placing new `prx` files in the SERVER_DIR/components/<custom>/ `properties` directory, where *<custom>* is any new directory that is created under components that conforms to the same directory structure as `components/core`. This mechanism avoids the necessity to change directly the initial, unmodified 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 leftmost has the highest priority, and the rightmost the lowest.

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

Figure 7. SERVER_COMPONENT_ORDER example

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

Note: After changing the component precedence order in SERVER_COMPONENT_ORDER, it is necessary to initiate a reinsert of the merged properties. This action is done by calling **build insertproperties**.

When you merge `prx` files, the components that are listed in the SERVER_COMPONENT_ORDER are taken in order of highest to lowest priority. In the preceding example, the `Application.prx` file from the `sample` component is merged with the `Application.prx` 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 it is shown in the preceding example, a more important main/source `Application.prx` file exists, and a file that is being merged into it. The second file is called the merge file in the following sections.

An `Application.prx` file can be customized by:

- Adding a property that provides mandatory property values.
- Overriding an existing properties description.
- Overriding an existing properties display name.
- Override an existing properties value or default value.
- Adding a 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` file 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 always are overwritten by the `Application.prx` file in the component with the highest precedence order. The main `Application.prx` example file and the merge `Application.prx` file that follow 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>
```

Figure 8. 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>
```

Figure 9. 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>
```

Figure 10. Resulting `Application.prx` File

Bootstrap.properties

Use this information to learn about the `Bootstrap.properties` file that contains the minimum set of properties necessary for obtaining a connection to the database.

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

The `Bootstrap.properties` file also might 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 by using the administration interface.

Note: Properties that are defined in the following are cached: `Application.prx`, `Bootstrap.properties`, and Java System properties at run time. Properties that are defined in `Application.prx` are loaded into the database and can be updated at run time by 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 by using the Application Server or by using **java.lang.System.setProperty()**) and also in `Application.prx` **curam.util.resources.Configuration.getProperty()**, the value of the property that is defined in the Java system properties always is returned when it uses the `Application.prx` and `Bootstrap.properties`, the value of the property in `Application.prx` is what takes 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
```

Figure 11. `Bootstrap.properties`

An automatically generated version of `Bootstrap.properties` is packed in the Enterprise Archive (EAR) when the EAR file is built. This file chooses its properties from the default `Bootstrap.properties` and is extended with extra properties that are related to the Application Server being used.

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

Figure 12. `Bootstrap.properties` in an EAR file

Note: The EAR file cannot be built for H2 database.[3]

Support for multiple time zones

Use this information to understand how to enable multiple time zone support in your Cúram application.

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

Only Date/Times 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 Date/Time field is to be used. An example of a date where the time zone is not relevant is someone's date of birth. It does not vary regardless of the time zone that person was born. An example of a date where the time zone is relevant is the current date. This DAT is different for two users who are working either side of the International Date Line, in this case a Date/Time 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; that is, 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 Date/Times passed to it from the server into the user's preferred time zone and also to convert all Date/Times 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()`.

---

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

Some of the Java supported time zones that are returned by
`java.util.TimeZone.getAvailableIDs()` method are in the following list:

- *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 more information on server time zone configuration, see the Time Zone Configuration chapter in the *Cúram Deployment Guide* for the appropriate application server.

Dates and date/times in Cúram

Use this information to understand the behavior of dates and time/dates in your Cúram application.

How the Cúram application describes the behavior of dates and date/times is covered here.

Look at these examples:

- The server is in the Greenwich mean time (GMT)time zone. A user is in time zone GMT -01. At 15:00 GMT the user registers a new person, and the server-side processing time stamps a resulting database record with the time 15:00. Twenty seconds later the user initiates a query and sees the time stamp displayed in the client user interface as 14:00. The user's clock is showing 14:00:20 - the new record's time stamp 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 that 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 afterward is 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 the information is converted 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 that is 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 its current local date/time to a date.
- The user requested a list of transactions with a start date of day 1. Because this information 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 that is being searched on is itself date/time. Users need to be aware that the current business day might not be the same date as the date in their local time zone. Fortunately, such situations likely are to be rare.

Data Manager

Use the Data Manager tool to create a database that contains a set of initial data, test data or both.

The Data Manager is based around database independent `.xml` files. Any setup that is done by a developer can be applied to any of the supported databases.

Intended Data Manager process

The Data Manager helps the 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. Data Base Administration (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 `BUFFERPOOL`.

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 that is required 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 similar commands, in order for the wanted SQL to be generated.

This behavior is due to a number of factors. The modeling tools are unaware 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 only to reset the database to a known state.

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*. Specific Cúram configuration is required when using MBCS data with DB2 or DB2 for z/OS so that the Data Manager functions compatibly. This configuration is enabled for Cúram as it is configured initially.

Cúram support for MBCS data with DB2 and DB2 for z/OS is enabled in its initial configuration to ensure error-free operation for users with languages that require MBCS data and for users who find they require MBCS data when copying or 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 that is required when MBCS data is used. This procedure 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 might not be appropriate in the following circumstances:

- If your data requirements do not necessitate the maximum expansion (as explained as follows) you can reduce the amount of expansion.
- If you are using only single-byte data (a Western language, such as English) and not using any other MBCS data (for example, by a browser copy or paste), disable multi-byte expansion support. However, this procedure is not recommended due to the likelihood of MBCS data that is introduced from external sources (for example, browser copy or paste) and later causing errors.

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` property in `Bootstrap.properties`. These properties are described in .

To be certain of not receiving any processing errors when processing MBCS data, the default expansion factor is set to the maximum. However, for many languages and data profiles it is unlikely that every database column character would require MBCS data or that all characters would require the maximum size of 4 bytes. A cost is associated with using the maximum expansion factor in terms of disk space used, network processor usage, memory usage, buffer pool performance, CPU usage, and so on. Therefore, it is best to use an expansion factor that balances resource usage and performance while avoiding or minimizing the possibility of application errors caused by data overruns. There are no strict rules for achieving this balance between resource usage and the possibility of application errors, but considerations, such as those that follow, 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 vary. For instance, if you are using English with only a few special characters (for example, smart quotation marks), you require little expansion. Or, if you are using a language that shares the Latin alphabet with some additional characters (for example, German), then you need more space for MBCS data. A language (for example, Chinese) that uses characters at the higher end of the Unicode range requires more space per character, which needs to be tempered by the number of characters that are required per word; that is, the language might convey more information in each character than a typical Latin alphabetic character. In other words, consider the average bytes required per character, word, and so on. Typically this average is only a rough estimate because, as studies show, character usage can vary depending on a number of factors; for example, data context, data that is more numeric (phone numbers), versus more textual data (names) and even free-form comments. So, some additional safety factor needs to be considered in choosing your expansion factor.

You also are able to control the expansion factor at a more fine-grained level in the modeling environment by specifying the `Multibyte_Expansion_Factor` option for a string domain, an entity string attribute,

or both, which might be appropriate for your customizations. For more information, see the *Cúram Modeling Reference Guide* for setting these options. You might 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, and so on, that can be exceeded by large expansion factors.

For more information on these limits, refer to the relevant DB2 or DB2 for z/OS SQL reference.

Invocation

The Data Manager is started by running a build command of **build database**.

DB2 development database optimization tip: During iterative development with DB2 on distributed operating systems, the dropping and creation of tables that are performed during the **build database** target can be optimized to run quicker by running the following script once per database:

ant -f %CURAMSDEJ%\util\db2_optimizedbrecreation.xml

Internally this command runs:

ALTER TABLESPACE USERSPACE1 DROPPED TABLE RECOVERY OFF;

ALTER TABLESPACE CURAM_L DROPPED TABLE RECOVERY OFF;

This step is not to be not be taken on a production database.

Database artifacts

Use this information to understand how the Data Manager works and how to use it and handcrafted artifacts to set up the database.

The Data Manager uses generated and handcrafted artifacts to set up the database. Those handcrafted artifacts are explained in the pages of the following sections.

- Data Definition XML files - The `.xml` files describe the database tables and the constraints that are placed on them.
- Data Contents Data Mining Extensions (DMX) files - In addition to creating the tables on the database, the Data Manager allows the developer to specify sample and test data that is to be placed on the database. The format of the `.DMX` file is introduced in "Data contents DMX files" on page 41. The developer typically edits this file by using a standard XML editor.
  - The `Table` element
  - How to customize a DMX file
  - Retrieving values from DMX files
  - Validation of DMX files
  - Tracing information for the DMX merging process

Data definition XML files

The `.xml` files describe the database tables and the constraints that are placed on them. For an introduction to these files, see the related information.

The code example that follows shows a sample table definition. An entity can have any number of `attribute` elements. Not all elements 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>
```

Figure 13. Table definitions

The code example that follows shows a sample foreign key constraint. Any number of `key`, `association`, and `foreignkeypair` elements are possible.

Note: If foreign keys are applied to a DB2 for z/OS database by the Data Manager, manual intervention is required to move the tables from the `check_pending` state. Consult with your local Database Administrator (DBA) to resolve this issue.

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

Figure 14. Foreign key constraints

The code example that follows shows a sample primary key constraint. Any number of `key` and `attribute` elements can be included.

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

Figure 15. Primary key constraints

The code example that follows shows a sample index constraint. Any number of `index` and `indexattribute` elements can be included.

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

Figure 16. Index constraints

The code example that follows shows a sample Unique Constraint. Any number of `constraint`, `association`, and `attribute` elements can be included as necessary.

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

Figure 17. Unique constraints

The code example that follows shows a sample of the metadata that is generated to support the batch processes that were modeled by the developer. Any number of batch processes that have any number of parameters can be included.

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

Figure 18. Batch metadata

The code example that follows shows a sample of the metadata that is generated to support the security
that was modeled by the developer. Any number of function identifiers (FIDs) can be included.

```
<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>
```

Figure 19. Security metadata

The code example that follows shows a sample of the metadata that is generated to support the field
level security that was modeled by the developer. Any number of fields that are returned can be included.

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

Figure 20. Field level security metadata

Related concepts
Under the Hood

Data contents DMX files
In addition to creating the tables on the database, the Data Manager allows the developer to specify
sample and test data to be placed on the database.

In addition to creating the tables on the database, the Data Manager allows the developer to specify
sample and test data to be placed on the database. The developer typically edits this file by 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>
```

Figure 21. Data contents file

The data contents Data Mining Extensions (DMX) file is made up of a number of elements that are described in the following sections, some of these elements and attributes are necessary to enable customization of DMX files, described in further detail in "How to customize a DMX file" on page 44.

The `table` element
Use this information to understand the `table` element, along with a description of its attributes and default settings.

The <table> element has the following components:

Table 8. Components of the `table` element

Attribute name	Required	Default	Description
name	Yes	None	Specifies the name of the database table.
override	No	false	Used to customize or completely override existing DMX files from within a component lower down in the SERVER_COMPONENT_ORDER.

The <column> element

The <column> element has the following attributes:

Table 9. Attributes of the `column` element

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 13 on page 43 describes the `type` that a `column` can be set to.
encoding	No	UTF-8	Specifies the CLOB data file encoding type. Check "LOB Manager" on page 59.

The <row> element

The <row> element has the following attributes:

Table 10. Attributes of the `row` element

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 is applicable to all locales. If present, this attribute must be set to a comma-separated list of locales, ensuring that there are no spaces between each locale. The following example indicates the <row> is applicable for the en and en_US locales: <row locales="en,en_US">.

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

The <attribute> element

The <attribute> element has the following attribute:

Table 11. Attributes of the `attribute` element

Attribute Name	Required	Default	Description
name	Yes	None	Specifies the name of the column.
encoding	No	UTF-8	Specifies the CLOB data file encoding type. Check "LOB Manager" on page 59.

Note: If the number of attributes that are defined for a row does not match the number of columns defined the Data Mining Extensions (DMX) processing fails.

Note: Therefore, 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. The ordering of the attributes should match the ordering of the columns.

The `attribute` element has a required subelement: *value*.

The *<value>* Element

The *<value>* element is the value to be inserted into the column for this row. For a Binary Large Object Block (BLOB), the value is a pointer to a file. To be meaningful, the *name* attribute of the `attribute` element takes its value from one of the `column` elements' *name* attributes within the same DMX file. Ordering also is important as when the database is being built. Database columns are 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 13 on page 43, describes the relation between the column *type* and `attribute` value.

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

Table 12. Attributes of the value element

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.

Important: The primary key or the composite key for a record never must 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.

Table 13. Attribute values

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*.

Table 13. Attribute values (continued)	
Column Type	**Attribute Value**
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.

How to customize a DMX file
Use this information to understand how to customize DMX files. You also can use this information to learn how to modify the elements of a DMX file and add DMX files to new components of your Cúram application.

The Data Manager processing allows for the customization of Data Mining Extensions (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 or attribute level, and completely override a DMX file. This process allows for DMX files that are included with the Cúram application to be customized easily 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 without customization, 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 change directly the uncustomized 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 that contains a comma-separated list of component names, the leftmost having the highest priority.

Note: It is possible that more than one DMX file contains data for a particular database table. As the merging of DMX files is based on file names, it might be necessary to customize multiple DMX files to achieve a wanted data customization for an individual entity.

Only DMX files that are placed within the structure as shown in the previous example are included in the merging process for DMX files. If subdirectories are used within the `initial`, demo and `test` directories, then these directories are not 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 merging DMX files

DMX files are merged based on precedence order. A more important main or source DMX file always exists, and other files are merged into it. The second file is called the merge file in the following sections.

The merging rules that are described in the list that follows are applied to decide whether the rows, attributes, or DMX files need to be merged into the new DMX file.

- A DMX file is considered for merging if the new DMX file does not have the `override` attribute on the `<table>` element set to `true`.
- A `<row>` is inserted into the new DMX file if it is determined, by using the primary key information for the record, that the `<row>` is not present already in the new file.
- If a `<row>` exists in the new DMX file and the `remove` attribute is set to `true`, then no merging occurs. If the `remove` attribute is set to `false` or is not present, then the attribute values for that row are considered for merging.
    - If the `<value>` element does not exist in the new DMX file, then the `<value>` element is copied.
    - If the `<value>` contains a different locale, then this `<value>` entry is copied into the new file. The locale is specified by the `language` and `country` attributes on the `<value>` element.

All examples that follow assume that custom is before core in the SERVER_COMPONENT_ORDER.

Example 1 that follows illustrates how merging works when the process uses the `<table>` level `override` attribute. To use the `override` attribute, copy the contents of the existing DMX file; that is, 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 element indicates that only DMX files in this *<custom>* component or in a component higher up in the SERVER_COMPONENT_ORDER is included in the merged DMX file output produced from the Data Manager processing.

```xml
<?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>
```

Figure 22. Example 1 - Core DMX file

```xml
<?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>
```

Figure 23. Example 1 - Custom DMX file

```xml
<?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>
```

Figure 24. 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 overriding completely the core DMX file through the following variable: *<table override="true">*, resulting in all entries in the core file to be excluded.

Example 2 that follows 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
<?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>
```

Figure 25. Example 2 - Core DMX file

```xml
<?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>
```

Figure 26. 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>
```

Figure 27. Example 2 - Resulting 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 the application is processing the merged DMX file in Example 2, the <row> where the *CONCERNID* is set to 2 are not included when it creates the SQL insert statements, thus ensuring no entry exists on the database for this <row>.

Example 3 that follows 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
<?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>
```

Figure 28. 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
<?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>
```

Figure 29. Example 3 - Custom DMX file

```xml
<?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>
```

Figure 30. Example 3 - Resulting merge file

In Example 3 shown previously, 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; that is, a merge of both core and custom *<value>* elements.

Example 4 that follows 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>
```

Figure 31. 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>
```

Figure 32. Example 4 - Custom DMX file

```xml
<?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>
```

Figure 33. Example 4 - Resulting merge file

In Example 4 shown previously, the value for the `locales` attribute is taken from the row in the component that is higher up in the SERVER_COMPONENT_ORDER; that is, the custom component.

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

Retrieving values from DMX files for database insertion

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

Data Mining Extensions (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 data that is specified in DMX files is inserted 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 overrides the setting in the `Bootstrap.properties` file. This property must be set to a valid locale; that is, 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` file 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 value takes precedence over the attribute level.

For example:

```
<?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 entry is 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 that is 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>; that is, 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 applicable only to the attribute <value> element; that is, it is not applicable to the <row> `locales` attribute. After a <value> is found and no direct match is found 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 element is not found, then the *country* attribute is removed and the search looks for a <value> where the *language* attribute matches, if this attribute is not found, then the search looks for a <value> that does not have the *language* and *country* attributes set; that is, a default match. If this attribute 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>
```

Figure 34. Locale Fallback Example

In "Retrieving values from DMX files for database insertion" on page 53, assume the `curam.dmx.locale` is set to *en*. The following variables are set for each attribute:

- *ELEMENTTYPE - EN_TYPE* is the value that is inserted onto the database for this attribute, as this element 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; that is, the default value, as this variable does not exist, null is inserted for this attribute.

Validation of DMX files

All Data Mining Extensions (DMX) files in `%SERVER_DIR%/components/componentName/data` directories are validated against a DMX schema file when the **build database** target is run. This schema file is in `%CURAMSDEJ%/lib/DMX.xsd`. For any DMX file that is not in the correct format, a warning is 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 is 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 variable is set to *true*, the warnings are treated as errors and the build database fails.

Tracing Information for the DMX Merging Process

It is possible to turn on tracing for the Data Mining Extensions (DMX) merging process. This action can assist in debugging any issues that might 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
```

Figure 35. Set tracing for DMX files.

Therefore, when set to *true,* this property outputs a large amount of data to the console to be used for debugging.

Database Object Naming

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

(deprecated) 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 necessitated the feature originally. Consequently, it no longer is necessary to use this feature and it has been removed from the product documentation. If you still require this feature, contact your IBM support representative for the information that was available previously in this document.

Primary key indices

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

If required, a prefix can be specified for the primary key index name by using the generator command line option **-primarykeyindexprefix**. For example, by setting the property extra.generator.options=-primarykeyindexprefix PI_ in Bootstrap.properties results 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 encounter an error during SQL processing.

Primary key constraints

Use this information to understand the constraints for assigning options to primary keys.

By default, the generated Data Definition Language (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 to 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 by using the **-primarykeyconstraintprefix** command line option. If the constraint name length is greater than the SQL identifier limit supported by your database, you encounter an error during SQL processing.

Automatic index generation

An automatic index creation feature supplements the manual addition of indexes to the model. The automatic index creation feature is enabled by default.

The automatic index creation feature discovers non-indexed fields that are included in the following operation types:

- Readmulti
- NsModify
- NsRead
- NsReadmulti
- NsRemove

If the non-indexed field is based on one of the domain types in the following list, excluding codetable domains and strings that are longer than 255 characters, an index is generated by the model generation process for the non-indexed field:

- SVR_INT64
- SVR_DATE
- SVR_DATETIME
- SVR_STRING

To facilitate repeatable naming, the generated index is named IND_*Hashcode of Table + FieldName.*

Disabling the automatic index creation feature

To disable the automatic index creation feature, specify the following command line parameter in the server build:

```
-Dgenerator.options=-disableautoindexgeneration
```

Tablespaces

Use this information to understand the exceptions when table spaces are not created automatically during table creation.

Note: The following instructions are specific to DB2 for z/OS.

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

- The `tablespace` that is named by the curam.db.zos.32ktablespace property is created explicitly by the **datamanager** and tables that exceed the 4K row limit are placed in this `tablespace`.
- Tablespaces 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 takes the default setting.

When the operation uses the `Tablespace.properties` file, the format of the entries is:

```
tablename=tablespacename
```

Comments are specified by the # character in column one.

Note: If the `tablespaceTablespace.properties`, then this `tablespace` is used over the one defined in the property `curam.db.zos.32ktablespace`.

Note: When you are 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 might also occur in other contexts. To avoid these errors, you take one of the following actions:

- Ensure that your Cúram default 32K `tablespace` is segmented (**SEGSIZE**. For more information, see *DB2 Universal Database for z/OS SQL Reference Version 8*.
- Explicitly define `tablespaces` for each Cúram table that defaults to the 32K tablespace (for example, SELECT * FROM SYSIBM.SYSTABLESPACE WHERE NAME = <curam.db.zos.32ktablespace value>) and assign each table to a specific tablespace by using the `Tablespaces.properties` file. (This complication is not an issue when you use 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 is changed as follows: "If the table space was implicitly created, the corresponding enforcing index 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 condition might not be the case. The symptom of this issue is an SQLCODE -551 error on a DROP INDEX statement that follows 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
```

Data Manager configuration

Use this information to understand how to use the Data Manager to properly set up database configuration.

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

• IBM Cúram Server Development Environment (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 wants to complete.

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

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

The structure of `datamanager_config.xml` is shown in the following example:

```
<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>
```

Figure 36. Data Manager configuration

The file is organized as follows:

Target Tag
    This tag has a name attribute that specifies the name of the target and a set of associated entry tags.

Entry Tag
    This tag has three attributes that are 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 attribute specifies whether the file is an SQL script, a `.DMX` file, or an `.xml` file.

    Base Attribute
        This attribute specifies the system-dependent offset of the file on the local computer. It can be specified as one of `basedir` (the directory above the Data Manager) or `sdescripts` (the location of the SDEJ installation).

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

The `datamanager_config.xml` file is used when the application is running the **build database** target. When this target is run, composite targets that are 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 that specifies the composite target to be called. For example, to call the initial composite target, the following command might be run:

```
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 block of code 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>
```

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 running 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 are to be extracted; for example, **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 Character Long Object (CLOB) or Binary Large Object Block (BLOB) also are extracted and stored in a file that is based on the naming format: `<tablename><rownumber>`.

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. This guess often can lead to performance issues, including unnecessary deadlock and timeout exceptions. The **runstatistics** target is available to gather these necessary statistics on the database and is run against all Cúram database tables.

Note: The **runstatistics** target is not supported by DB2 for z/OS due to the architectural differences of this application. Consult with your local database administrator in regard to engaging the equivalent DB2 for z/OS functions.

LOB Manager

The Large Object Block (LOB) Manager is part of the Data Manager that enables Character Long Object (CLOB) and Binary Large Object Block (BLOB) to be loaded onto the database.

In the data contents, file BLOB and CLOB fields are handled differently to a degree from other fields, in that the `value` element does not contain the literal data, but instead contains a reference to a file that contains the data.

The "LOB Manager" on page 59, illustrates how a table with a numeric and BLOB column can be populated with one record that uses 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>
```

Figure 37. BLOB Data Contents File

To load BLOBs, the LOB Manager can be used only on tables for which the primary key fields are known. This restriction 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 using its primary key.

The "LOB Manager" on page 59, illustrates how a table with a numeric and CLOB column can be populated with one record by using a character data file from disk. Here, the CLOB data file is encoded with UTF-16 format, and this information 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>
```

Figure 38. CLOB Data Contents File

The "LOB Manager" on page 59, illustrates how a table with a numeric and CLOB column can be populated with two records by using the character data files from disk. Here, if all the CLOB data files are encoded in UTF-16 format, then this information can be specified at the column level, by using encoding attribute, so all the rows for CLOB type use the same encoding type of that column. To override action this for a single row, the encoding type can be specified as in previous example at the 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>
```

Figure 39. CLOB Data Contents File in encoded format

The LOB manager identifies primary keys by using the `datamanager_config.xml` file, so this file must contain a reference to the generated `_PrimaryKeys.xml` relating to the table that contains the LOB.

SQL Checker

Use this information to understand how the SQL Checker can validate Java Data Base Connectivity (JDBC) SQL statements.

The IBM Cúram Social Program ManagementServer Development Environment (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.

Under the Hood

The SQL checker is invoked by an Ant target and generates a simple Java program that uses SQL For Java (SQLJ) rather than Java Data Base Connectivity (JDBC). This program is generated into `/build/sqlcheck/SQLJTemp.sql.` This Java program contains all the elements that should be checked, namely the handcrafted 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 also can check the contents of the model for database portability. This is useful in situations where primary development is against one type 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 handcrafted 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.

Limitations

The SQL Checker reduces 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 SQL For Java (SQLJ) compiler that it invokes. Any syntactical or semantic errors that are not reported by the compiler will not be reported by the SQL Checker.

Portability Warnings

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

Limitation with H2

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

SDEJ development and application programming interfaces

Use this information to develop compliantly with Application Programming Interfaces (APIs) in the Server Development Environment (SDEJ). Learn about how to use Eclipse, how logging and tracing works, and what "deprecation" means in the context of IBM Cúram Social Program Management development.

Eclipse

Use this information to understand how to use the Eclipse Integrated Development Environment (IDE) with the Cúram Server Development Environment (SDEJ). Read about some relevant aspects of Eclipse and some tips for Eclipse usage.

Eclipse is the underlying technology for these applications:

• IBM Rational Application Developer for IBM WebSphere Application Server
• IBM Rational Software Architect Designer
• IBM Rational Software Architect Designer for WebSphere Application Server

The term *Eclipse*, which is used throughout this information, applies to all supported tooling based on Eclipse, such as Rational Application Developer.

For more information about general features or usage of Eclipse, see http://www.eclipse.org/.

Cúram projects to import into Eclipse

Use this information to learn about the four Cúram projects that need to be imported into Eclipse.

Four projects are provided that need to be imported into Eclipse:

Table 14. Transaction settings		
Project Name	**File System directory**	**Contents**
CuramSDEJ	CuramSDEJ	The Server Development libraries.
CuramCDEJ	CuramCDEJ	The Client Development libraries, depends on the CuramSDEJ project.
EJBServer	EJBServer	The Cúram Server application, depends on the CuramSDEJ project.
Curam	webclient	The Cúram Client application, depends on the CuramCDEJ project.

Dependencies allow for exposed compressed 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 needs to be done within the `EJBServer` and `Curam` projects.

Eclipse configuration files

Use this information to learn about 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 preferences and settings at a project level is that this configuration, which forms files and entries in a.Settings folder under the project, can be distributed with the project in a team environment.

The configuration is maintained by right-clicking on a project within the Project Explorer view in Eclipse and selecting `Properties`.

.project file
Use this information to understand the `.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 an 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.

The .classpath file
Use this information to understand how the `.classpath` file maintains the project's source and target references for Java compilation and compressed file or project dependencies

The `.classpath` maintains the project's source and target references for Java compilation and compressed file 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 the **Java Build Path** page.

Optionally, Access Rules and JavaDoc references can be configured on JAR files. Access Rules are discussed further in "Access Rules option" on page 64.

Eclipse .classpath generation
Use this information to understand about Eclipse `.classpath` file generation.

The Eclipse `.classpath` files for the `EJBServer` and `webclient` projects can be generated from a build target - **build createClasspaths** that can be run from the `EJBServer` directory. This action allows for the class paths to tailor to the contents in your environment and avoids the need for manual maintenance of this file.

It is advised that you add the invocation of this target to your default build invocation wrapper to ensure that it gets run with each build. Example in the `EJBServer\.build.bat` file. The class path is not regenerated unless changes are made in your environment.

The class paths are formed from:

- `source` directories under the `EJBServer\components` directories
- `tests` directories under the `EJBServer\components` directories
- JAR files in the `lib` directories under the `EJBServer\components` directories
- `javasource` directories under the `webclient\components` directories
- JAR files under the `webclient\components` directories
- Standard build output directories
- JAR files on the *PRE_CLASSPATH*, *POST_CLASSPATH*, and *J2EE_JAR* environment variables
- `CuramCDEJ` and `CuramCDEJ` project references.

.settings directory

Use this information to understand the `.settings` directory folder.

The `.settings` folder contains a number of the other preferences that can be maintained at the project level; for example, compiler warning/error levels or code style settings. The preference pages that offer 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.

Access Rules option

Use this information to understand the Access Rules option and how it works with compressed files in an Eclipse project.

The `Access Rules` option allows compressed files within an Eclipse project `.classpath` to define an access level for packages and classes. Three different levels of access exist: non-accessible, discouraged, and accessible. When the compiler within Eclipse detects access to a type that should not be accessed, it creates a problem marker rather than a 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 compressed files in the `.classpath` files of the Eclipse projects. These access rules complement each compressed file's application programming interfaces (APIs) and through the `accessible rule` indicate access that is compliant according to the *Cúram Development Compliancy Guide*. Access Rules can be applied only to compressed files, so don't treat them as a complete solution to police compliancy. Classes that are defined as `non-accessible` or `discouraged` are not supported, are subject to change without notice, and might not respect their API. Hence, they can affect the ability to easily integrate IBM Cúram Social Program Management upgrades.

Note: Discouraged accesses in the unmodified IBM Cúram Social Program Management Platform might be copied into your codebase as part of subclassing or extension work. These accesses can be removed in future releases and appropriate alternative APIs provided if necessary. To reduce future impact to your codebase in regard to access to discouraged code, you must treat these accesses as non-accessible and work to seek a different API.

Working Sets

A common issue 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,* a method to specify, in a global location, which working set you are interacting with currently. 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.

For example, working sets can be useful especially on the Problems View, in terms of viewing which issues relate to your owned code. The following steps detail how to set a working set on the Problems View to display issues only 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 (for example, `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 box, select the **Selected Working Sets** radio button and click the **New...** button.

4. The **New Working Set** wizard then can 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 box, your Problems View will be updated to display only errors, warnings, or informationals relating to the newly created **Custom** filter.

Logging that uses Apache log4j 2 API

Use this information to understand logging in IBM Cúram Social Program Management. Logging in the application is provided by the `curam.util.resources.Trace` class that provides a convenient wrapper onto the Apache log4j 2 API.

log4j 2 is a logging framework that is provided by the Apache Jakarta project. For more information, see Apache Log4j 2.

Logging allows developers to log any information whether the program is being run in online or batch mode. The final destination of the trace information is configurable. It can be a log file that is associated with the application server, a stand-alone log file, a console, or even a database.

Logging usage

Use logging information in the tracing application API.

The interface into the tracing application programming interface (API) is through an instance of the `org.apache.logging.log4j.Logger` class. The infrastructure provides a number of named instances that match the categories that are described in "Logging hierarchy" on page 66. The top-level category is accessed through `curam.util.resources.Trace.kTopLevelLogger` as shown in the following example:

```
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);
```

Figure 40. Usage of the loggers

Note: The code example in figure 1 produces two trace records. These records are not visible easily if Apache Log4j 2 is configured to use a flat file or the console. However, if a Apache Log4j 2 viewer is used, then the two trace records result in a needless entry that complicates the view without any added benefit. As such, it is recommended that trace statements that contain logically dependent data be recorded in a single call.

A formatted textual representation of a IBM Cúram Social Program Management struct class object can be obtained through a call to the class `curam.util.resources.Trace.objectAsTraceString` call, for example see figure 2.

```
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));
```

Figure 41. Tracing a struct

Logging hierarchy

Understand the logging hierarchy for trace records and learn about the categories and levels of the records.

The IBM Cúram Social Program Management infrastructure produces trace records in specific categories with specific levels. This recording allows the records to be filtered easily in a Apache Log4j 2 viewer. The categories and levels that are supported are described in table 1 where *<BPO>*, *<Entity>*, and *<Facade>* are the names of the relevant Cúram classes. The *<CodePackage>* field is left empty if the class is not in a code package.

Table 15. Logging hierarchy		
Category	**Level**	**Meaning**
Trace	Error	Loggable exceptions that were detected 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 run 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
Trace.ServerCalls.*<CodePackage>*.*<Facade>*	Debug	Arguments and types of arguments for server method invocation.
Trace.Tools	Info	Progress of build time tools; for example, **configtest**

Table 15. Logging hierarchy (continued)		
Category	**Level**	**Meaning**
`Trace.Tools`	`Warning`	Warnings from build time tools.
`Trace.Tools`	`Error`	Errors from build time tools.

Logging trace levels

Learn about trace levels when you are logging information to the IBM Cúram Social Program Management server and learn how to use trace options.

When you are logging to the server, trace level needs to be considered. These settings can be used to guard the calls that are made into Apache Log4j 2 to improve the performance in environments where tracing is not required[4].

To check the current trace level setting, call the `curam.util.resources.Trace.atLeast(Trace t)` method, where the parameter to the method can be one of the following options:

- `curam.util.resources.Trace.kTraceOff`
- `curam.util.resources.Trace.kTraceOn`
- `curam.util.resources.Trace.kTraceVerbose`
- `curam.util.resources.Trace.kTraceUltraVerbose`

To specify the trace level for the application, set the `curam.trace` property, as defined in to one of the following values:

- `trace_on` (corresponds to O in)
- `trace_verbose` (corresponds to V in)
- `trace_ultra_verbose` (corresponds to U in)

The following code sample demonstrates how the amount of logging information that is output by your application code depends on the current trace level that is configured in the application:

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

Figure 42. Logging example in application code

The Cúram infrastructure supports a number of standard trace options that provide a convenient view in addition to the trace levels. All of the options write information to the log and therefore reduce the performance of the application. You can set the following properties as described in *Cúram configuration settings*, and the level at which they are set at default (O is On, V is Verbose, U is Ultra).

Table 16. Diagnostic tracing options		
Trace property name	**Description**	**Enabled**
`curam.trace.servercalls`	Trace server method invocations by remote clients. This information includes the name of the user who is requesting the invocation.	O

---

[4] While Apache Log4j 2 imposes a minimal memory allocation, it cannot avoid the cost of the parameter construction inside the method invocation. Application developers must take this operation into consideration.

Trace property name	Description	Enabled
Table 16. Diagnostic tracing options (continued)		
`curam.trace.methods`	Trace all business object method invocation.	V
`curam.trace.method_args`	Memory dump arguments, including their types, to business object method invocations.	U
`curam.trace.sql`	Trace SQL statements run by entity objects.	V
`curam.trace.sql_args`	Memory dump results of SQL select statements.	U
`curam.trace.rules`	For more information, see the Runtime Rules Logging in the *Cúram Rules Codification Guide*. Enables logging of rules	U
`curam.trace.smtp`	Trace the messages that are sent to the mail server.	

Configuring the Apache Log4j 2 logging utility

Understand how to use the Apache Log4j 2 logging utility.

The Apache Log4j 2 logging utility provides extensive support for configuring the destination of the trace information. The documentation that follows does not duplicate the Apache Log4j 2 documentation but places this information in the context of IBM Cúram Social Program Management. The configuration information needs to 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 Apache Log4j 2 setting is taken from `log4j2.xml` is in `log4j2-config.jar` in the CURAMSDEJ/lib directory. This default log4j setting is to use a Console Appender. The Console Appender outputs everything at the default (or higher) Apache Log4j 2 level to System Out. The default Apache Log4j 2 level for the top-level logger (and all inherited loggers) is set to DEBUG.[5]

Configuration results in trace information to be written to a rolling file appender. This operation means that the output is placed in a file until it reaches a specified size. After 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 first is renamed to `.2`.

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

---

[5] The set of possible levels (in order of priority) defined byApache Log4j 2 is ALL, DEBUG, INFO, WARN, ERROR, FATAL, and OFF. Only those items that are logged at the specified level or higher levels are included in the log.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
| For more configuration information and examples
| see the Apache Log4j website:
| https://logging. in the CURAMSDEJ/lib directory apache.org/log4j/2.x/manual/configuration.html
-->
<Configuration status="INFO">

<!-- =========================== -->
<!--  Append messages to a File  -->
<!-- =========================== -->
<Appenders>
    <RollingFile name="OutputToFile" fileName="CuramProps/CuramAppLog.log"
    filePattern="CuramProps/CuramAppLog.log.gz">
        <PatternLayout
pattern="[%-5p] [%d{dd MMM yyyy HH:mm:ss}] [%c] - %m%n"/>
        <Policies>
            <SizeBasedTriggeringPolicy size="500" />
        </Policies>
        <DefaultRolloverStrategy max="3" />
    </RollingFile>
</Appenders>

<!-- ====================== -->
<!--  Setup the Root Logger  -->
<!-- ====================== -->
<Loggers>
    <Root level="INFO">
        <AppenderRef ref="OutputToFile"/>
    </Root>
</Loggers>
</Configuration>
```

Figure 43. Configuring Apache Log4j 2

A number of customizable values exist 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 that are preserved is set to 3.
- The maximum file size is not explicitly set so the default of 500 Kb is used.
- The conversion pattern has several parameters that results in the following output:
  - **%-5p** - The level of the trace message after it is left padded to be a five character string.
  - **%c** - The category of the trace message.
  - **%m** - The trace message itself.
  - **%n**- A platform-specific line separator.
- The Apache Log4j 2 level is set to INFO, which means that all items that are logged at the DEBUG level are ignored. This operation overwrites the default level of DEBUG set by the infrastructure.

Two application properties examined when the process populates the **{user}** and **{alternateuserid}** parameters:

Table 17. Application properties examined when the *{user}* and *{alternateuserid}* parameters are populated	
Property Name	**Explanation**
`curam.security.altlogin.enabled`	A Boolean flag to indicate that users can log in to the application by using their alternative login ID.
`curam.trace.deferred.user.name`	A Boolean flag for deferred processing transactions that, when set to true, indicates that the name of the user who initiates the deferred process transaction is made available for logging purposes.

The tables that follow represent use cases that can be achieved with the new feature. The feature contains two aspects - online transactions and deferred process transactions.

Depending on the values that are specified for the two properties that were described, the following data is made available for logging in the **{user}** and **{alternateuserid}** parameters:

Table 18. Use case scenarios for online transactions

Property name: `curam.security.altlogin.enabled`	Username	Alternate Login ID	Login Used	Available Values
TRUE	caseworker	caseworkeralt	caseworker	user=caseworker alternateuserid=caseworkeralt
TRUE	caseworker	caseworkeralt	caseworkeralt	user=caseworker alternateuserid=caseworkeralt
TRUE	caseworker	-	caseworker	user=caseworker alternateuserid=caseworker
TRUE	caseworker	-	caseworkeralt	ERROR
FALSE	caseworker	caseworkeralt	caseworker	user=caseworker alternateuserid=""
FALSE	caseworker	caseworkeralt	caseworkeralt	ERROR
FALSE	caseworker	-	caseworker	user=caseworker alternateuserid=""

Table 19. Use case scenarios for deferred process transactions

Property name: `curam.trace.deferred.user.name`	`curam.security.altlogin.enabled`	Username	Alternate Login ID	Deferred username	Deferred username Alternate Login ID	Login Used	Available Values
TRUE	TRUE	SYSTEM	-	beantester	beantesteralt	beantester	user=beantester alternateuserid=beantesteralt
TRUE	FALSE	SYSTEM	-	beantester	beantesteralt	beantester	user=beantester alternateuserid=""

Property name: curam.trace.deferred.user.name	curam.security.altlogin.enabled	Username	Alternate Login ID	Deferred username	Deferred username Alternate Login ID	Login Used	Available Values
FALSE	TRUE	SYSTEM	-	beantester	beantester alt	beantester	user=SYSTEM alternateuserid=""
FALSE	FALSE	SYSTEM	-	beantester	beantesteralt	beantester	user=SYSTEM; alternateuserid=""

Table 19. Use case scenarios for deferred process transactions (continued)

However, direct access to a file might not be an ideal mechanism if the trace output needs to be monitored. Configuration results in trace information to be written to a socket. A listener (such as Apache Chainsaw that is delivered with Apache Log4j 2) can then be used to display the resultant information.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--
| For more configuration information and examples
| see the Apache Log4j website:
| https://logging.apache.org/log4j/2.x/manual/configuration.html
-->
<Configuration status="INFO">

<!-- ========================== -->
<!-- Append messages to a Socket -->
<!-- ========================== -->
    <Appenders>
        <Socket
name="OutputToSocket"
host="localhost"
port="4445">

            <PatternLayout
pattern="[%-5p] [%d{dd MMM yyyy HH:mm:ss}] [%c] - %m%n" />
        </Socket>
    </Appenders>

<!-- ====================== -->
<!--  Setup the Root Logger  -->
<!-- ====================== -->
    <Loggers>
        <Root level="INFO">
            <AppenderRef ref="OutputToSocket"/>
        </Root>
    </Loggers>

</Configuration>
```

Figure 44. Configuring Apache Log4j 2 to log to a socket

The conversion pattern that is 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` for the hostname and 4445 for the port of the remote server.

Numerous other possibilities exist for this configuration and the explanation that is presented here does not attempt to duplicate the existing Apache Log4j 2 documentation. However, Nested Diagnostic Contexts are not supported.

Related concepts

Alternate Login IDs

Logging statistics

Collect and use performance information about client visible IBM Cúram Social Program Management server functions.

Tracing facilities 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; that is, any operations started by the Cúram web client. However, writing trace informational impacts performance because the Apache Log4j 2 appender always needs to maintain the contents after a server crash (for example, do not use buffered file access). For performance benchmarking, the benchmarking process must not cause a performance processor issue on the application that is being measured. For this reason, a way to collect server function performance statistics is provided that imposes less processor usage than server tracing. The process also produces output in a format that is suitable for automated processing as part of benchmark analysis.

To avoid performance processor usage issues 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 4 Kb memory buffer, so a memory usage limit imposed by the collection of server benchmarks. It is assumed that a realistic benchmark configuration involves application servers with a significant amount of physical memory.

The statistics files are created in the directory that is 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 format in table 1:

Table 20. Statistics file elements	
Summary	**Meaning**
Timestamp	This timestamp is in a sortable format (ISO 8601 complete) and indicates the time at which the method was started. 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:** 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 on which this function ran.
Session bean name	The name of the statistics class, Statistics, is always printed.
Process ID	Currently hardcoded to zero.
Server function signature	The function signature that includes class and method name, and method argument types.
Success indicator	A flag that indicates whether the server function succeeded with no errors that are returned to the client. A value of 1 indicates success. A value of 0 indicates0 failure. The specific error message is not recorded.

Table 20. Statistics file elements (continued)	
Summary	**Meaning**
Elapsed time in milliseconds	This number is the time (in milliseconds) that is spent running this function that excludes time that is spent by the middleware software in dispatching the function call and marshalling arguments.

Localization of log messages

Use this information to understand how to create and when to use localized log messages.

In cases where log messages need to be localizable, class `LocalisableString` can be used. For more information, see "Localized output" on page 80. However, it is important to note that logged messages typically are targeted at a system administrator who might 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 is French and the log message is written in French. In the following example, the default server locale explicitly is passed into `getMessage`, otherwise `getMessage` returns 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()));
```

Figure 45. Localizable logging example in application code

Note: To display the localized content (in languages other than English) correctly on a command line, you need to change the system locale. (Change the language setting in **Control Panel** > **Region and Language Administrative** > **Formats** > **Format** and **Control Panel** > **Region and Language Administrative** > **Language for non-Unicode programs** > **Change system locale**)

How to enable dynamic UIM tracing

Use this information to understand how to enable dynamic UIM tracing, which is unavailable by default, and how to set properties to enable logging.

Logging of a missing Dynamic User Interface Metadata (UIM) resource is unavailable by default. To enable this logging, the `Tracing Level` property must be set to `trace_on` or higher and the `Enable tracing of Dynamic UI` property must be set to `true`.

Both of these properties can be set by using the **System Administration** application. From the **Shortcuts Panel** of the **System Configuration** section, select **Application Data** > **Property Administration**.

How to use exceptions

Exceptions are the recommended mechanism for handling errors in a Cúram application. Exceptions save the developer from having to check the status of each attempted operation. A single `try..catch` construct can enclose many statements, each of which can 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 when a database error occurs, application developers can throw pre-defined exceptions or customized exceptions. Two basic forms of exceptions are 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. Typically, database problems (such as a `RecordLockedException`) are thrown as unchecked exceptions. This action means that no need exists 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. After exceptions reach the RIL, they are converted to a different form of exception that is thrown to the client, and might write information from the exception to the log file. To avoid this problem, a developer can write code to catch exceptions and handle them, rethrow them before the exception reaches the RIL, or both.

The following actions occur 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 is 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 information is written to the log file.
- An exception is created and thrown to the client. This exception 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 run 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 that has the details of the original exception. Specifically, the exception includes 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 exceptions 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.

Constructing an exception

Use this information to understand how to construct an exception.

Exceptions[6]are created typically with a catalog name and message identifier. If these are not specified default values are used. The server infrastructure will take care of delivering the message text to the client or log file or both. 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);
}
```

Figure 46. Constructing an AppException

The purpose of exceptions is to communicate that an error has occurred and to communicate information about that error. Often it is necessary to include additional information in addition to the error code. This can be done using arguments.

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

To attach an argument to an exception, the arg method (.arg()) is used. "Constructing an exception" on page 75 shows a code example 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;
```

Figure 47. 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; for instance, Date, DateTime, Money, and CodeTableItemIdentifier objects can also be added. For more information, see the JavaDoc for curam.util.exception.AppException.

---

[6] 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.

```
// 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;
```

Figure 48. Using the arg method with a complex type

Creating messages with argument placeholders

Argument place holders are tokens that are included in error message source text and replaced by an argument at run time.

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

- s - string
- n - numeric
- d - date
- t - time
- z - date/time
- c - code table item

This source message is displayed with the actual values. For example:

```
The first name is %1s and the surname is %2s.
```
```
The first name is John and the surname is Smith.
```

Place holders are numbered and they can appear in the message in any order. For example:

```
The second name is %2s and the first name is %1s.
```
```
The second name is Smith and the first name is John.
```

The exception is constructed and thrown as shown in this example of an 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;
```

Handling exceptions

Use this information to understand how exceptions are handled, particularly in `try..catch` constructs.

When an exception is thrown in an application, it might be caught within a `try..catch` construct or it can be allowed to filter up to the Radio Interface Layer (RIL).

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

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

  An example of this operation is where the program must check for the existence of a record on the database. If the Data Access Layer (DAL) throws a `RecordNotFoundException`, then this action indicates that the record does not exist. This exception is not 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 rethrowing the actual exception.

  An example of this action is a `try..catch` construct that 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 an exception and throw the new exception.

  An example of this situation is where the handler would replace a generated DAL exception with an application exception that contains an 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 an exception, attach the original exception to this new exception, 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 action has the effect of creating a linked list of exceptions with the most recent exception at the head of the list and allows a detailed history of an exception to be built up for auditing or debugging purposes.

Logging exceptions

Use this information to understand how to use loggable exceptions that use the `setLoggable` method.

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

Loggable exceptions are written to the application log file by the Radio Interface Layer (RIL). The exception message is read from the error message catalog file. If any exception arguments exist, they 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 that is being logged includes any other attached exceptions, then these exceptions also are logged.

General exception guidelines

Use this information to understand 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**, **nsmodify**, **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 non-keyed 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**, and **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**, and **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**, and **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 previously outlined exceptions.

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 "Coding Conventions for Exceptions" on page 78.

```
//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;
}
```

Figure 49. Incorrect usage of hard-coded literals

How to use the Record Not Found indicator

Use this information to understand how to use the Record Not Found indicator variable.

Each of the singleton reads of the database (**entity read**, **nsread**, and **nkread** operations) that potentially can throw a `RecordNotFoundException` has overloads added to take a Record Not Found Indicator variable.

The reasons for providing a Record Not Found Indicator are as follows:

- To save the processor usage of creating and throwing an exception whenever a record cannot be found, as this process is expensive in some Java virtual machines (JVMs).
- To make it easier to write code that checks for the existence of a record.

This indicator (`curam.util.type.NotFoundIndicator`) wraps a Boolean value that indicates whether the required record might not be found. When this indicator is passed into one of the previously outlined read operations, the operation never throws a `RecordNotFoundException` if the record does not exist but instead sets 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 wants to pass a `NotFoundIndicator` into a singleton read operation, it is always passed in as the first argument. This operation is shown in the following examples:

```
try {
  bankAccountDtls = bankAccount.read(bankAccountKey);
} catch (RecordNotFoundException rnf) {
  // record was not found...
}
```

Figure 50. A typical read operation that might 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...
}
```

Figure 51. The overloaded version of the one previous, using the `NotFoundIndicator`

```
try {
  bankAccountDtls = bankAccount.read(bankAccountKey, true);
} catch (RecordNotFoundException rnf) {
  // record was not found...
}
```

Figure 52. A typical read operation for update that might throw a `RecordNotFoundException`

```
bankAccountDtls =
  bankAccount.read(notFoundInd, bankAccountKey, true);
if (notFoundInd.isNotFound()) {
  // record was not found...
} else {
  // record was found...
}
```

Figure 53. The overloaded version of the one previous, using the `NotFoundIndicator`

Localized output

Use this information to understand when it is necessary to use the `curam.util.exception.LocalisableString` class to present data to the client for localization.

In IBM Cúram Social Program Management, the client is responsible for converting the text of an exception into the language that a user chooses. However, certain situations exist where the server must present data to the client for localization. To facilitate these situations, the `curam.util.exception.LocalisableString` class was created. This class is used in a similar manner to AppException, as shown in the following example:

```
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();
```

Figure 54. Use of `LocalisableString`

This string can be passed back to the client as an output parameter to be localized by the client.

Use of the Informational Manager

Use this information to understand how and when to use the Informational Manager function.

The standard exception handling and string presentation features described in the "Using exceptions" 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 validation a number of warnings, or errors, can occur independently as they are based on different elements of the user input. These errors need to be reported together to simplify the corrective actions that a user must take. The `InformationalManager` class allows for exceptions and informationals to be grouped in this manner. 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;
  }

}
```

Figure 55. Use of the Informational Manager

A number of points are worth emphasizing in this code fragment:

• This sample is based around the presentation of informationals to the client. It does not throw an exception, it is a successful invocation of the method. This action means that the transaction is be

committed and any database updates is 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 that depends on whether the informational manager contains any warnings or errors. If the informational manager contains an error or warning, then this method throws an exception that means the transaction is 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) included with IBM Cúram Social Program Management.

- The second parameter to `InformationalManager.addInformationalMsg` currently populated with `GeneralConstants.kEmpty` (as in "Use of the Informational Manager" on page 80) is intended to name a field. However, this feature is not supported in the current release

The *Cúram Web Client Reference Manual* needs to be consulted to determine the client-side configuration that is necessary to use the `InformationalManager`. At its simplest, the field in the struct that contains the informationals must be named in the UIM.

The `InformationalManager` logs informationals to the Curam log. Please see "Logging" on page 126 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.

Message files

Message catalogs allow an application to be globalized without manipulating hand-crafted code. Review the message fundamentals and understand how you can augment them to produce customized messages in a Cúram application.

Traditionally message files or catalogs are binary files that hold text messages that are 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 globalized without needing to recompile.

In keeping with this approach, Cúram message catalogs are generated from message `.xml` files by 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 that contains constants for message identifiers that are enumerated in the message `.xml` file, and the name of the message file itself. In Java terms, a constant is a static final. This file should be imported into any Java source files that use that catalog. The message catalog is a properties file that is opened by the Cúram application at run time.

The **msggen** build target merges message files and then converts the resultant message file (which are stored in `/build/svr/message/scp`) into symbol definition (Java code) and message catalog (property) files.

The **msggen** build target is automatically started by the provided build scripts, and runs against any message files that are placed in the suggested source locations, that is, the `/message` directory of a component.

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 "The Format of Message Files" on page 82.

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>
```

Figure 56. 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
Table 21. Attributes of the messages Element			
package	Yes	None	The Java package name to use for the generated Java file.

The `<message>` Element

The <message> element groups a number of <locale> elements together. The message element has the following attributes:

Attribute Name	Required	Default	Description
Table 22. Attributes of the message Element			
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.

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:

Table 23. Attributes of the locale Element

Attribute Name	Required	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 SERVER_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 language_country. For example: SERVER_LOCALE_LIST=en, en_US, en_GB.
country	No	None	Set to the country relevant to the locale language attribute.

Customizing a Message File

Message text files are located in the /message directory of a component. IBM Cúram Social Program Management Platform includes a set of message files. You can override these messages by placing new message files in the SERVER_DIR/components/<custom>/ message directory, where *<custom>* is any new directory that is 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.

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

Figure 57. SERVER COMPONENT ORDER example

The order in SERVER_COMPONENT_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 that are not specified are placed alphabetically above core and below those that are specified.

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 the example SERVER_COMPONENT_ORDER, 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 "Rules of Message Merges" on page 84, and the merge file of "Rules of Message Merges" on page 84, 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>
```

Figure 58. 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>
```

Figure 59. 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>
</messages>
```

Figure 60. Resulting Message File

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`.

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");
}
```

Figure 61. 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.
```

Figure 62. 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.

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 with the example below 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>
  <locale language="en">The en message</locale>
</message></messages>
```
Figure 63. 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 performed. 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 the example above with runtime locale en_US will return the message text "The message".

- If nothing is found for the runtime locale, then a lookup will be performed using the fall back locale of en. Looking up the message named A_MESSAGE in the example above with runtime locale es will return the message text "The en message", i.e. the lookup will revert to the fall back locale of en as nothing can be found for es.

If nothing can be found for either the runtime locale or the fall back locale, then the search will be determined based on the underlying message lookup mechanism 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 fall back mechanism provided.

If the runtime locale does not find a match in the message file and no match can be found using the fall back locale of en, and no match can be found after applying the fall back rules described by `java.util.ResourceBundle`, a `MissingResourceException` is returned and server logs are updated if appropriate.

Writing Messages To Server Logs

Messages from message catalogs are used in many instances in Cúram and localized at runtime as described in "Retrieving Messages from Message Files" on page 86. 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`.

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.

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.

Code table files

Code table files allow a Cúram application to use a level of indirection when it stores commonly used constants on the database. Review the code table fundamentals and understand how you can augment them to produce customized code tables in a Cúram application.

Code table files are included 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 that is 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 that contains constants for code table identifiers that are used in the code table XML file. The generation of code table hierarchies also produces `.properties` files as described in "Artifacts produced by the ctgen build target" on page 100. Generating code tables is supported by the **ctgen** build target.

For more information about code tables, see the Domain Definitions chapter in the *Cúram Modeling Reference Guide* and the *Cúram Web Client Reference Manual*.

The code table file format

The code table file is an XML document that consists of a number of distinct elements and attributes. For a sample code table, see "Rules of code table merges" on page 94.

As a standard XML document, the `encoding` attribute indicates that the file is encoded in UTF-8. While this encoding is respected and maintained by an XML aware editor, some other editors do not maintain this encoding by default. You might have to specifically save a file that contains UTF-8 characters as UTF-8 with these editors.

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 is part of.
hierarchy_name	No	None	Identifies the code table file as containing a hierarchy of code tables.

Table 24. Attributes of the `codetables` element

The <description> element for the <codetables> element

The <description> element is an optional subelement in the <codetables> root element. It is used to define a description for the code tables. It must be listed first, before the other subelement, <codetable> and must be listed only once. The <description> element has no attributes.

The <codetable> element for the <codetables> element

The <codetable> element is a subelement in the <codetables> root element. The <codetable> element must follow the <description> element where specified. For an ordinary code table file definition, only a single <codetable> element can be defined. If a `hierarchy_name` attribute is specified in the <codetables> element, multiple <codetable> elements are allowed when 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:

Table 25. Attributes of the `codetable` element

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. For more information, see "Artifacts produced by the ctgen build target" on page 100.
java_identifier	Yes	None	The name of the generated symbol definition Java file. This identifier cannot be duplicated for code tables with different names.
parent_codetable	No	None	Used to define the name of the parent code table in the hierarchy, where the code table file was defined as a hierarchy of code tables.

The <codetabledata> element

The <codetabledata> element is an optional subelement of <codetable> that groups the
locale-specific comments for a code table. Each <codetable> element can have one optional
<codetabledata> element. The <codetabledata> element can contain multiple optional <locale>
elements.

Note: The <codetabledata> element and its child elements are optional elements.

The <codetabledata> element has the following attributes:

Table 26. Attributes of the `codetabledata` element

Attribute Name	Required	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.

The <locale> element for the <codetabledata> 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:

Table 27. Attributes of the locale element			
Attribute Name	**Required**	**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.

The `<comments>` element for the `<codetabledata>` 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, and it also groups a number of code table name `<locale>` elements together. It is an optional element. However, if present it can contain any one `<name>` element or `<locale>`, having a `<locale>` element helps the client to display the code table name in the locale set for the current user. The `displaynames` element has no attributes.

The `<name>` element

The `<name>` element is optional when the `<displaynames>` element is present. The name that contains the locale for the current user is displayed when displaying the `<name>` values on the client. However, if the current user's locale does not match any of the locales that are specified within the `<name>` element, then the `<codetable>` name attribute is displayed.

The name element has the following attributes:

Table 28. Attributes of the name element			
Attribute Name	**Required**	**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.

The `<locale>` element

The `<locale>` element is optional and is used to add localizable display names to represent the code table name when the `<displaynames>` element is present. The name that contains the locale for the current user is displayed when displaying the `<codetable>` name attribute on the client. However, if the current user's locale does not match any of the locales that are specified within the `<locale>` element, then the `<codetable>` name attribute is displayed.

The `locale` element has the following attributes:

Table 29. Attributes of the locale element			
Attribute Name	**Required**	**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.

The <code> element

The `<code>` element is a subelement of `<codetable>` and groups a number of `<locale>` elements together. The code element has the following attributes:

Table 30. Attributes of the code element			
Attribute Name	**Required**	**Default**	**Description**
`value`	Yes	None	A unique identifier for the code in the code table.
`status`	Yes	None	Indicates whether 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 whether this code is the default code for the code table. You must specify only one default code. The default code is used to define the initially selected value in an editable code table list in the client. For more information, see Cúram web client reference.
`java_identifier`	No	None	Used as part of the generated symbol definition Java file
`removed`	No	`false`	Set to `true` to indicate whether the code is to be removed and hence not included in the generated artifacts.
`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. For more information about defining a code table hierarchy, see "Code table hierarchy" on page 103.

The <locale> element for the <code> element

The `<locale>` element contains two mandatory subelements (`<description>` and `<annotation>`) and one optional subelement `<comments>`, which are used to describe the code.

To be included during generation of the code table artifacts, 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 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:

Table 31. Attributes of the locale element

Attribute Name	Required	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 are displayed in the drop-down list on an edit page in the client.

The `<description>` element for the `<locale>` element

The `<description>` element is used to provide a description for the `<code>` element. The `description` element has no attributes.

The `<annotation>` element for the `<locale>` element

The `<annotation>` element is used to provide an annotation to the `<code>` element. The `annotation` element has no attributes.

The `<comments>` element for the `<locale>` 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.

The `<views>` element

The `<views>` element is an optional subelement of the `<codetable>` element that groups one or more views of the code table. Each child view corresponds to a specific application context. For more information, see Application definition.

The `<views>` element has no attributes.

The `<view>` element

The `<view>` element is a subelement of the `<views>` element. Each view corresponds to a specific application context. The `<view>` element groups a number of view `<code>` elements together. The `<code>` elements in the `<view>` element are structurally different from the elements that are defined in the `<code table>` element. The view `<code>` element contains the code of a code table item that is displayed when the application is running in a context that is defined by the parent `<view>` element.

The `<view>` element has the following attributes.

Table 32. Attributes of the views element

Attribute Name	Required	Default	Description
`context`	Yes	None	The application context code table code value that is taken from the `ApplicationContext` code table. For more information, see Application definition.
`default_code`	No	None	A code table code value that is defined in the `<code table>` element and must be one of the codes that are defined by the child code elements of the current view element. This code table item of this code is displayed as the default code table in the user interface when this code table view is accessed.
`overwrite`	No	False	Boolean value that indicates whether this view overwrites a view that is located in another code table in a different component when the code tables are merged. When two code tables are merged, the views that have the same application context are merged. For more information about merging code table views, see "Rules of code table merges" on page 94.

The `<code>` element for the `<view>` element

The `<code>` element is a subelement of `<view>` element. The `<view>` element groups one or more `<code>` elements together. Each `<code>` element holds the code of a code table item that is displayed when the application is running in a context that is defined by the parent `<view>` element. This `<code>` element in the `<view>` element is structurally different from the one that is defined for the `<codetable>` element.

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

Table 33. Attributes of the `<code>` element

Attribute Name	Required	Default	Description
`value`	Yes	None	A code table code value that is defined in the `<codetable>` element.

Customizing a code table file

You can customize code table files without making changes directly to the included code table files, which would complicate later upgrades. Typically code table files are customized to add new entries, localize descriptions or to add new locales.

Code table files are located in the `/codetable` directory of a component. IBM Cúram Social Program Management Platform includes a set of code table files. You can override these code tables 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 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 first component has the highest priority, and the last

component the lowest. For more information about SERVER_COMPONENT_ORDER, see "Customizing a Message File" on page 84.

Rules of code table merges

Code table files are merged based on the order of precedence, there is always a more important main or source code table file, and a file that is being merged into it. The second file is called the merge file.

The merging rules are applied to decide whether the `<code>`, `<locale>`, `<displaynames>`, `<name>`, `<views>`, and `<view>` elements are merged into the new code table file.

- A `<code>` is 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>` is 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 is 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 need to be merged. If the `<name>` element with its `language` and `country` attributes is not already present in the new file, then it is merged into the new file.
- A `<views>` element is merged into a new code table file if its associated `<views>` element is present in the new file and the child `<view>` element is not already present in the new file.
- A `<view>` element is merged into a new code table file if its associated `<view>` with the same application context is present in the new file with a set of different child `<code>` elements.

  If the overwrite attribute of the `<view>` is set to true, the view overwrites the contents of the associated view in the new code table file.

  If a `<view>` element has the *default_code* attribute set, the *deafult_code* of the associated `<view>` in the new file is overwritten if it exists.

  A `<code>` element is merged into a `<view>` element that has the same application context in the new code table file if the `<code>` is not already present in that `<view>` of the new code table.

Merging sample main code table file 1 and sample merge code table file 1 illustrates the rules of merging `<code>`, `<codetabledata>` and `<locale>` elements, as shown in the resulting code table file 1.

```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>

    <views>
      <view context="CTX1" default_code="ACS1">
        <code value="ACS1"/>
        <code   value="ACS2"/>
        <code value="ACS3"/>
      </view>
      <view context="CTX2">
        <code value="ACS1"/>
        <code value="ACS3"/>
      </view>

      <view context="CTX4" overwrite="true">
        <code value="ACS2"/>
        <code value="ACS3"/>
      </view>
    </views>

  </codetable>
</codetables>
```

Figure 64. Sample Main Code Table File 1

Figure 65. Sample Merge Code Table File 1

As a result of the merge process the resulting code table file would be:

```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="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>
      <views>
        <view context="CTX1" default_code="ACS4">
          <code value="ACS1"/>
          <code  value="ACS2"/>
          <code value="ACS4"/>
        </view>
        <view context="CTX2">
          <code value="ACS3"/>
        </view>
        <view context="CTX3">
          <code value="ACS3"/>
          <code value="ACS4"/>
        </view>
        <view context="CTX4">
          <code value="ACS4"/>
          <code value="ACS3"/>
        </view>
    </views>

  </codetable>
</codetables>
```

```xml
<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>
    <views>
      <view context="CTX1" default_code="ACS1">
        <code value="ACS1"/>
        <code  value="ACS2"/>
        <code value="ACS3"/>
        <code value="ACS4"/>
      </view>
      <view context="CTX2">
        <code value="ACS1"/>
        <code value="ACS3"/>
      </view>
      <view context="CTX3">
        <code value="ACS3"/>
        <code value="ACS4"/>
      </view>
      <view context="CTX4" overwrite="true">
        <code value="ACS2"/>
        <code value="ACS3"/>
      </view>
```

Merging sample main code table file 2 and sample merge code table file 2 illustrates the rules of merging `<displaynames>` and `<name>` elements, as shown in the resulting code table file 2.

```
<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>
```

Figure 67. 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>
```

Figure 68. 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>
```

Figure 69. Resulting Code Table File 2

Artifacts produced by the `ctgen` build target

The artifacts that are produced from code table files are a symbol definition file (Java class) and an SQL file.

The symbol definition file is a Java file that contains constants for code table identifiers that are used in the code table XML file. This file can be used 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 that is specified in the code table file. For example, *package="curam.codetable"* places Java artifacts in the /build/svr/codetable/gen/curam/codetable directory.

The code table SQL file contains inserts for the `CodeTableHeader`, `CodeTableItem`, `CodeTableView` and `CodeTableViewCode` database tables. All SQL file artifacts 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) artifacts are placed in the package that is specified by the code table file of the component with the highest precedence order. The precedence is listed in the SERVER_COMPONENT_ORDER environment variable.

A sample Java file from a code table file.

```java
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");
}
```

This pattern of generation means that the Strings are not 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 artifacts.

The persistent data that is associated with code tables is generated into the common /build/svr/ codetable/sql/ directory.

A sample SQL file from a code table file.

```sql
--
-- 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(''));
INSERT INTO CodeTableView (TABLENAME, CONTEXT, DEFAULTCODE, LASTWRITTEN)
    VALUES ('AcceptanceStatus', 'CTX1', 'ACS1', CURRENT TIMESTAMP);
INSERT INTO CodeTableViewCode (TABLENAME, CONTEXT, CODE, LASTWRITTEN)
    VALUES ('AcceptanceStatus', 'CTX1', 'ACS1', CURRENT TIMESTAMP);
INSERT INTO CodeTableViewCode (TABLENAME, CONTEXT, CODE, LASTWRITTEN)
    VALUES ('AcceptanceStatus', 'CTX1', 'ACS2', CURRENT TIMESTAMP);
INSERT INTO CodeTableView (TABLENAME, CONTEXT, DEFAULTCODE, LASTWRITTEN)
    VALUES ('AcceptanceStatus', 'CTX2, 'ACS3', CURRENT TIMESTAMP);
INSERT INTO CodeTableViewCode (TABLENAME, CONTEXT, CODE, LASTWRITTEN)
    VALUES ('AcceptanceStatus', 'CTX2, 'ACS3, CURRENT TIMESTAMP);
INSERT INTO CodeTableViewCode (TABLENAME, CONTEXT, CODE, LASTWRITTEN)
    VALUES ('AcceptanceStatus', 'CTX2, 'ACS4', CURRENT TIMESTAMP);
```

Note: If any `<locale>` entries specify a language, and optional country, that are not in the SERVER_LOCALE_LIST environment variable, they are ignored during generation and a warning is produced.

Also, while generating the code table SQL artifacts containing the contents for the CodeTableItem and CodeTableHeader database tables, the **LASTWRITTEN** field with an initial value is populated. The initial value is a time stamp that is set to the time when the data is inserted into the database.

The same artifacts are produced for the code table file of , also, because the file contains a `<displaynames>` element, additional artifacts are created. That is, a properties file is generated for each `<name>` element it contains.

The **ctgen** build 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 that are 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 is 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 that are 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 `prp.warningstoerrors` property.

If the locale specified for the `<name>` element is not supported, then the **ctgen** build target displays a warning and no properties file for that locale is generated.

The following is an example of properties files that are produced by the **ctgen** build target on the "Rules of code table merges" on page 94. Each properties file is generated to `/build/svr/codetable/gen/`

CarMake_en_US.properties

```
CarMake=Car Make US
```

CarMake_fr.properties

```
CarMake=Marque
```

CarMake_en_GB.properties

```
CarMake=Car Make CustomGB
```

CarMake_lt.properties

```
CarMake=Masinos Gamintojas
```

CarMake_en.properties

```
CarMake=Car Make Custom
```

CarModel_en.properties

```
CarModel=Car Model
```

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, selecting Ireland as the country in the sample address hierarchy returns a sub-list of Meath and Wexford and selecting Meath as the county returns a sub-list of Trim and Navan. Alternatively, selecting England returns a sub-list of Stafford and London.

Table 34. Sample Address Hierarchy		
Level 1	**Level 2**	**Level 3**
Country	County	Town
Ireland	Meath	Navan
		Trim
	Wexford	Gorey
		Enniscorthy
England	Stafford	Bednall
		Stone
	London	Earlsfield
		Eltham

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>
```
Figure 70. Usage of hierarchy_name attribute

Each <codetable> defined must then be linked using the `parent_codetable` attribute of the <codetable> element. The `parent_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">
```
Figure 71. 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 `parent_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">
```
Figure 72. Usage of parent_code attribute

The hierarchy defined in <u>"Code table hierarchy" on page 103</u> can be represented as follows in a code table file.

```xml
<?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">
```

The artefacts listed in "Artifacts produced by the ctgen build target" on page 100 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.

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.

Note: Code table views are not supported by code table hierarchies. Code tables that are used in code table hierarchies must not contain views.

Retrieving locale 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, optionally, 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 the example below, 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="en">
        <description>The en code</description>
        <annotation></annotation>
      </locale>
      <locale language="fr">
        <description>Le code</description>
        <annotation></annotation>
      </locale>
    </code>
  </codetable
</codetables>
```
Figure 74. Code File Search

Since code table files are not guaranteed to contain an entry for every country, 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 the example above, with runtime locale `fr_CN` would return the text "Le code".
- If nothing is found for the runtime locale for either language and country or language only, then a search using the fall back locale of en will be used. Looking up the code named ACODE in the example above, with runtime locale `es` would return the text "The en code".

For example, if the runtime locale is set to `fr_CA`, then the following will be the search path:

- Search on fr_CA,
- Search on fr
- Search on en

If nothing is found for either the runtime locale or the fallback locale of en, then an empty string is returned.

Localizing SDEJ code table files

You can localize or modify the code table files shipped with the SDEJ. These code table 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` code table file to be modified from the SDEJ to the code table directory of a component in your Cúram application, for example, `SERVER_DIR/components/custom/codetable`. The `.itx` code table file can then be modified in the same way as any code table file; overriding a code or adding a new locale for all the codes.

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` code table file with the original one located in the SDEJ. The localized code table file will have the higher precedence order. It will then generate the sql files only. No Java artefacts will be generated for code table 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"/>
```
Figure 75. Datamanager entry for the code table SQL artefacts location

Note: The <description> sub-element is an optional element for the <codetables> element in the code table (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.

Specialized readmulti operations

Use generated readmulti operations to run SQL **SELECT** statements and return the resulting record sets as an `ArrayList`. Readmulti operations consist of a Data Access Layer function and a Business Object Layer function.

Readmulti operations are implemented as two distinct pieces:

- A Data Access Layer function that establishes the result set, through building up the statement, running an `executeQuery` on it, then a series of `getResultObject` statements, and
- A Business Object Layer function that 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 that provides an implementation of "normal" readmulti functions.

"Specialized readmulti operations" are handcrafted functions "plugged into" the Data Access Layer that uses generated helper classes. The pattern in use here is similar to the "Visitor" design pattern described in *Design Patterns*. 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.

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`.

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 Unified Modeling Language (UML) application model. For this example, assume that you add a standard readmulti operation that is called `readmulti` to an entity called E. The standard readmulti operation whose "details" structure is called `EDtls`. However, this example applies equally to readmulti, nsreadmulti, multithread, 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.StandardReadMultiOperation`. This specification is because this class builds up an in-memory list of the that is`structs` read by the readmulti operation to make it available to the`Post Data Access` and`On-fail` operations.

If your readmulti operation processes large numbers of records, then this operation might cause an excessive memory usage. Cution is advised if you are using specialized readmulti operations with`Post Data Access` or`On-fail` operations.

3. Implement `MyReadMultiOperation. operation` to initiate your specific processing. This method is 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 reach the end of the record set, return `false`. This instruction means that the operation method is not be called again.

4. Write the code that starts the readmulti operation. This code appears in a Business Process Object (BPO) implementation and looks like the following:

```
// 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 that exists solely for use in code such as displayed previously. The helper class is scoped inside the entity class and has a run method that begins a readmulti.

Extra features of readmulti operations

- The READMULTI_MAX option in the model limits the number of records processed by a standard "normal" readmulti operation. However, it has no effect when you hand-craft your own operations. As a result none of the overrides for this option (defined in "Cúram Configuration Settings" on page 31) has any effect. To limit the number of records that are 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 AppException,`
  `InformationalException;`

  Each record is passed to `filter` before it is passed to your `operation` method. Any record that 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, InformationalException;`

  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 AppException, InformationalException;`

  The same record is also passed to the `operation` method.

  Note: The `first` is called if there is at least one row in the result set, regardless of whether `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, InformationalException;`

  Be aware. This function is called always 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` that specifies whether records that are read from the database are updated later in the transaction. This parameter can be used as in:

```
E.newInstance().readmultiHelper(key, op, true);
```

  This process means that each record that is read from the database is locked for write access as it is read.

You can use a combination of the previously shown methods, with your own data members, to achieve many common styles of readmulti operation. For instance, "Extra features of readmulti operations" on page 109 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(
```

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*.

Summary

The order your readmulti operation methods are called is as follows:

1. `pre` - always called once before anything else.
2. `first` - called once with the first record, provided at least one record exists.
3. `filter` - called for each record (including the first).
4. `operation` - called for each record for which filter returns `true`;
5. `post` - always called once after everything else.
6. `getMaximum` - specifies the maximum number of records that are matched.

If you are designing processing that maintains locks, remember performance implications when you do so.

Deprecation

Use this information to understand how deprecation is used to reduce the impact of change on custom applications.

This chapter describes deprecation in Cúram: what it is, how it can affect custom code, and what it means for support and the build infrastructure that helps pinpoint custom artifact dependencies on deprecated Cúram artifacts.

Overview

In enhancing Cúram, the necessity occasionally arises where a feature or API is deprecated. It might be necessary to replace the feature or API with an updated version, or to remove the feature where a better alternative is available, or where features are unused or no longer consistent with the strategic direction of the product.

When a feature or API is deprecated, IBM will continue to support the feature or API. However, IBM no longer plans to enhance the feature or API, and might remove the capability in a subsequent release of the product. Deprecation can happen in code, in documentation, or in both, where all methods are valid and equivalent.

At a technical level, deprecation means that an artifact will no longer be enhanced and might be removed in a future release. In Java terms deprecation frequently means that the API is replaced by an alternative. In such cases, all calls to the deprecated API in a default installation are replaced by calls to the replacement API. Deprecation can also mean notice to remove without replacement. In all circumstances, a deprecation comment reflects the intended action.

Deprecation in documentation means that within the product documentation in the IBM Knowledge Center, references to the deprecated feature are indicated as deprecated in all guides that discuss the feature. A prefix, "(deprecated)" and also the deprecated icon will be added to the names of deprecated sections, and also within sections that discuss other functionality and refer to the deprecated feature.

Features can be deprecated in any Version Release Modification (V.R.M.) release, and can then be removed in a V.R.M release that follows the release in which the features are deprecated. The period between the two releases will be no less than 12 months. For example, if a feature was deprecated in Version 7 that was released in December 2016, then IBM can remove the deprecated feature in a V.R.M. release in or after December 2017. Features will not be deprecated in refresh packs, or other maintenance releases.

No further notice will be given after deprecation before the feature is removed, though in some cases a feature might not be removed for a number of releases.

If customers have concerns about the removal of a feature because they need more time to either find an alternative solution, or to break their dependency on the deprecated feature because of their project timelines, they should raise this concern through the usual support channels.

Other Sources of Information

Information about specific deprecated artifacts can be found in the artifact itself and also in the *'Notes on Deprecation'* section of the Cúram release notes.

In the artifact itself, the deprecated element will be marked as described in Artifact 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 from looking up the deprecated artifact. However, this enhanced build warning is only available from Cúram compilers and validations, the command-line Java compiler does not have equivalent functionality. It is recommended you view Java warnings in your Integrated Development Environment (IDE) for fast navigation between artifacts.

If the information in the artifact'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 artifact or by the release note ID referenced in its deprecation comment.

Effect of Deprecation on a Custom Application

In Cúram, a *deprecated* artifact means an artifact that will be removed in a later version of the product. Artifacts can be deprecated if there is a new better alternative, however, they can also be deprecated if they form part of a feature that IBM intends to remove from Cúramo for which there is no replacement. In both scenarios, the deprecated artifacts will no longer form part of the default flow within Cúram. Deprecated artifacts remain present in the application codebase, but they are not referenced by the out-of-the-box runtime application.

To quickly pinpoint where custom dependencies exist on deprecated Cúram artifacts, 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.

Customizations and References

Custom artifacts can depend on deprecated Cúram artifacts 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 User Interface Metadata (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 artifact.

Where code *references* a deprecated Cúram artifact (for example, calls a deprecated method), the deprecated artifact 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 artifact, the base Cúram application no longer invokes that artifact - 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 artifacts 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 (for example, if equivalent functionality has since been implemented), re-applying the customization to the artifact that replaces the deprecated one, and so on.

The deprecation build infrastructure provided uses special tags in deprecation warnings to help distinguish between references-to and customizations-of deprecated artifacts. This will be described in more detail later in this chapter.

Support for Deprecated artifacts

Deprecation of an artifact is an indication of the intent to remove it in a future version. Deprecated artifacts will be supported as long as they exist in the product. If customers have concerns about the removal of an artifact, because they need more time to break the dependency on that artifact, for example, they should raise this concern as soon as possible through the usual support channels.

You are advised to address any dependencies from custom code on deprecated Cúram artifacts at the earliest opportunity. When deprecated artifacts are removed in a future release, it can cause compilation failures and this seriously can hamper effective planning of upgrade tasks.

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. The default flow is also affected when client pages are deprecated and not replaced. This has two results that do not occur when other artifacts are deprecated:

Consistency of the User Interface: If existing client pages have been customized or new pages added that 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.

If out-of-the-box client pages have been deprecated and are not being replaced, that is, they form part of a feature that will be removed in the future, the links to those pages are removed from the user interface and not replaced.

Documentation/Training Materials: If descriptions or screen shots, or both, of the deprecated pages have been included in custom documentation or training materials, these may need to be updated to describe or show the replacement pages. If links to deprecated pages have been removed from the user interface, new screen shots may be required to reflect that fact.

Reinstating Deprecated Functionality

For features that have been deprecated without replacement, links to deprecated pages within those features will have been removed from the user interface.

Customers wanting to reinstate these links can do so by overriding the necessary application configuration files and re-adding the necessary links. For customers who already had the feature in an earlier release, the use of a diff utility will help expedite this process.

Scope

Artifact types that can be deprecated

Use this information to understand the types of artifacts that can be deprecated.

The following artifact types can be deprecated:

Area	Artifact Type
Table 35. Artifact types that can be deprecated	
Area	**Artifact Type**
Modeled Artifacts	Process Class, Struct Class, Process Method, Entity Method
Java Code	Identical to Java deprecation (Class, Interface, Method, Attribute, and so forth)
Javascript	The javascript class or operation can be tagged as deprecated.
Client Artifacts	UIM Page, VIM file, Page Property (.property that is associated with a UIM or VIM file)
Application Configuration files	Tab, Menu, Nav, Section and Shortcut files can be deprecated if they relate to features that are intended to be removed in the future. The same goes for their associated properties, or for properties that relate to entries that have been removed from an application configuration file.
Messages	Message Catalog Entry
DMX	DMX files or entries that relate to features that will be removed in the future.
Rules	Rule-sets, and their associated properties
Properties	Environment Variables - for these a comment is added to the description and the display name is updated to make it obvious that this property relates to deprecated functionality, for example, <Existing display name> - Deprecated. For entries inside adapter.properties and ShortNames.properties, a comment highlighting the fact that this property relates to a deprecated entity is added. At the point that the entity is removed from the application, the property will also be removed.
Events	Event Handlers, .evx entries relating to features that will be removed in the future.
Workflow	Process definitions

All of these artifact types support explanatory comments attached to the deprecation tag. These artifacts can be found easily by searching for the string "deprecated" within the artifact in question. For `.java` files (and model artifacts), the **`@deprecated`** JavaDoc tag is used in the normal way. For XML files such as User Interface Metadata UIM/VIM files and message catalog entries, the **`<?curam-deprecated`** XML processing instruction is used. Finally, in property files, the string **`.deprecated`** is appended to the name of a property to denote that the property is deprecated.

Entity Classes: Entities are not listed above as modeled artifacts that can be deprecated. The rationale for this has to do with the fact that the DDL is not generated for entities which have been tagged as deprecated in the model. This can result in unwanted impact. For example, existing unit tests that write to those entities will fail to execute if the tables do not exist on the system. Even though the entities are not physically tagged as deprecated in the model, they will be listed as deprecated in the external release notes if the intention is to remove the entity in the future. Deprecation of an entity relates to the entity itself, which includes its generated entity and key structs as well as its entity operations. It does not refer to data associated with the entity. As with all other deprecated artifacts, customers should remove dependencies on deprecated entities at their earliest convenience.

Limitations of the deprecation infrastructure
Use this information to understand the limitations of the deprecation infrastructure.

Users need to be aware of certain limitations of the deprecation infrastructure, which are:

- No build warnings are produced for non-typed references to deprecated artifacts. For example, if the User Interface Metadata (UIM) page `Participant_viewAddress.uim` was deprecated and a Java method contained a `Participant_viewAddress` string literal - this string 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 composed of a deprecation tagging capability and build/validation warning capability (reporting dependencies on tagged artifacts). The build/validation warning capability is intended for customer use. Therefore, the deprecation tagging capability is not intended for customer use and is not supported. For example, the **<?curam-deprecated** processing instruction in custom XML files is not supported.

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 artifacts. It also helps distinguish between references-to and customizations-of deprecated artifacts in custom code.

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 `bootstrap.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.

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

Generating the Deprecation build output

Execute the build target that follows to capture the build output to a `%SERVER_DIR\buildlogs\ %Deprecation<timestamp>.log` file for further analysis.

- **cd %SERVER_DIR%**
- **build deprecationreport**

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 already have confirmed that the server, client, and database builds completed 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, 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 reporting success or failure at the end of the build, so the deprecation information still will be produced for all files that pass validation.

Finally, to get a summary report of all exposure to deprecated artifacts, filter the `deprecation.log` for the [deprecation] tag. You can use **grep** or the Windows **find** utility for this, or your preferred text editor.

For example:

```
grep "\[deprecation\]" Deprecation<timestamp>.log
1> deprecation_summary.log 2>&1
or
find "[deprecation]" Deprecation<timestamp>.log
1> deprecation_summary.log 2>&1
```

Figure 77. Getting a Summary Report

Tip: The resulting `deprecation_summary.log` file will contain only the deprecation warnings produced by the build.Since some warnings can be broken over more than one line, it also is useful to hold on to the original `deprecation.log`.

Notes on running the Deprecation Report

- This build can take some time to run, as it has to do a clean build followed by server and client builds, 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 run in isolation. After fixing all the validation issues, the **deprecationreport** target should be executed to ensure the deprecation build log is complete.

- The **deprecationreport** calls the **validation** target. For example, the **deprecationreport** will fail if the **validateallworkflows** target reports 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.

- Intelligent Evidence Gathering (IEG) scripts also can contain dependencies on server or client artifacts or both 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 inspect manually the User Interface Metadata (UIM) page and server interface references to identify any dependencies on deprecation pages or interfaces.

Note: Since some warnings can be broken over more than one line, it also is useful to hold on to the original `deprecation.log`.

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.

Identifying overrides of deprecated artifacts

As described in "Customizations and References" on page 112, there are significant differences between the effects of deprecation on references and on customizations. Identifying overrides of deprecated artifacts relatively is simple. The deprecation summary report you produced in "Running a Deprecation Report" on page 115 pinpoints all dependencies on deprecated artifacts using the standard Java [deprecation] tag in the build log. Cúram code generators and command-line validations also check for dependencies on deprecated artifacts and produce the same build warning as Java (using the same [deprecation] tag).

In addition to this, Cúram code generators augment the [deprecation] tag with an additional [customization] tag where your custom artifact is overriding a Cúram artifact, rather than referencing it.

Any lines in your deprecation summary report tagged with [deprecation] [customization] indicate places where you are overriding an artifact that Cúram has since deprecated (that is, removed from the default flow of the base application). Since Cúram has removed this artifact from the default flow of the out-of-the-box application, it reasonably is likely that it also has been removed from the flow of your custom application. Where this happens, it will be necessary to address the override.

The example that follows shows a custom VIM file that is overriding an out-of-the-box Cúram VIM file. The Cúram 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 artifact, followed by the [deprecation] tag and, in this case, a [customization] tag also. This is followed by the name of the artifact that has been deprecated. Finally (and this differs from the Java format) where possible, any comments attached to the deprecated artifact 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 Cúram 6.0,
replaced with Case_listAnotherView.vim. See release note:
CR12345
```

Figure 78. Example: override of a deprecated artifact

In the preceding example, the VIM file no longer is used in the default flow of the out-of-the-box Cúram application. If your application relies on the out-of-the-box flow, your customization of this file no longer will appear in that flow.

Addressing overrides of deprecated artifacts

There is no single approach to addressing overrides of deprecated artifacts. You must analyze the modifications you made to the original (now-deprecated) artifact and determine a suitable course of action for your customization. Some options are to drop the customization (for example, if Cúram has since implemented equivalent functionality), to re-apply the customization to the artifact that replaces the deprecated one, and so on. There are sources of information that can help when deciding the appropriate course for your customization.

For more information, see, "Other Sources of Information" on page 112

Identifying references to deprecated artifacts

References (for example, calls to) to deprecated artifacts also can 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 Cúram 6.0, replaced with ProductDelivery_cancelDeduction1
```

Figure 79. Example: reference to a deprecated artifact

In the previous example, the UIM page is no longer used in the default flow of the out-of-the-box Cúram application and is deprecated.

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.

User Preferences

To specify settings that can be customized for a particular user, configure user preferences. 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:

Table 36. Out-of-the-box user preferences		
Name	**Description**	**Default Value**
Time Zone	The user's time zone.	Europe/Dublin

User Preferences Definition

Data definition XML file

It is possible to create new user preferences, or override existing user preferences, by creating a custom `DefaultPreferences.xml` file.

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>
```

Figure 80. 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 should be displayed in the user preference editor in the web client for an internal user, i.e. a user on the Users table.	no	true
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 37. 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.

Properties files

When defining a user preference in the `DefaultPreferences.xml` file a corresponding entry should also be made in an accompanying `DefaultPreferences_<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 "Localizing display names" on page 120 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 `EJBServer\components\<component_name>\userpreferences` directory 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 `SERVER_COMPONENT_ORDER` merge rules that apply to `DefaultPreferences.xml` files.

Development support

User Preferences can be accessed at development time by using the `getValue()` and `setValue()` methods in the `curam.util.userpreference.impl.UserPreference` class.

A user preference must have been created previously prior to starting the `setValue()` method. For more details on creating user preferences, see "User Preferences Definition" on page 118 .

External users

To make user preferences available to an external user, you need to make both client and server changes. These changes are described as follows.

For the client, you need to set the USER_PREFS_PAGE attribute to `true` within a *<link>* element. For more information about setting this element, refer to the Cúram web client reference.

The `ExternalAccessSecurity` interface is used to retrieve information for an external user. This class contains two new methods, **getUserPreferenceSetID()** that reads user preferences for an external user and **modifyUserPreferenceSetID()** that updates user preferences for an external user. These methods must be implemented to retrieve user preferences for an external user. For more information on the `ExternalAccessSecurity` interface, see the Customizing External User Applications chapter in the *Cúram Security Handbook.*

After the client and server changes are implemented, you must ensure that the relevant user preferences are visible to the external user. The *<externalVisible>* element within the `DefaultPreferences.xml` allows you to manage the visibility of each user preference to an external user. This element is described in "User Preferences Definition" on page 118.

If you want to make user preferences available for external users and *<externalVisible>* is set to *false* or is not defined for all user preferences, then user preferences are not displayed for an external user. If you do not want to display any user preferences for external users, it is suggested that the User Preferences link not be available within the external user application.

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`. The *<locale>* component 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, create the following default preference properties file

```
      DefaultPreferences_en_US.properties
```

As there can exist multiple `DefaultPreferences_<locale>.properties` files in different components, the contents of these default preference properties are merged to a `MergedDefaultPreferences_<locale>.properties` file according to the SERVER_COMPONENT_ORDER[7]. This merging happens when running either the **mergeuserpreferenceproperties**, or **server** targets.

Before merging the `.properties` files, the following validations cause an error during a build where:

- The specified *<locale>* is not present in the SERVER_LOCALE_LIST[8].
- 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 attempts to display the correct localized name by matching the country part and language part of the user's locale. If the country component of the user's locale does not exist, the infrastructure attempts to match the language part only. If this data does not exist, it falls back to a default language. The localization of display names is illustrated in the example that follows.

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 searches for a `MergedDefaultPreferences_en.properties` file where locale is set to the

---

[7] See "Customizing a Message File" on page 84, for further explanation of SERVER_COMPONENT_ORDER.
[8] See "The Format of Message Files" on page 82, for further explanation of SERVER_LOCALE_LIST.

default system locale. If the display name is not present, the system falls 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 that is defined for the *name* attribute for a user preference in the `DefaultPreferences.xml` file is used as the display name. For more information on the *name* attribute, see "User Preferences Definition" on page 118 .

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`

Localizing infrastructure preferences display names

The application uses a number of Infrastructure Preferences 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 localization, can be found in `SDEJ\lib\InfrastructurePreferences.properties`.

SPM runtime behavior

Use the information as a starting point for learning about how the IBM Cúram Social Program Management application behaves at runtime, including what you need to know about transaction control, how a cache stores the results of an SQL query, and how deferred processing works in Cúram.

- – "Transaction control" on page 122 details the aspects of Transaction Control within a Cúram application that must be understood by a developer.
  - – "Use of the transaction SQL query cache" on page 124 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.
  - – "Deferred processing" on page 127 describes how to achieve deferred processing in a Cúram application
  - – "Cúram timers" on page 131 describes the functions that allows timers to be defined to start client-visible methods at a specified time.
  - – "Events and event handlers" on page 136 describes Events, a mechanism for loosely coupled parts of the Cúram application to communicate information about state changes in the system.
  - – "Unique IDs" on page 143 details the infrastructure support for Unique Identifier numbers generated by the Cúram infrastructure for use as unique database keys.

Transaction control

Use this information to understand how the Cúram Server Development Environment (SDEJ) abstracts transaction management from the developer.

This section provides a brief overview for the developer and then details what is happening "under the hood". This task is difficult task because of multiple database support, which provides different ways of supporting the ACID nature of a transaction. A transaction might be Atomic. Its result seen to be[9] be Consistent[10], Isolated[11], and Durable[12].

Transactions and method invocations

Use this information to understand how Cúram maps Facade method invocations to transactions.

Typically in Cúram a Facade method invocation maps to a single transaction. The exception to this action is where the method starts 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 check ensures that long running transactions do not place locks on the Key Server tables as this condition would provide an unacceptable bottleneck.

Optimistic locking and the `forUpdate` flag

When a developer creates operations on an entity they first must determine whether that entity supports optimistic locking.

Optimistic locking is described in Optimistic locking for concurrency control and provides a suitable method of ensuring that transactions satisfy the characteristics of Atomicity, Consistency, Isolation, and Durability (ACID). However, situations occur when the use of optimistic locking can impact unnecessarily 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 changed underneath the developer. Rather than using the version number, it often is more suitable to lock the record when it is read. This precaution means that it is impossible for another transaction to change the record, so the modified record does not need to be guarded with a version number. However, it also means that the possibility increases for locks and deadlocks.

This form of locking is supported in Cúram by way of an extra parameter that can be passed to any of the standard read operations. This parameter (forUpdate), when set to *true*, results in an update lock to be taken on the records that are being accessed as part of this query. These locks are not released until the end of a transaction.

General guidelines

The golden rule of locking and performance in database transactions is that any records that you lock need to remain locked for the minimum possible time to reduce database contention that is caused by other users who are seeking the same records.

This rule means that operations that take out locks might 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 validation the transactions first followed by the updates. Try to defer

---

9 Atomicity requires that all of the operations of a transaction are carried out successfully for the transaction to be considered complete. If all of a transaction's operations cannot be completed, then none of them can be carried out.

10 Consistency refers to data consistency. A transaction must move the data from one consistent state to another. The transaction must preserve the data's semantic and physical integrity.

11 Isolation requires that each transaction seem to be the only transaction currently manipulating the data. Other transactions might 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.

12 Durability means that updates that are 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.

update operations (or reads with locks) until as late as possible. Do not scan a million-record table after you take out a record lock that ought to be short-lived.

Underlying design

The information in this explanation describes the underlying design of transaction management.

Transaction management happens on the server, rather than the client side. Client-initiated transactions would involve complicated and largely unnecessary communication processor load. However, this condition imposes a requirement on the application to ensure that the database data remains consistent across a series of client/server calls. In practice, this condition usually involves deferring the database updates done by a business function until the last client/server interaction in a series.

Transactions typically must encompass interactions with more than one resource manager even if older systems are not used. The server database is one resource manager and the queues that are used for deferred processing and workflow are another. To ensure atomicity of a transaction that is distributed across multiple resource managers, a two-phase-commit protocol is required to coordinate the distributed transaction.

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 connection ensures that dirty data is not read and that a second read returns the same data as a first.

Specific categories of statements that run at a lower isolation level:

- Cursor Stability - Cursor stability is the DB2 implementation of the SQL standard `Read Committed` isolation level. This statement ensures that a transaction cannot read a row with uncommitted changes in it. It does not ensure that a second read returns the same data as a first.

This connection is not a separate connection to the database, rather the DB2 keyword WITH CS is appended automatically to the **SELECT** statement.

All queries that 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 the Repeatable Read isolation level. This check means that they place a lock on the row or rows that were read so that they cannot be updated by anyone else. In the case of **modify** operations be read by anyone else. This lock is not released until the transaction commits.

Oracle

Oracle does not really support the Java Data Base Connectivity (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, a dirty read occurring is not possible.

Use of the transaction SQL query cache

Use this information to understand the transaction SQL query cache in the Cúram Server Development Environment (SDEJ).

Benchmarking shows that the same database query often is run numerous times during a single transaction in the Cúram application. This behavior is costly in performance terms. The transaction SQL query cache in the SDEJ counteracts this .

The transaction SQL query 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 during the transaction in which the operation was started. Subsequent calls in the same transaction retrieve the required results from the SQL query cache and does not read the results from the database.

How results get stored in the query cache

Use this information to understand how SQL query results get stored in the query cache.

The SQL query cache stores the results in memory of any SQL query that runs a **SELECT** statement on a database table. Invocation of the following entity operation stereotypes results in the responses to that query that is stored in the cache:

- **read**
- **nsread**
- **nkread**
- **readmulti**
- **nsreadmulti**
- **nkreadmulti**
- **nsmulti**
- ns with handcrafted SQL containing a **SELECT** statement

Two exceptions to this rule are:

- SQL queries that have the FOR UPDATE flag set to true do not have their results cached. These queries always result in direct database access. This action is because this data is being read for modification and the subsequent update operation results in that cache entry to be invalidated.
- The results of specialized readmulti operations, where the operation is not an instance of StandardReadMultiOperation class, are not cached. This action is because a customized ReadMultiOperation can modify the result set for an SQL query that is run. Since these results are not yet present in the cache, the cache cannot be invalidated which results in invalid data in the cache (that is, the data that is cached for the SQL query does not reflect the data for that SQL query on the database).

How the cache gets invalidated

Use this information to understand when an SQL query cache that is associated with a transaction is invalidated

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 that is associated with that transaction is invalidated.

Any time an update (that is, 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 (that is, **modify**, **nsmodify**, **remove**, and similar commands), the invalidation of cache entries is partially intelligent. The table that is affected by the update is determined from the SQL statement that is run and used to directly invalidate only the cache entries that are related to the table. However, for ns operations that are run and contain anything other than a **SELECT** SQL statement, the complete SQL query cache that is associated with that transaction is invalidated.

Therefore, the following entity operations cause the cache entries that contain the table that is 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 **SELECT** statement

- **batchinsert**
- **batchmodify**

As detailed previously, the transaction SQL query cache endures for the lifetime of a transaction only. Database updates result in the invalidation of associated entries in the local transaction cache only. As a result, any subsequent reads within a different transaction returns data from the cache and not as updated on the database.

How to set the property for the transaction SQL cache

Use this information to understand how to set the property for the transaction SQL cache.

The transaction SQL cache is enabled by default, meaning that the results of SQL queries are 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 can lead to memory problems in transactions that endure for a long period. The most likely queries that might lead to such problems are those queries that return data of type Character Long Object (CLOB) and Binary Large Object Block (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 might be stored in the transaction SQL query cache. This property is called `curam.sqlquerycache.lob.max.size` and its default size is set to 500KB.

Further details that concern these properties can be found in "SPM configuration parameters" on page 147.

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.

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.

Logging

Use this information to understand how lifecycle events that concern the transaction SQL query cache are logged under certain tracing levels for the Cúram application.

When the tracing level for the Cúram application is set to `curam.util.resources.Trace.kTraceUltraVerbose` (see "Logging trace levels" on page 67 for more details on logging), various lifecycle events that concern the transaction SQL query cache are

logged. These entries might 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 that is either committed or rolled back.

Deferred processing

Use this information to learn how to implement deferred processing for appointed Business Process Objects (BPOs) in your Cúram application.

Before reading the following chapter, you need to be familiar with the *Cúram Modeling Reference Guide* and the Server Development Environment (SDEJ).

In Cúram, describing a business process method as a "deferred process" means that this method is started 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. Any request over the queued enactment interface is deferred.

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 start a deferred process method by using these parameters. These parameters are passed to the method by the deferred processing server after 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 parameter gives the deferred process method access to any information it requires to initiate the required processing (long datatype).
- A **boolean** flag. This parameter is internal to the deferred processing infrastructure. It needs to 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
}
```

Figure 81. wmdpactivity stereotype method

The previous example 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 after the enactment request). Still, you must code the logic of your deferred process into this method.

Note: Your deferred process should not attempt to initiate any begin, commits, or rollbacks by using the **TransactionInfo** class or attempt any other forms of Java EE Transactional Control. This restriction also applies to any methods that are started by workflows or deferred processes, regardless of how deep in the call stack. In addition to deferred processes, examples of the methods include implementations of workflow or deferred processing interfaces (such as **NotificationDelivery**, **WorkResolver**, **curam.util.deferredprocessing.impl.DPCallback**, and similar) and any methods called by either of the previously referenced commands.

Deferred process enactment

Deferred processes are enacted by way of the Deferred Processing Enactment Service.

Consider the situation where a Business Process Object (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 "Deferred process enactment" on page 128. Within the calling BPO, populate a `WMInstanceData` record (see "WMInstanceData" on page 129 for 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 IBM Cúram Server Development Environment (SDEJ).

```
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);
}
```

Figure 82. Using DeferredProcessing startProcess

"Deferred process enactment" on page 128 shows a Cúram application BPO that calls a deferred process method. The key points to note are that the `WMInstanceData` record is set up as part of the calling BPO implementation. The **DeferredProcessing.startProcess()** command then is used to request the enactment of the deferred process method. The parameters of this method are as follows:

1. The name of the deferred process method that is 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 "Configuration of Deferred Processing Table" on page 129.

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. If an error occurs, the error handler that implements the `curam.util.deferredprocessing.impl.DPCallback` interface must be started. If the parameter is not provided, the global error handler that is configured through the `curam.custom.deferredprocessing.dpcallback` property is started.

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.

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.

Table 38. WMInstanceData Properties

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

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`.

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.

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.

details the properties of the `DPProcess` table.

Table 39. DPProcess Properties

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
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

"Configuration of Deferred Processing Table" on page 129 shows an example of how a DPProcess table might be populated.

Table 40. Example DPProcess Table

processName	interfaceName	methodName	ticketType	Subject
DO_DEFERRED_ OPERATION	server.curam. bizinterface. SomeProcess	doSomething	CLAIM	This method does something.
DO_ANOTHER_ DEFERRED _OPERATION	server.curam. bizinterface .SomeOther Process	doSomethingElse	CASEREVIEW	This method does something else.

Error Handling

The Deferred Processing Engine provides an error handling callback mechanism when deferred processes fail, that is, the deferred method you defined throws an exception. The curam.util.deferredprocessing.impl.DPCallback interface is provided with the infrastructure. It has a single method definition: dpHandleError.

Note: The curam.util.deferredprocessing.impl.DPCallback interface should not be confused with the curam.core.impl.DPCallback interface.

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).

There are two ways an error handler can be configure:

1. Define a single (global) error handler callback for deferred processing by setting the `curam.custom.deferredprocessing.dpcallback` property to the fully- qualified name of a class that implements the `curam.util.deferredprocessing.impl.DPCallback` interface. The `dpHandleError()` method on that class is invoked when any deferred method fails.

2. Supply the fully-qualified name of any class that implements the `curam.util.deferredprocessing.impl.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.

Figure 1 shows an implementation example:

```
void dpHandleError(String processName, long instDataID)
  throws AppException {
  // Method logic goes here
}
```

Figure 83. DPCallback.dpHandleError implementation example

This callback operation could be used to:

- Notify the client that a deferred process failed.

- Take some remedial action.

Security

Deferred processes run under the user name SYSTEM, so the effective locale for deferred processes is the default locale for this user as specified in `defaultLocale` field on the `Users` table.

In the case of offline unit-testing of deferred processes, the user name is blank and the effective locale is the default locale for the Cúram server.

Deferred Processing summary

- The incorporation of Deferred Processing into your application is achieved largely by:

  – Modeling appointed Business Process Object (BPO) methods with **<<wmdpactivity>>** stereotype.

  – Configuring the `DPProcess` table in your database.

  – Using the `DeferredProcessing` to request deferred process methods.

- The appropriate deferred processing queues must be set up before run time by following the steps given in the *Cúram Installation Guide*[13].

- Application-specific error handling can be achieved by using the **DPCallback.dpHandleError()** method. An error handler then can be targeted in the code by passing the error handler class name when you start the **DeferredProcessing.startProcess()** method.

Cúram timers

You can use the Cúram timer bean to start timers that start client-visible methods at a specified point in the future.

For deployed applications, Cúram timers use the Java Platform, Enterprise Edition timer service that is provided by the EJB container. When you run applications that are outside an application server environment, the timer functionality is provided by the `java.util.Timer` JDK class instead of the EJB timer implementation of the application server. For more information about the `java.util.Timer` class, see the JDK documentation.

Java Platform, Enterprise Edition 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. You can use the timer service

---

[13] Refer to the installation guide for your particular operating system; that is, Windows or UNIX.

of the enterprise bean container 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 might set timers to go off at 10:30 AM on 23 May, 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 can occur on a regular schedule. Essentially, three types of timers are possible, as outlined in the following table:

Table 41. Types of timers

Type of timers	Description
Single-event timer.	A single-action timer that expires after a specified duration.
Single event with expiration date.	A single-action timer that expires at a specific point in time.
Interval timer with initial expiration duration.	An interval timer where the first expiration occurs after a specified duration, and where the subsequent expirations occur after a specified interval.
Interval timer with initial expiration date.	An interval timer where the first expiration occurs at a specific point in time and subsequent expirations occur after a specified interval.

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`

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:

Table 42. List of API's in `TimerInfo` Class

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.
`startTask(DateTime, TimerTask)`	Create a single-action timer that expires at a given point in time.
`startTask(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 which were invoked by a timer, calling this method from a non-timed method will have no effect.

Table 42. List of API's in *TimerInfo* Class (continued)	
Method Name	**Description**
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.

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.

Table 43. List of parameters from *TimerTask* 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 specified when this timer was created, or zero if it is a one event timer.

TimerCallback interface

TimerCallback is an interface for which developers can provide an implementation and the interface is started if a timed operation fails. The interface `curam.util.timer.TimerCallback` has only one method `handleError(Exception, TimerTask)` defined and users can provide an implementation of this method.

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);
```

Rules for using Cúram timers

There are some considerations and limitations to Generic Timer Bean provided as part of Cúram infrastructure and they are listed below.

1. Cúram 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. Cúram timers do not have any facility to return a value from an operation.

3. When deployed in an application server, timer creation and cancellation are transactional. For example, 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. When deployed in an application server, timers are persistent and remain active until they are canceled 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. To enable developers to use and test timer related functionality, when the application is deployed timers in the SDEJ are provided by the J2EE `javax.ejb.TimerService` class. Similarly, when the application runs in the development environment, timers in the SDEJ are provided by the JDK `java.util.Timer` class for testing purposes only. However, the `java.util.Timer` class has the following limitations:

   - The `java.util.Timer` class is not transactional. For example, if you start a timer and then roll back the transaction, the timer stays active instead of being rolled back.

   - The `java.util.Timer` class is not persistent. For example, the `java.util.Timer` class does not resume if you stop and restart the JVM.

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.

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 using 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 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?

  Yes. However, when you use timers inside the development environment, the timer is provided by the JDK `java.util.Timer` class that has the following differences to the Java Platform, Enterprise Edition timer:

  – The `java.util.Timer` class is not transactional.

  – The `java.util.Timer` class is not persistent.

- How can I debug timers?

  To isolate the Cúram task trace output by the task identifier, set the Cúram trace level to `trace_verbose` (`curam.trace=trace_verbose`) and configure a log4j appender for the 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.

Events and event handlers

Use this information to understand events and event handlers. Events provide a mechanism for loosely coupled parts of the 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 that occurred, provided they are registered as listeners for that event.

To use this function, some events need to be defined, some application code must raise these events, and some event handlers need to be defined and registered as listeners to such events. Developers must write and register event handlers (classes that initiate an action when an event is raised) and optionally event filters (logic that determines whether to start the handler for a specified event). Event handlers and filters are classes that implement callback interfaces in much the same way as in the classic observer pattern[14].

The Format of Event Files

Event definition

Events are defined in Cúram in XML files that specify both the event classes and the event types. Use this information to understand how to events are used.

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 in the example that follows:

```
<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>
```

Figure 84. Event definition file

events
> This element is the root tag of an event definition file under which all the event classes and types are defined.

> **package**
> > This element specifies the Java code package into which the Java constants for event classes and their types are generated.

annotation
> This element is optional and is specified for both event classes and types intended for descriptive text for the element. The text that is 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 element is the identifier of the event class for code generation and is the class name for the constant class that contains all the event types in the class. Since this element is a Java class name, it must be a valid Java identifier.

> **value**
> > This element represents how an event class is referenced at run time and it is this value that event handlers are registered against. This value needs to be unique in the system and is a 100-character string.

---

[14] 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.

event-type
> Defines an event type within a specified class. Since an event is identified by its own name and that of its parent class, an event type needs to be unique only within a specified class.

> **identifier**
>> This element is the identifier of the event type for code generation and is the field name for the constant containing the value of the event type. Since this element is a Java field name, it needs to be a valid Java identifier.

> **value**
>> This element is how an event type is referenced at run time and the value needs to be unique within a specified event class and is a 100-character string.

Event handler registration

Use this information to understand event handlers and how to register them against a particular event class.

Event handlers and their associated (optional) filters need to be registered against a particular event class to be started when an event of the specified class is raised. This operation 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>
```

Figure 85. Event handler registration file

registrations
> This element 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: "Event handlers" on page 142).

> **filter**
>> The fully qualified name of an optional event filter class (see: "Event filters" on page 142).

> **removed**
>> An optional attribute that is used by components of a higher precedence to disable previously registered event handlers, (see: "Rules of event handler merges" on page 139).

event-classes
> This element 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 started.

identifier
> This element identifies the event that the handler is registered against. This value corresponds to the *value* attribute of an *event-class* element in the event definition files.

How to merge event files

Use this information to understand how to merge event files with files included with your Cúram application.

Both event definition and handler registration files are in the `/events` directory of a component. The Cúram reference application includes a set of event files. These files can be augmented by placing new event files in the SERVER_DIR/components/<custom>/events directory, where <custom> is any new directory that is created under components that conforms to the same directory structure as `components/core`. This mechanism avoids the need to modify directly to the unmodified 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 leftmost has the highest priority, and the rightmost the lowest.

After changes are made to the component precedence order in **SERVER_COMPONENT_ORDER**, it is necessary to run a clean build to ensure that you are using the appropriate files. This procedure is done by starting **build clean server**.

Rules of event definition merges
For event definitions to be merged, the files that are provided to customize the events need to be named the same as the existing files that contain the event classes to customize. Use this information to understand how to merge event definitions.

For event definitions to be merged, the files that are provided to customize the events need to be named the same as the existing files that contain the event classes to customize. Placing event classes with the same name in files with different names results in errors that occur when the application loads the event definitions onto the database.

The customizing behavior for events is simple - events cannot be removed as existing functions might be using an event that other components then decide to remove. As a result, such code would 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.

Rules of event handler merges
The event handler (and filter) configurations that are used at run time are from the component with the highest precedence that specifies the event handler in question. Use this information to understand the rules for event handler merges.

The event handler (and filter) configurations that are used at run time are from the component with the highest precedence that specifies the event handler in question (for 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 that is specified by lower precedence components by setting the `removed` attribute of the `event-registration` element to `true`.

Artifacts produced by generate events

Use this information to understand two types of output generated by the **evgen** command.

Two types of output are generated by the **evgen** command: `.java` files (for code constants that use events less error prone) and `.dmx` files (database scripts for loading event definitions onto the database).

The Java artifacts that are produced from 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 artifacts to be placed in the /build/svr/events/gen/curam/events directory.

The database scripts that are produced from a merged event file are placed in the /build/svr/events/gen/dmx directory.

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 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>
```

Figure 86. 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>
```

Figure 87. Generated event type database script

Java event code example

Use this information to learn about how to generate a Java code generated constants file for an event class.

Events are identified in the system by their names as specified by the value attribute of the event-class and event-type elements. However, just using text in application code to reference events might be error prone. In particular, an event is fully identified by its type in addition to its class. Thus, using string literals to refer to an event might be ambiguous, as an event type is unique only when qualified by its associated event class.

The following code 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 procedure 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";
    }
  }
```

Figure 88. Generated event Java constants

How to raise an event

Raising an event is a matter of creating an event struct, populating it with data, then calling the event service API to raise the event. Use this information to learn how to raise an event

Raising an event is a matter of creating an event struct, populating it with data, then calling the event service application programming interfaces (API) to raise the event. The event infrastructure notifies any registered handlers that the event is being raised. How to raise an event is shown in the example that follows.

```
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);
```

Figure 89. Raising an event

eventKey
  This element 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 that is being raised: this element is the value on which handlers are registered.
  eventType
    The type of the event that is being raised: this element identifies the specific type of the event in the specific class.

primaryEventData

    This element is the primary payload of the event and is a 64-bit integer. Typically this element is (though not necessarily) the identifier of an entity in Cúram, the entity in question that is being identified by the class of the event. The event type commonly is used to indicate the action that takes place on the entity.

secondaryEventData

    This element is any additional data that might be associated with an event when it is raised. Unlike the primary event data, the secondary event data is optional.

Event handlers

Use this information to understand how to create an event handler.

How to register handlers was described previously. To create an event handler one needs to implement the interface: `curam.util.events.impl.EventHandler`, which is shown in the example that follows.

The action that is taken by an event handler when the event is raised is up to the developer. Event handlers are started synchronously when the event is raised (and hence run within the same transaction context as the code that raises the event). This action has two implications:

- Throwing exceptions from an even handler results in the transaction from which the event was raised being rolled back.
- Long running actions need to be avoided in event handlers as they affect the running time of the code that raises 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;
  }
```

Figure 90. Event handler interface

Event filters

The purpose of a filter is to decide whether the handler needs to be notified about the event that is being raised. Use this information to understand how to create an event filter.

As mentioned, an event handler can be configured to have a filter. The purpose of a filter is to decide whether the handler needs to be notified about the event that is being raised. To create an event filter, the user needs to implement the interface: `curam.util.events.impl.EventFilter`, which is shown in the example that follows.

If the `accept` method returns *true* the event is passed on to the event handler (that is the `eventRaised` method of the associated event handler is started), 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;
  }
```

Figure 91. Event filter interface

Unique IDs

Use this information to understand what Unique IDs are in the context of Cúram and how to use them in your application.

Unique IDs are numbers that are generated by the Cúram infrastructure for use as unique database keys. They come in two types:

- 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 that must be configured as part of a Cúram installation.

What Unique IDs are used for

Use this information to understand the purpose of Unique IDs and how to use them.

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 that is ensured to be unique throughout the application. One classic approach that is involved locking and updating a key control database table each time a key needs to be allocated. Unfortunately, this approach can lock the control table during 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 that use a unique ID generator, also known as the Key Server[15]. A process requests a block of Unique IDs by calling the key server. This action updates a database control table each time it returns a block of Unique IDs to a requesting process. After a block is allocated, the requesting process can allocate keys from this block locally; that is, 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.

However, it needs to be noted that a process that requests a Unique ID block might or might not use the keys from that block. If it does not, then the unused keys represent holes in the key sequence. For instance, processes that use one key value before they shut down leave large holes in the key sequence. Note also that no time limit exists on how long a process can wait between allocating a Unique ID block by using the key values in it. Thus, even for human-readable keys that are in an ascending sequence that starts 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 are better to not assume that this condition is the case.

The limit of allocating Unique IDs

Use this information to learn about the allocation of Unique IDs.

A process that used only one key out of each Unique ID block, and allocated 1000 of these IDs per second non-stop, would take more than 2 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.

---

[15] The design is loosely based on the Sequence Block pattern described by Floyd Marinescu in EJB Design Patterns (ISBN: 0471208310).

When Unique IDs need to be used

Use this information to understand when Unique IDs need to be used in your design.

Use Unique IDs in your design when each of the following criteria is met:

- You need a unique key for a database entity.
- The database key has no "business meaning".
- Instances of the entity might be created by multiple contending online or batch functions.
- Holes in the key sequence are acceptable (which always need to be true if the key has no business meaning).

When not to use Unique IDs

Use this information to understand Unique IDs are not to be used.

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).
- Your key requires a specification other than a simple numeric format.
- Contending processes do not create instances of the entity (in which case no need exists for key control at all).

Do keys need to be human-readable?

Use this information toil your keys need to be human-readable.

This decision is up to you. The general rule is that Unique ID values that are displayed to a user need to be human-readable. Otherwise, you can choose to use non-human readable Unique IDs. The advantage of these is that their values are spread across a large range, so that database indexes are not always being extended at the end, as for ascending sequences.

When contiguous human-readable Unique IDs are required

Use this information to understand when contiguous human-readable Unique IDs are required.

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 is 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 action is important for performance reasons).

In instances where a requirement exists 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` or `InternalEmployerID`). An example of such a business requirement is the issuing of Social Security numbers. These tables are read and updated in the context of the application transaction, meaning the ID is allocated only 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 this process causes performance high processor usage, 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 necessary and
- Your design needs to strive to ensure that transactions the use this mechanism are kept as short as possible to minimize contention on the key table.

The way to design Unique IDs

Use this information to understand the method for designing Unique IDs.

Designing Unique IDs into your Cúram application is straightforward. In your Unified Modeling Language (UML) application model, set the appropriate domain definitions to be of the data type SVR_INT64. The developer's view of this data type 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 allocates 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 configuration can be done by creating a Data Mining Extensions (DMX) file that defines all key entries. "The way to design Unique IDs" on page 145 details the fields of the KeyServer database table.

Table 44. KeyServer Database Table

Field	Description
keySetCode	An identifier for the key set; for example, MYKEYSET.
nextUniqueIdBlock	The next Unique ID block t0 be allocated. For human-readable IDs, this field can be used to skip preallocated Unique IDs.
humanReadable	True if the Unique IDs are to be human-readable.
lastUpdated	The time stamp for when the entry was last updated.
strategy	Represents the strategy that is used to generate next Unique ID block for a particular key set.
Annotation	A description of the key set.

If you are using human-readable Unique IDs, and non-Cúram-generated keys already have been allocated, then you can ensure that these values are never reallocated by Cúram (that is, Unique IDs never "clash"). This condition 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 that were allocated previously.

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` is used to generate next Unique ID block for the key set. The `Range Aware Key Server` is explained in more detail in "Overview of the Range Aware Key Server" on page 145.

Warning: Care needs to be taken when custom key sets are defined and used. The same key set always is used when Unique IDs are used 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 might 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 only can support numbers between `Long.MAX_VALUE` and `Long.MIN_VALUE + 1`.

Overview of the Range Aware Key Server

Use this information to understand how to use the Range Aware Key Server.

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 composed of a number of entities. Each of these entities has a

primary key. The standard Key Server implementation ensures uniqueness of a primary key within a single system installation. This action means that when a Business Object is transported from a Source System and applied on a Target System, a strong possibility exists for 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 that the existing entity is updated with the transported entity data.

Overview of the Range Aware Key Server
Use this information to understand how to use the Range Aware Key Server.

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 composed of a number of entities. Each of these entities has a primary key. The standard Key Server implementation ensures uniqueness of a primary key within a single system installation. This action means that when a Business Object is transported from a Source System and applied on a Target System, a strong possibility exists for 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 that the existing entity is updated with the transported entity data.

How the Range Aware Key Server generates primary keys
Use this information to understand how the Range Aware Key Server generates primary keys by using key block ranges, and how key block ranges are configured.

The approach that is used by the Range Aware Key Server to generate primary keys hinges on ensuring that non-overlapping key block ranges are allocated to every system. The Range Aware Key Server then ensures that all of the primary keys on a particular system are generated from the range or ranges assigned to that system. Therefore, the primary keys that are generated by each system are unique.

Key block ranges

At system installation or upgrade time, a system administrator allocates a unique primary keyrange from which all of the primary keys provided by the Range Aware Key Server implementation are generated. The system administrator specifies the key block range by using two components: a group number and a range number.

In general, key block ranges must be configured on systems before they are first started. An exception is where the default key block range is used for a system. If no key block range is configured, a default key block range of *group 3, range 2* is used.

After a system is started for the first time, entities that are drawn from the key block range might be created. Therefore, to avoid issues, the key block range that is allocated to the system cannot be removed.

Multiple key block ranges

Key block ranges are unlikely to be exhausted in normal usage. Each *group, range* pair contains over 4 billion range block. In normal usage, 256 keys are allocated from each key block, therefore the number of keys that are available from each range allocation is large. In addition, each business object generally

uses a separate key set, or collection of key sets, and effectively has a separate key counter for use by its entities. This means that keys are used up at an even slower rate that usual.

To cater key block range exhaustion, a system administrator can configure a system with multiple key block ranges. If necessary, these additional key block ranges can be added later, for example, after the system is first started. If a system is configured with multiple key block ranges, it starts using key blocks in the additional ranges only when all of the key blocks in the original range are used up.

After a system is configured with a particular key block range, it is not possible to remove that key block range from the system.

Group number and range number

The key block range group number is a number between 3 and 32,767 inclusive. Group numbers 1 and 2 are reserved for existing data and for application usage, so customer configurations must start with group 3. The range number is a number between 1 and 512 inclusive. Each system is configured with a unique group range pair.

Related information

Configuring key block ranges

Where to use the Range Aware Key Server
Use this information to understand where it is needed to use Range Aware Key Server.

The Range Aware Key Server is used only for Key Sets that are created specifically for the entities that form part of transportable Business Objects. Existing Key Sets continue to use the current Cúram Server Development Environment (SDEJ) Key Server implementation, unchanged.

Note: it is important that existing Key Sets are not changed to use the Range Aware Key Server - the Range Aware Key Server should be used only 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.

SPM configuration parameters

You can set configuration parameters for IBM Cúram Social Program Management applications that control characteristics of how the application is run. Generally, and unless otherwise noted, these parameters are set in `.property` and `.prx` files that are associated with your application.

The configuration parameter descriptions are organized according to the file in which they are set and in functionally related groups. Some parameters are of a "BOOLEAN" type, where noted. BOOLEAN means that the value "true" or "yes" in uppercase, lowercase, 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.

Bootstrap.properties

The following properties relate to the `Bootstrap.properties` file.

Select a properties category.

- Database
- Environment
- Test
- Custom

Database

These settings configure IBM Cúram Social Program Management for database communication.

Table 45. Database settings		
Property Name	**Type**	**Meaning**
`curam.db.type`	STRING	The property that specifies the database type. Valid values are: DB2, ORA, or 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 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 that specifies the name of the table space used for 32k storage on DB2 z/OS.

Table 45. Database settings (continued)

Property Name	Type	Meaning
`curam.db.zos.enable foreignkeys`	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 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.disableforeignkeys`	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.enableforeignkey` s is set to false.
`curam.db.disableInvalid LobFileError`	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.shortnames`	BOOLEAN	It is recommended strongly 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 versions of Curam, 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.

Table 45. Database settings (continued)		
Property Name	**Type**	**Meaning**
`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 (IBM Db2®). The database server TCP/IP port.
`curam.db.enable` `.bindings.generation`	BOOLEAN	Suggested: false. Causes a bindings file to be generated for the Java Database Connectivity (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. It is 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, for instance 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 for Db2 and Db2 for IBM z/OS. Default value is true.

Table 45. Database settings (continued)

Property Name	Type	Meaning
`curam.db.multibyte .default.factor`	FLOAT	Specifies the default expansion factor for multi-byte string fields if the multi-byte expansion feature is enabled. The value must be a float between the values of 1 and 4. Default value is 4.
`curam.db2.ssl`	BOOLEAN	Default: false. Indicates that Secure Sockets Layer (SSL) is to be used for Db2 database communications.
`curam.db2.ssl.truststore.locat ion`	STRING	Default: none. Specifies the SSL trust store location to be used for secure Db2 database communications.
`curam.db2.ssl.truststore.passw ord`	STRING	Default: none. Specifies the SSL trust store password to be used for secure Db2 database communications.
`curam.db2.purescale`	BOOLEAN	Default: false. Indicates that the Db2 pureScale property, `enableSysplexWLB` , is set for the Db2 DataSource and IBM WebSphere Application Server configuration.

Back to properties category list

Environment

These settings configure the environment for your SPM application.

Table 46. Environment settings

Property Name	Type	Meaning
`curam.environment.as.vendor`	STRING	Suggested: Should be set to BEA or IBM to reflect the Application Server being used. If running outside an application server, this should be empty. This 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: 1221. Port on which the `tnameserv` is running.
`curam.environment. bindings.location`	STRING	Suggested: C:/Temp. Name of the file system location containing data sources.

Table 46. Environment settings (continued)

Property Name	Type	Meaning
`curam.environment. default.dateformat`	STRING	Default: yyyy MM dd. The date format. Can be set to one of: "d M Ayyubid," "M d Ayyubid," "yyyy M d," "dd MM Ayyubid," "MM dd Ayyubid," "yyyy MM dd," "d MMM Ayyubid," "MMM d Ayyubid," "yyyy MMM d," "d MMM Ayyubid," "MMMM d Ayyubid," "yyyy MMMM d," "dd MMM Ayyubid," "MMM dd Ayyubid," or "yyyy MMM dd."
`curam.environment. default.dateseparator`	STRING	The date separator. Can be set to one of: ".", ",", "/", "-".
`curam.environment.default.time format`	STRING	Valid time formats are hh mm ss a, hh mm a, HH mm, and HH mm ss. Default is HH mm ss
`curam.environment.default.time separator`	STRING	Valid time separator formats are : and .. Default is :
`curam.disable. dynamic.properties`	BOOLEAN	This indicates if dynamic properties should be enabled or disabled. This is used by command line tools that require access to properties but cannot access the database.
`curam.deprecation.reporting`	BOOLEAN	This 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 (for example, rules and workflow validation).
`curam.entity.struct.deprecatio n`	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.

Back to properties category list

Test

These settings configure elements of SPM that are useful for unit testing.

Important: Do not use these settings in a deployed application as they can degrade performance or cause failures.

Table 47. Test settings		
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 only to specify the date. If the time portion of the property is not set explicitly the time will be default automatically to midnight (00:00:00). For example, the string value '20070101T175930' represents 17:59:30 on Jan. 1, 2007. The string value '20070101' represents 00:00:00 on Jan. 1, 2007.
`curam.test. treatreadmultimaxaserror`	BOOLEAN	Default: false. Specifies that a run time error should be thrown in addition to 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

Back to properties category list

Custom

These settings allow a developer to replace elements of the SPM infrastructure with their own customized handlers.

Table 48. Custom settings		
Property Name	**Type**	**Meaning**
`curam.custom. workflow.webservicebpo`	STRING	The name of the application Business process Objecs (BPO) that workflow process enactment web services go through.

Back to properties category list

Dynamic properties in Application.prx

The following properties relate to the available dynamic properties in the `Application.prx` file.

Environment

These settings configure the environment for your IBM Cúram Social Program Management (SPM) application.

Table 49. Environment settings

Property Name	Type	Meaning
`curam.environment.default.locale`	STRING	Default: en. The default value of the language code for the server.
`curam.environment.recordlocked.systemexception`	BOOLEAN	Specifies whether a `RecordLockedException` is set to a System exception. The default is false here, that it is an Application exception.
`curam.environment.readmultimax.systemexception`	BOOLEAN	Specifies whether a configparam is set to a System exception. The default is false here, that it is an Application exception.
`curam.transaction.sqlquerycache.disabled`	BOOLEAN	Specifies whether any SQL queries that do a SELECT on a database table have their results that are cached for the duration of the transaction in which the operation was invoked. Subsequent calls that use the same SQL query retrieve the results from this thread local transaction SQL query cache and not read the results from the database. The default setting for disabling this cache is false so that the results of SQL queries are cached.
`curam.sqlquerycache.lob.max.size`	INT64	Specifies the maximum size of a field of type Character Long Object (CLOB) or type Binary Large Object Block BLOB) in a result set that is allowed to be cached in the transaction SQL query cache.
`curam.enable.logging.client.authcheck`	BOOLEAN	Default: false. When set to true, all client authorization checks will be logged to the `AuthorisationLog` database table.

Table 49. Environment settings (continued)		
Property Name	**Type**	**Meaning**
`curam.audit. audittrail. datacompressionthreshold`	INT32	Specifies the size of the audit data stored in the `detailinfo` column of the `audittrail` database table that causes data compression to be invoked. Default: -1 (off). This value is checked per audit operation. To turn compression on for all `audittrail detailinfo` data set this value to 0. When turned on rows that contain compressed data have the boolean attribute ISCOMPRESSED set to true. Note that short audit data is not likely to see performance gains, but will for large data rows. The performance of Curam auditing Out Of The Box (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 by way of the `curam.util.resources`. `ByteArrayUtil.byteArrayToBase64 EncodedString` method and decompression can be done by way of the corresponding `ByteArrayUtil.base64 EncodedStringToByteArray` method.

JMX

These settings configure the Java Management Extensions (JMX) infrastructure for your SPM application.

Table 50. JMX settings		
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 also.
`curam.jmx.transaction_tracing _url_filter`	STRING	Regular expression to identify URLs for which transaction tracing data is collected.

Table 50. JMX settings (continued)		
Property Name	**Type**	**Meaning**
`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 be preserved only for this number of threads.
`curam.jmx.transaction _tracing_purge_period`	INT32	The period of time, in seconds, between checks to ensure that 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.pe r_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.

Table 50. JMX settings (continued)		
Property Name	**Type**	**Meaning**
`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.

Test

These settings configure elements of SPM that are useful for unit testing.

Important: Do not use these settings in a deployed application as they can degrade performance or cause failures.

Table 51. Test settings		
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 that has 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.stubdeferred processinsametransaction`	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.

Rules

These settings configure the rules infrastructure of Curam.

Table 52. Rules settings

Property Name	Type	Meaning
`curam.rules.file.access.location`	STRING	The directory where the XML representation of rule sets will be created. <Cannot be used for Cúram express rules (CER) rules>
`curam.rules.file.access.multilocation`	BOOLEAN	Specifies that rule set files exist in multiple locations. <Cannot be used for CER rules>
`curam.rules.model.file.rdo.access`	BOOLEAN	Specifies that Remote Data Objects (RDOs)should be retrieved from a Curam model file. <Cannot be used for CER rules>
`curam.rules.default.locale`	STRING	Default: `en_US` . Default locale used when creating the XML representation of rule sets. <Cannot be used for CER rules>
`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` <Cannot be used for CER rules>
`curam.rules.enable.optimization`	BOOLEAN	Specifies the rules optimization. <Cannot be used for CER rules>
`curam.rules.enable.fulltext`	BOOLEAN	Specifies the rules engine construction of full result text. <Cannot be used for CER rules>
`curam.debug.rules`	BOOLEAN	Default: false. Specify whether the rules debugging should be enabled. <Cannot be used for CER rules>
`curam.disable.empty.objectivelistgroups`	BOOLEAN	Default: true. Specify whether the rules decision should include empty Objective list groups.

Table 52. Rules settings (continued)		
Property Name	**Type**	**Meaning**
`curam.rules.date.range. includes.calculation.date`	BOOLEAN	Specifies the new objective calculation. <Cannot be used for CER rules>

IEG

These settings configure the properties for the Intelligent Evidence Gathering (IEG) environment.

Table 53. IEG settings		
Property Name	**Type**	**Meaning**
`curam.iegeditor. callback.class`	STRING	Specifies the IEG Editor Application Callbacks class.
`curam.iegruntime. questionpage. separatequestionsforloopstyle`	BOOLEAN	Specifies whether to use separate question pages when "for" looping.
`curam.ieg.answers. mustmatch. currentpagequestions.disabled`	BOOLEAN	Specifies whether to disable the default behavior for testing purposes. By default, when the IEG REST API for submitting answers is called, it compares any submitted question-answer pair against the questions that are defined on the current question page. Answers from questions that are not on the current question page are not processed. A new server logging message displays when `curam.trace` is set to `trace_on` and logs when an answer is submitted by a user to a question that is not available from the current question page. When set to `true`, this behavior is disabled.

Custom

Developers can use these settings to replace elements of the SPM infrastructure with their own customized handlers.

Table 54. Custom settings		
Property Name	**Type**	**Meaning**
`curam.custom. deferredprocessing.dpcallbac` k	STRING	The name of the application class that implements the DPTicketCallback interface.
`curam.custom. workflow.workresolver`	STRING	The name of the application class that implements the WorkResolver interface.

Table 54. Custom settings (continued)		
Property Name	**Type**	**Meaning**
`curam.custom.` `workflow.processcachesize`	INT32	Default: 250. Specifies the maximum size of the process definition cache.
`curam.audit.` `audittrail.noxmlaudit`	BOOLEAN	If set to true this property will disable the existing audit writer.

`curam.custom.`
`notifications.notification`
`delivery`
	STRING	Specifies the name of the application class that implements the NotificationDelivery interface.

`curam.custom.`
`dataaccess.database`
`writecallback`
	STRING	The name of the application class that implements the DatabaseWriteCallback interface.

`curam.custom.` `dataaccess.transactioncallback`	STRING	The name of the application class that implements the TransactionCallback interface.
`curam.custom.` `disable.database.callback`	BOOLEAN	If set to true this property will disable the database callback.

Trace

These control which extra diagnostic information, in addition to errors that are always logged, is written to the application server's diagnostics file. You can set the "curam.trace.*" settings independently of the "curam.trace" settings, resulting in the union of these settings.

Property Name	Type	Meaning
Table 55. Trace 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.

Table 55. Trace settings (continued)		
Property Name	**Type**	**Meaning**

`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. <For classic rules only>
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 that 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. invocation_differentiators	STRING	Comma separated list of invocation 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 55. Trace settings (continued)		
Property Name	**Type**	**Meaning**
curam.trace.deferred.user.name	BOOLEAN	A Boolean flag that indicates which user name will be available for logging purposes for transactions of type Deferred. Either the Deferred User Name (the user that initiates the deferred process) or the name of the currently logged in user for that transaction is made available depending for logging on the value of this property. *true* When set to true, then the name of the user who initiated the deferred process will be available to be added to the logs. *false* When set to false, then the user associated with the current transaction will be available to be added into the logs. By default, the property is set to *true*.

Security

These settings configure the SPM authentication behavior.

Table 56. Security settings		
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.

SMTP

These settings configure the environment for the Simple Mail Transport Protocol (SMTP) client element of SPM.

Table 57. SMTP settings

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.

XML Server

These settings configure the environment for the XML Server.

Table 58. XML Server settings

Property Name	Type	Meaning
`curam.xmlserver.host`	STRING	The host on which the XML Print Server resides. The property also may be specified as a slash (/) separated list of host names in order to use multiple XML Servers. For further information, 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 slash (/) separated list of ports in order to use multiple XML Servers. For further information, 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 XML Server.
`curam.xmlserver.serializelocaleneutral`	BOOLEAN	Specify that XML Server data will be serialized in a locale-neutral way instead of being based on the locale properties on the server.

Database

These settings configure SPM for database communication.

Table 59. Database settings		
Property Name	**Type**	**Meaning**
`curam.db.readmultimax`	INT32	Default: 100. This allows the developer to override the default maximum number of records returned by the readmulti (readmulti, nsreadmulti, multithread, and nsmulti) operations in an application. This default value is only used if an explicit value is not set in the model. Unless the `Readmulti_Informational` option is set in the model there is no enforcement of this limit.
`curam.db.locktimeout`	INT32	Default: 30. This 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 only is 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.

KeyServer

These settings are for customers to configure the behavior of the KeyServer.

Table 60. KeyServer settings		
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 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.

Table 60. KeyServer settings (continued)		
Property Name	**Type**	**Meaning**
`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, for instance, 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.

BatchLauncher

These settings configure the behavior for when problems occur calling batch programs.

Table 61. BatchLauncher settings		
Property Name	**Type**	**Meaning**
`curam.batchlauncher .erroremail.recipient`	STRING	The email address of the recipient of error emails from Curam.
`curam.batchlauncher. erroremail.nostacktrace`	BOOLEAN	Default: false. Suppress the stack 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 workflow functionality for batch programs should be enabled. When set to true, the
		
		`curam.batchlauncher.dbtojms. notification.host`
		
		and
		
		`curam.batchlauncher. dbtojms.notification.port`
		
		properties also must be set.

Property Name	Type	Meaning

`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.ssl. protocol`	String	Default: SSL. The protocol name appropriate and valid for your environment, which is dependent on your JDK and application server; e.g.: SSL, TLS, etc. For this property to be used `curam.batchlauncher.dbtojms .notification.ssl` must be set affirmatively.
`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.

Table 61. BatchLauncher settings (continued)

Table 61. BatchLauncher settings (continued)		
Property Name	**Type**	**Meaning**

`curam.batchlauncher. dbtojms.notification.test. stubtrigger`	BOOLEAN	Default: false. For debugging batch jobs which use DBtoJMS: stubs out
		
		`DBtoJMS.beginTransfer()`
		
		to prevent it from creating deferred processes.
		
		`JMSLiteEngine`
		
		must be started to process the messages.

`curam.batchlauncher. dbtojms.messagesper transaction`	INT32	Default: 512. The number of messages per transaction processed by the database-to-JMS conversion.

Workflow

These settings configure the properties which relate to the Workflow Environment.

Table 62. Workflow settings		
Property Name	**Type**	**Meaning**
`curam.workflow.disable.audit. wdovalueshistory.before. 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.audit .wdovalueshistory.after.activi ty`	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.audit. wdovalueshistory.transition. 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.

CTM

These settings configure the properties that relate to the Configuration Transport Manager (CTM).

Table 63. CTM settings		
Property Name	**Type**	**Meaning**
`curam.ctm.landscape.name`	STRING	Default: `nolandscape` . The landscape name for CTM to transport change set from source to target systems with in the configured landscape.

Static properties in Application.prx

The following properties relate to the available static properties in the `Application.prx` file.

Custom

These settings allow a developer to replace elements of the IBM Cúram Social Program Management infrastructure with their own customized handlers.

Table 64. Custom settings		
Property Name	**Type**	**Meaning**

`curam.custom.audit.writer`	STRING	Default:
		
		`curam.util.internal.misc.`
		`StandardDatabaseAudit`
		
		.
		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.

Security

These settings configure the authentication behavior of Curam.

Table 65. Security settings		
Property Name	**Type**	**Meaning**
`curam.security.` `disable.authorisation`	STRING	Default: false. Suppress the authorization checks normally performed by Curam.

Table 65. Security settings (continued)		
Property Name	**Type**	**Meaning**
`curam.security.` `casesensitive`	BOOLEAN	Authentication and authorization of user names 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 (for instance, caseworker, CaseWorker), authentication will fail due to an ambiguous user name. Such duplicate names also will 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, for example, INTERNAL or EXTERNAL.

Trace

These control what diagnostic information (in addition to errors which are always logged) is written to the application server's diagnostics file.

Table 66. Trace settings		
Property Name	**Type**	**Meaning**

`curam.trace.method_handler`	STRING	Default:
		
		`curam.util.resources.Trace.Curam`
		`MethodInvocationHandlerDefault`
		
		.
		Name of a class implementing
		
		`curam.util.resources.Trace.`
		`CuramMethodInvocationHandler`
		
		to perform custom method tracing.

`curam.trace.` `dataaccess.maxstringlength`	STRING	Default: 1000. Maximum length of a String or CLOB logged by the Data Access Layer when SQL tracing is enabled.

Environment

These settings configure the environment for your SPM application.

Table 67. Environment settings		
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. This indicates if tab caching should be disabled. **Note**: this only applies to caching on the server side.

KeyServer

Set the property `curam.keyserver.keyset.cachesize` to control the number of unique ID key sets that are consumed and cached per KeyServer transaction to optimize your batch program execution.

The KeyServer is a mechanism that generates unique IDs for the `curam.util.type.UniqueID`. Use the KeyServer to consume key sets in batch rather than individually. A key set contains 256 IDs. By default, for every 256 unique IDs generated the KeyServer consumes another key set from the database. The process involves the KeyServer reading and updating a record in the KeyServer table. When a larger volume of unique IDs must be processed, it is more efficient to consume more than one key set per update to the KeyServer table.

When you set the property `curam.keyserver.keyset.cachesize` to a value in the range 1 - 64 in either the `Bootstrap.properties` or from the batch launcher command line, you can control the number of unique ID key sets that are consumed and cached per KeyServer transaction.

Table 68. KeyServer settings		
Property name	**Type**	**Meaning**
`curam.keyserver.keyset.` `cachesize`	INT32	Default: 1: Specifies the number of unique ID key sets to be consumed and cached per KeyServer transaction.

Example 1: To specify a key server cache size of 20 for all batch programs

To specify a key server cache size of 20 for all batch programs, add the entry `curam.keyserver.keyset.cachesize=20` to `Bootstrap.properties`.

Example 2: To specify a key server cache size of 20 for a single run of the batch launcher

To specify a key server cache size of 20 for a single run of the batch launcher, use the command line:

```
build runbatch -Djava.extra.jvmargs="-Dcuram.keyserver.keyset.cachesize=20"
```

Example 3: To specify a key server cache size of 20 for a single run of the batch launcher for a single batch program named `CloseCasesPendingClosure.closeCasesPendingClosure`

To specify a key server cache size of 20 for a single run of the batch launcher for a single batch program named `CloseCasesPendingClosure.closeCasesPendingClosure`, use the command line:

```
build runbatch
-Dbatch.program=curam.core.intf.CloseCasesPendingClosure.closeCasesPendingClosure
-Djava.extra.jvmargs="-Dcuram.keyserver.keyset.cachesize=20"
```

.

Note: The following property affects batch programs only:

```
build runbatch
-Dbatch.program=curam.core.intf.CloseCasesPendingClosure.closeCasesPendingClosure
-Djava.extra.jvmargs="-Dcuram.keyserver.keyset.cachesize=20"
```

The property does not affect online, workflow, or deferred processing transactions. Likewise, the property does not affect range aware key sets.

The property can be specified in `Bootstrap.properties` only or on the command line.

If the property is specified in the properties table, the property has no effect.

For more information about unique IDs and the KeyServer, see the *Unique IDs* related link.

Related concepts
Unique IDs
Use this information to understand what Unique IDs are in the context of Cúram and how to use them in your application.

Variable property settings

The following properties whose name is defined variably.

Transaction

Use this information to understand the properties connected with the runtime setting of transactional options.

This table contains properties connected with the runtime setting of transactional options.

Table 69. Transaction settings		
Property Name	**Type**	**Meaning**
`<fully qualified code package>.intf.<class name>.<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 times out. Format: `PROJECTNAME.CODEPACKAGE.intf.CLASSNAME.OPERATIONNAME.transaction.timeout;` for example, `curam.core.facade.intf.Person.createAddress.transaction.timeout=60`
`LoginBeanTransaction.transaction.timeout`	INT32	Used to control the transaction timeout for the user login operation. The value is the number of seconds before the user login transaction times out. If this property is not specified, the login transaction timeout defaults to the JTA timeout value that is for the application server.

Audit

Use this information to understand properties that are connected with the editing options with runtime setting.

Contains properties connected with the runtime setting of auditing options.

Table 70. Audit settings		
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.

Table 70. Audit settings (continued)		
Property Name	**Type**	**Meaning**
`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 that implements `curam.util.audit.DataAccessHook` and that is used to audit client-visible operation calls.
`curam.custom.predataaccess.hook`	STRING	Specify the name of the class that implements `curam.util.audi.DataAccessHook` and is used to audit data access calls.
`curam.custom.audit.writer`	STRING	Specify the name of a class that implements `curam.util.audit.AuditLogInterface` and is 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 property saves XML from being generated for each invocation of the operation done so far. Default: false.

Infrastructure auditing settings

Use this information to understand the database operations that are available in Cúram and the default value of their table-level auditing flag.

Default table-level audit setting

Information that is listed in the tables on this page show the operations names and their default audit settings for database operations in the Cúram application. Use this information to learn the operation names and understand the default settings for each operation.

The tables that follow list the database operations in the IBM Cúram Social Program Management infrastructure and the default value of their table-level auditing flag. This value might be overridden by setting application properties. For more information, see the *Cúram Modeling Reference Guide*. Certain database operations do not support auditing; for example, operations with stereotype ns with handcrafted SQL. These settings are listed with a default value of N/A

Table 71. Audit settings 1	
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
`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

Table 71. Audit settings 1 (continued)	
Operation Name	**Default Auditing Setting**
`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
`BPOMethodLibrary.searchByCompoundKey`	False
`BatchErrorCodes.getAllErrorCodes`	N/A

Table 71. Audit settings 1 (continued)	
Operation Name	**Default Auditing Setting**
`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

Table 71. Audit settings 1 (continued)	
Operation Name	**Default Auditing Setting**
`BatchProcDesc.insert`	True
`BatchProcDesc.modify`	True
`BatchProcDesc.read`	False
`BatchProcDesc.readAll`	False
`BatchProcDesc.remove`	True
`BatchProcGrpAssoc.insert`	True
`BatchProcGrpAssoc.readmulti`	False
`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

Table 71. Audit settings 1 (continued)	
Operation Name	**Default Auditing Setting**
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
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

Table 71. Audit settings 1 (continued)	
Operation Name	**Default Auditing Setting**
`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

Table 71. Audit settings 1 (continued)	
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

Table 71. Audit settings 1 (continued)	
Operation Name	**Default Auditing Setting**
`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
`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

Table 71. Audit settings 1 (continued)	
Operation Name	**Default Auditing Setting**
`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
`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

Table 71. Audit settings 1 (continued)	
Operation Name	**Default Auditing Setting**
`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

Table 71. Audit settings 1 (continued)	
Operation Name	**Default Auditing Setting**
KeySetRange.read	False
MatchedEvtArchive.getMatchedEventsForActivityInstance	False
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

Table 71. Audit settings 1 (continued)	
Operation Name	**Default Auditing Setting**
`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 72. Audit settings 2	
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

Table 72. Audit settings 2 (continued)	
Operation Name	**Default Auditing Setting**
`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
`ProcessInstance.remove`	False
`ProcessInstance.searchByBizObject`	N/A
`ProcessInstance.searchByEventWaitDetails`	N/A

Table 72. Audit settings 2 (continued)	
Operation Name	**Default Auditing Setting**
`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

Table 72. Audit settings 2 (continued)	
Operation Name	**Default Auditing Setting**
`Reminders.insertReminder`	False
`Reminders.scanReminders`	N/A
`RuleSetInformation.insert`	False
`RuleSetInformation.listByType`	False
`RuleSetInformation.modify`	False
`RuleSetInformation.read`	False
`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

Table 72. Audit settings 2 (continued)	
Operation Name	**Default Auditing Setting**
`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
`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

Table 72. Audit settings 2 (continued)	
Operation Name	**Default Auditing Setting**
`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
`SecurityRoleGroup.readRolesInGroup`	False

Table 72. Audit settings 2 (continued)	
Operation Name	**Default Auditing Setting**
`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

Table 72. Audit settings 2 (continued)	
Operation Name	**Default Auditing Setting**
`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
`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

Table 72. Audit settings 2 (continued)	
Operation Name	**Default Auditing Setting**
`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
`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

Table 72. Audit settings 2 (continued)	
Operation Name	**Default Auditing Setting**
`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

Table 72. Audit settings 2 (continued)	
Operation Name	**Default Auditing Setting**
`WDOTemplateLibrary.read`	False
`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

Table 72. Audit settings 2 (continued)	
Operation Name	**Default Auditing Setting**
`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
`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

Table 72. Audit settings 2 (continued)	
Operation Name	**Default Auditing Setting**
XSLTemplateInst.read	False
XSLTemplateInst.remove	False

Notices

This information was developed for products and services offered in the United States.

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, MD-NC119 Armonk, NY 10504-1785 US

For license inquiries regarding double-byte character set (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. 19-21, Nihonbashi-Hakozakicho, Chuo-ku Tokyo 103-8510, Japan

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 jurisdictions 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 typographical 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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide 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 Director of Licensing IBM Corporation North Castle Drive, MD-NC119 Armonk, NY 10504-1785 US

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.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

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-IBMproducts. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

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

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

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

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

COPYRIGHT LICENSE:

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

Privacy Policy considerations

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

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

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

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

Trademarks

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/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

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

The registered trademark Linux is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft, Windows, Windows NT, and the Windows logo 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.

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

IBM®

Part Number:

(1P) P/N: