Cúram 8.2

Development Compliancy Guide

Note

Before using this information and the product it supports, read the information in Notices on page 25

Edition

This edition applies to Cúram 8.2.

Contents

Chapter 1 Compliant development with Cúram

When you develop Cúram applications, you must comply with certain guidelines to ensure that you can easily upgrade to future versions without affecting your custom functionality. Complying with these guidelines is essential to ensure that the support team can better support your custom implementation.

1.1 Starting a new project

When you start a new project, it is important to review the development directory structure and place the appropriate files under source code control so that you can track all changes.

When the files are under source code control, tag all development artifacts. Ensure that the tag reflects to the version of the application. At any point, you can then produce a report to identify all files that were added or changed by using file comparison tools. This report is useful when you are upgrading the application.

Review the client and server development directory structure

Review the development directory structure to understand where development artifacts are located, how they are organized, and where to store changes to these artifacts.

The client and server development artifacts are installed in the following directories:

- Client development artifacts are installed into the *`webclient`* directory.
- Server development artifacts are installed into the *`EJBServer`* directory.

Within both the *`webclient`* directory and the *`EJBServer`* directory, there is a *`components`* subdirectory, which has a further subdirectory called *`custom`*. The *`custom`* subdirectory is where all project-specific development artifacts should be placed. The other *`components`* subdirectories contain all of the application development artifacts that are delivered with the product.

> **Important:** The *`custom`* folder contains a starter structure for first usage and is referred to throughout developer documentation as the area in which all artifacts are developed. This convention is not mandatory and it is a project choice to develop within this component or create a new named component appropriate for your project.

Within the *`EJBServer\components\custom\model`* directory, there is a starter model file and some model fragments.

Related information

1.2 Compliancy for server development

Learn how to avoid common server compliancy issues, and how to develop server applications in a compliant manner.

Avoiding common server compliancy issues

Follow the guidelines to avoid these common compliance issues. Following these guidelines from the early stages of a project is relatively easy. However, if you do not, it can result in serious disruptions later and fixing these disruptions can be both costly and difficult.

Use project-specific prefixes in custom artifact names

Avoid naming collisions when you upgrade by ensuring that you always name new, custom artifacts with a consistent prefix for your project. Naming collisions can be difficult to fix afterward. Prefix all new source artifact names with a relevant acronym or abbreviated word to prevent naming collisions from occurring between your custom artifacts and artifacts that Merative ™ might add over time.

Use the same acronym or abbreviated word throughout. As the project progresses, this prefix makes project additions to core artifacts more obvious. This distinction becomes more useful as the development effort grows. Most projects are described by some kind of acronym and this acronym is a good candidate to use as the prefix.

Project-specific prefixes might not apply when you override some application artifacts. Where supported, override mechanisms typically require the custom artifacts to have the same name as the default artifacts that they override, but some exceptions exist.

Some further considerations are as follows:

- There are many different types of identifiers. For example, a file name, an XML ID, a Java class name, or a combination of identifiers.
- A short prefix is advisable because there might be restrictions on name lengths. For example, some types of database identifiers have length restrictions.

Note: In addition to source artifacts, it is also important to consider identifier values that might conflict with values that are used by Merative ™.

Some artifact types have more than one identifier. Remember this when you name your custom artifacts. The following list describes examples of common development artifacts that can cause naming collisions when you take on a new release.

- **Database fields**
  New database fields can be delivered in fix packs. Use project prefixes for database fields to prevent duplicating the names delivered in the fix packs.

- **Application code table items**
  New application code table items can be delivered in fix packs. Use a project prefix when you name custom code table items to prevent duplicating the names delivered in the fix packs. Custom code table items have a value and a Java identifier, and both share a flat namespace with application items in the same code table.
- **Entity classes**

  Custom Entity classes have a table name that shares the flat namespace and database schema with application tables and must have a unique table name within that namespace. It also has a Java class name, which shares a hierarchical namespace and package structure with application Java classes. Use project-specific prefixes for custom entity classes to prevent duplication of the names of the new entity classes.
- **Identifier values that conflict with values used by Merative ™**
  Consider identifier values that might conflict with values used by Merative ™.
  For example, the `TransactionInfo.setFacadeScopeObject` and `TransactionInfo.getFacadeScopeObject` APIs enable developers to access objects that are associated with the current transaction. When you use these APIs, use a String as your object identifier and prefix this string with an appropriate project-specific word to ensure that your data for the transaction does not conflict with Merative ™ data.

Use numeric identifiers in custom initial and demo data

Pre-defined initial data and demo data is loaded into an application database using DMX files. This data is installed into the database when a system is first set up, or when a system is upgraded. You might also want or need to add your own initial data or demo data.

Reserved ranges for unique identifiers for primary keys

To avoid clashes with the initial and demo data that is included in the application and with data created by the runtime system, it is important that the identifiers (for example, primary keys) for your initial and demo data are drawn from reserved ranges. A set of ranges is reserved for customer use.

- Non-human readable primary keys:
  - 45,000 - 49,999 (inclusive)
  - 900,000 - 949,999 (inclusive)
- Human readable primary keys: 11,521 - 12,799 (inclusive)
- Rule sets: 4,500 - 4,999 (inclusive)

Large data sets

Instead of using keys from the allocated ranges, use the key server to generate the key values required. If this data is to be imported into a pre-built database, extract the final value of the key set and load it into the key set table, replacing the initial key set value supplied in the application. If you have any questions about this process, contact Cúram Support for further information.

To request a product enhancement, see the Merative ™ Ideas Portal .

Avoid directly modifying application files in place

Continuous delivery, Fix Pack, and iFix releases must be able to safely move, restructure, or overwrite application files. If the included application files are modified, upgrades might overwrite them without notice and the changes might not be compatible with the modifications. Reapplying the in-place changes afterward might not be possible.

Client and Server: Exceptions for in-place modifications

A list of the small number of exceptions to the in-place modifications rule for client and server development.

- EJBServer
  - *`/project/config/datamanager_config.xml`*
  - *`/project/config/deployment_packaging.xml`*
  - *`/project/properties/Bootstrap.properties`*
  - *`.classpath`*
  - *`.project`*
- Webclient
  - *`/JavaSource/curam/omega3/ApplicationConfiguration.properties`*
  - *`/JavaSource/curam/omega3/i18n/CDEJResources.properties`*
  - *`.classpath`*
  - *`.project`*

Never create dependencies on sample or demo artifacts

Never create dependencies on sample or demo artifacts. Never rely on dependencies or references to sample or demo artifacts from custom code. Sample or demo artifacts are subject to change without notice

Different product areas in Cúram take different approaches to marking artifacts as Internal, Sample, or Demo, so this information cannot give a concise statement of how to identify them. However, there are a few instances where they can be identified. These instances are artifacts whose name, code package, model package, or file path contain the words Internal, Sample, or Demo, or obvious derivatives of those words. If in doubt, contact Cúram Support.

To request a product enhancement, see the Merative ™ Ideas Portal .

Related information

The CPMSample folder

The `CPMSample` folder is internal; all code and artifacts within this folder can change without any notice. If customers want to use functionality in the *`CPMSample`* folder, they must duplicate it in their code base.

Apply changes to dynamic artifact types back to the development system

If you modify dynamic artifact types on production or test systems, always ensure that these modifications are applied to the development system.

Various 'Dynamic' development artifacts exist in the application that can be modified at runtime on a production or test system (for example, code tables and workflows). Runtime changes to these artifacts should always be synchronized back to the development codebase so that concurrent development changes can be integrated with these runtime changes prior to deployment.

Concurrent changes to these artifacts may happen during routine project milestone development, or when taking on Fix Packs or other upgrades. In every case, there must be one central place where concurrent changes are merged and validated and this is the development codebase. The system of record for these artifacts is the development codebase.

Overview of compliant server development artifact changes

In addition to your custom code, you can customize the default application by adding message files, code tables, events, and so on.

The following table summarizes the range of compliant changes you can make to development artifacts.

Table 1: Cúram Development Artifact Compliant Changes

Type of Change	Compliant Changes
Change or remove an existing message file or add additional locale (language) support to an existing message.	Message file (externalized server informational, warning, and error messages - `.xml` files in the *`message`* directory). For more information,.
Change an existing code table display name or description, add a code table item into an existing code table, or enable or disable an existing code table item.	Code Table file (code value pairs - `.ctx` files in the *`codetable`* directory). For more information, .
Add an event registration (to augment initial Cúram functionality, or disable an existing event handler.	Event Definition file (`.evx` files in the *`events`* directory) and Event Handler Registration file (`handler_config.xml` in the *`events`* directory). For more information, .
Override an existing user preference.	User Preference file (`DefaultPreferences.xml` file in the *`userpreferences`* directory).
Customizing workflow process definition files.	For more information, .

Type of Change	Compliant Changes
Override an existing application property.	You cannot override an application property directly. For information about how to customize properties, . Application Property File (*`Application.prx`* file in the *`properties`* directory).
Add initial demo or test data (rows) to an existing database table.	DMX File (script for populating the database with data - *`.dmx`* files in the relevant *`data`* subdirectory).

Java APIs

Java class operations are marked as Internal, Restricted, or External by annotations. By default, classes with no annotations are internal. External operations are the official Java API, which you are encouraged to use and call from your own code.

Internal APIs

Internal APIs are annotated with `@Accesslevel(INTERNAL)` or have no annotation. Do not reference internal APIs in custom code. Restricted APIs are annotated with `@Accesslevel(RESTRICTED)`, never reference a restricted API in custom code.

If you reference restricted or internal APIs in code, restricted APIs produce Eclipse errors and unsupported APIs produce Eclipse warnings.

External APIs

External APIs are annotated with `@Accesslevel(EXTERNAL)` and you can reference them directly from custom code. Javadoc is provided for all external APIs on a per-component basis. Do not reference any classes that do not have Javadoc.

The Javadoc for each component is in *`components\<component name>\doc\api.zip`*. Some components might not have external APIs so have no Javadoc.

External APIs can evolve over time, while remaining compatible with previous versions. If you need some capability that you cannot fulfill through a combination of external APIs and allowed extension mechanisms, contact Cúram Support. If appropriate, a new API, customization hook, strategy pattern or configuration-based approach might be made available. In some circumstances an internal API might be re-designated as external.

Source code

All application Java functionality is distributed as JAR files. If required by the use of customer extension mechanisms, you can regenerate and rebuild applications in a customer installation.

The customer build process does not need to rebuild the entire Java source code base; only project-specific source code and any dependent regenerated Java source code needs to be rebuilt.

Java source code is not delivered for a limited number of key functional areas. Source code for the remainder of the application is included as sample code for documentation purposes only and is not directly involved in the build process. This sample source code is distributed in JAR files on a per-component basis as follows: *`components\<component name>\sample\src.zip`* The built versions of each component can be found in the following location: *`components\<component name>\lib\<component name>.jar`*

Changing server source artifacts

There are many types of server artifacts, including application classes. Some of these artifacts are represented in an application model. Other Java interfaces are "handcrafted". While it is possible to change limited aspects of a modeled interface by changing the model and regenerating code, it is not possible to change a handcrafted interface.

It is important to be able to distinguish between the application implementations of both categories of class.

- **Modeled interfaces**
  Appear in the application UML model
- **Handcrafted interfaces**
  - Do not appear in the application UML model
  - Appear in the component directories of your development environment
  - Cannot be customized
  - Contain the `@ImplementedBy` Google Guice annotation to indicate the application implementation class

Some components can contain interfaces that do not fall into either of these categories, and these interfaces are described in component-specific documentation. Both modeled and handcrafted application interfaces can have implementations that can be customized. You must look at an implemented interface to determine its category.

Related information

Source code for new methods and classes

Create new source files for all new code, including classes that wrap existing classes. Put all new source files in the *`source`* subdirectory of the *`EJBServer\components\custom`* directory.

For modeled classes, the generated class hierarchy dictates the package structure of the new source files.

For handcrafted implementations, you can choose how to package the new class. You can use Google Guice to configure new subclasses.

Related information

Changing CER rule sets

The CER Editor stores its rule sets on the database rather than in the file system. Do not customize rule sets that are included in the core component.

For more information about rule sets, see the *CER Rule Sets Included with the Application* related link.

Related information

Extending code tables

Some code tables are safe to extend and some are restricted. If you want to customize a restricted code table you must request a product enhancement.

You can use the Cúram Analysis Documentation Tooling (CADT) to help you to identify restricted code tables. The Cúram Analysis Documentation Tooling (CADT) is available for download from Cúram Support.

A list of restricted code tables is provided in the project documentation folder structure for every installation, in a folder called `RestrictedCodeTables`. You must not customize these code tables without specific guidance from Cúram Support.

To request a product enhancement, see the Merative ™ Ideas Portal .

For more information about code tables, see the *Code tables* and the *Configuring Code Tables* related links.

Related concepts

Use project-specific prefixes in custom artifact names on page 10
Avoid naming collisions when you upgrade by ensuring that you always name new, custom artifacts with a consistent prefix for your project. Naming collisions can be difficult to fix afterward. Prefix all new source artifact names with a relevant acronym or abbreviated word to prevent naming collisions from occurring between your custom artifacts and artifacts that Merative ™ might add over time.

Related information

Server extension mechanisms

While the default Cúram server application includes some sample source code, customers do not have the source code for most other areas of key functionality, and in addition a large number of APIs are marked as Internal. However, you can apply certain customization or alter existing application behavior according to the permitted extension practices for customer projects.

These extension mechanisms apply to extending or altering default server application artifacts only. With customer-defined classes, you can use all extension mechanisms, such as subclass-with-replace, and all the artifacts can be external in nature, and invoked from any other part of a customer implementation.

Summary Guidance

Summary guidance for referencing or customizing application classes.

Where you want to reference an application class in your custom code:

- If the class is External, you are allowed to reference it.
- If the class is Internal, you do not reference it in your code.
- If the class is Access Restricted, you are not supported in referencing it.

Where you want to customize an application class:

- If the class is modeled, follow the detailed guidance for allowed customization.
- If the class is non-modeled, refer to its Javadoc or any configuration or development guide for its parent component for details of customization points.

Entity classes

Direct customer use and modification of application Entity classes is not allowed. In many cases, application Entity class operations have direct Facade-layer equivalents, which are marked as External, and can be used by customers.

However, the addition of stereotyped and non-stereotyped operations to application Entities is allowed, as is the setting of a number of Entity options.

Customers that want to add data to application screens should add new customer-specific Entity classes, and wrap external application maintenance operations in their own process classes to maintain both tables atomically. Application screens can then be changed to point to the new process classes.

Table 2:

Action	Model Option	Extension class	Subclass With Replace	Subclass Without Replace	Comments
Add a stereotyped entity Operation (for example, `<<ns>>`, `<<nsreadmulti>>`)	N/A	No	No	Yes	Addition of new operations to an existing entity.
Add a non-stereotyped Entity operation	N/A	No	No	Yes	Addition of new operations to an existing entity.
Change an Entity operation option	Auto ID Field Auto ID Key No Generated SQL Optimistic Locking Order By SQL Where	No	No	No	
	Database Table-level Auditing	No	No	No	Use runtime properties to set this option.

Action	Model Option	Extension class	Subclass With Replace	Subclass Without Replace	Comments
	On Fail Operation Post Data Access Operation Pre Data Access Operation Treat Readmulti Max as Informational Exception Readmulti Max Records Returned	No	No	No	
Change an Entity class option	Enable Validation	No	No	No	
	Abstract Allow Optimistic Locking No Generated SQL	No	No	No	
	Audit Fields Last Updated Field	Yes	No	No	
Add an Entity attribute	N/A	No	No	No	
Change an Entity attribute option	Allow Nulls	No	No	No	

Struct classes

Application struct classes are all essentially external in nature, in that they can be referenced in customer-specific functionality.

Customers must not directly create aggregations from application structs to any other struct because they don't have full visibility on where these application structs are being used. However, customers can continue to use aggregation to include application structs in their own project-specific structs.

Table 3:

Action	Model Option	Extension class	Subclass With Replace	Subclass Without Replace	Comments
Add an attribute to a struct	N/A	No	No	No	Create a new project-specific struct, and aggregate the application struct from the project-specific struct to the application struct (not the other way around). Use the new 'composite' struct in required customer-specific functionality.
Change a struct attribute	N/A	No	No	No	Addition of new operations to an existing entity.
Change a struct option	Audit Fields	Yes	No	No	If you feel you have a valid need to change an attribute of an application struct, raise a Support case. Refer to the related link for a description of how to make an enhancement request.

Other modeled classes

For other modeled classes in the application, such as Process, Facade, and WSInbound, no extensions mechanisms are allowed.

Similar to Entity classes, customers should instead model and code their own Process, Facade or WSInbound classes, either wrapping existing external APIs, or implementing new functionality. For Facade operations, you can point affected UIM pages at the new Facade operations.

Domain definitions

In general, customer use and the overriding of application domain definitions is allowed. However, changing the fundamental type of a domain definition is not allowed, nor are some code table related options.

Table 4: Overriding Domain Definitions

Extension Action	Model Option	Allowed
Change a Domain Definition option	Code table Name Code table Root	No

Extension Action	Model Option	Allowed
	Compress Embedded Spaces Convert to Uppercase Custom Validation Function Name Default Maximum Value Minimum Size Minimum Value Pattern Match Remove Leading Spaces Remove Trailing Spaces Storage Type	Yes
	Maximum Size	Yes Allowed for increasing the size only. If you want to decrease the size of an application Domain Definition, raise a Support case. Refer to the related link for a description of how to make an enhancement request. Do not use to change the maximum size of the USERNAME Domain Definition.
Change the Type of a Domain Definition	N/A	No Create a Domain Definition with the appropriate Type, and wrap it in your own processing. Customers are not allowed to change the fundamental types of application Domain Definitions.
Create a Domain Definition based on an application Domain Definition	N/A	Yes

Non-modeled classes

Some components contain non-modeled classes. For these classes, the use of each External interface or class is described in the Javadoc information for the class.

Some non-modeled classes come with Eclipse access restrictions in place to provide customers with guidance in relation to which APIs they can and cannot call or customize. Certain classes and packages are marked as restricted; these classes must not be used as they are internal classes that can change over time. Access restrictions should not be removed from the `Eclipse.classpath` file because it might result in the consumption of restricted classes, which can cause problems during upgrades.

Some non-modeled components contain package protected classes; these classes should not be used in custom code. Customers must not place any custom code in the same package structure to call or reference package protected classes.

Many non-modeled APIs are not directly customizable. Only interfaces or classes that are tagged with the `@Implementable` annotation can be extended or implemented. Refer to Javadoc information detailing how to customize or implement such classes. Non-modeled classes that are not tagged with the `@Implementable` annotation must not be extended or implemented because new operations might be added over time, which might cause upgrade impact.

For classes tagged with the `@Implementable` annotation, the typical customization mechanisms for these types of class are events and strategies.

Events allow customers to add custom logic at various points in the application. For details on how to add event listeners, please refer to the *Developing with the Persistence Infrastructure* related link. Event classes are typically named 'xxxEvent', so they can be easily identified.

Strategy patterns allow customers to change the default behavior of certain functions within the application. Each strategy class has a default implementation provided; however customers can choose to override the default implementation of any of the strategy operations through the use of Guice bindings. Strategy classes are typically named 'xxxStrategy', so they can be easily identified.

Related information

Relationships

Extension mechanisms for relationships.

Table 5: Assignable Relationships

Action	Supported
Make a customer-supplied struct assignable to an application struct or entity.	Yes
Make an application struct that is assignable to another application struct or entity.	No

Table 6: Aggregation Relationships

Action	Supported
Aggregate an application struct in a customer-supplied struct. That is, create a customer struct that 'contains' an application struct.	Yes
Aggregate a customer-supplied or application struct in an application struct. That is, add any struct to an application struct by aggregation.	No

Table 7: Foreign Key Relationships

Action	Supported
Create a foreign key where a customer-supplied Entity is the child	Yes
Create a foreign key where an application Entity is the child	No

Table 8: Indexe Relationships

Action	Supported
Create an index on either an application or customer-supplied entity by using a customer-supplied struct.	Yes
Create an index on either an application or customer-supplied entity by using an application struct.	No

Table 9: Unique Index Relationships

Action	Supported
Create a unique index on an application entity.	No
Create a unique index on a customer-supplied entity by using an application struct.	No

1.3 Compliancy for client development

You customizea*Cúram* web client application without modifying the original components or their artifacts. Custom data conversion and sorting allows most aspects of the management of data in the presentationlayer of*Cúram* applications to be customized.

For more information, .

1.4 Compliancy for individual components

Read the following compliance information for individual components. Unless otherwise indicated, a number of general compliance statements apply to all components.

Where you want to reference an application class in your custom code:

- If the class is External, you are allowed to reference it.
- If the class is Internal, do not reference it in new code. You are supported for existing references in your code but they are discouraged.
- If the class is Access Restricted, you are not supported in referencing it.

Where you want to customize an application class in your custom code:

- If the class is Modeled, follow the detailed guidance for supported customizations.
- If the class is Non-Modeled, refer to its Javadoc or any configuration or development guide for its parent component for details of customization points.

The following table lists some examples of compliancy information for specific components. Ensure that you check the component documentation for each component that you work with.

Table 10: Specific compliancy guidance for individual components

Component	Details
Client Development Environment (CDEJ)	Files from the *`CuramCDEJ`* folder are copied to temporary build folders during the application build process. The presence of these files outside of the *`CuramCDEJ`* folder does not make them available for customization. For more information, see the 1.3 Compliancy for client development on page 22.
Server Development Environment (SDEJ)	**Important:** Cúram's cryptographic functionality is not supported for customer use beyond the documented usage in the *Cúram Server Developer's Guide* and *Cúram Security Handbook*. The *`bin`* directory of this component contains Apache Ant build scripts that must not be modified directly. You can update these scripts by creating new custom Ant scripts that use the Ant inheritance functionality. The drivers folder of this component contains database drivers for the application database. If necessary, you can replace these drivers can be replaced with the relevant driver for the database that you use, provided the database is a supported database version. **Note:** If a problem arises with a driver that was not included in the application, that is, was not tested and verified for use with the application, the customer might be requested to replace the driver with a tested version, while the specific issue is raised with the third-party vendor. Files from the *`CuramSDEJ`* folder are copied to temporary build folders during the application build process. The presence of these files outside of the *`CuramSDEJ`* folder does not make them available for customization.
Persistence Infrastructure	The Persistence Infrastructure cannot be customized. Customers must not place any custom code in the Persistence Infrastructure code packages (curam.util.persistence and all subpackages). For more information about how to use these APIs, .
CER Infrastructure	CER entities are considered internal and subject to change, and customers must not update them or query them except through the CER public API or DMX files. For more information about compliance for CER infrastructure, .
Dependency Manager	The Dependency Manager cannot be customized in any way. All Dependency Manager APIs are for internal development use only. The Dependency Manager includes all server artifacts in the `curam.dependency` code package and all its subpackages. The following components contribute to the Dependency Manager code package: • The CER Infrastructure • The core application For more information, .
Eligibility and Entitlement Engine API	For more information,.
Funded Program Management	For more information about how to customize this component, and the component Javadoc.
Cúram Incidents	For more information about how to customize any Incident Entities or replacing any Incident implementation, and the component Javadoc.
Cúram Citizen Context Viewer	For more information about how to customize this component, and the component Javadoc.
Inbox	For more information about how to configure and customize this component, .

Component	Details
Cúram Waitlists	For more information about how to customize this component, and the component Javadoc.
Cúram Business Intelligence and Analytics	For more information about how to customize this component,see .
Cúram Social Enterprise Collaboration	SocialEnterpriseCollaboration are the server components that are delivered with Social Enterprise Collaboration. For more information,.
Merative ™ Cúram Universal Access	Universal Access consists of the , CitizenWorkspace, CitizenWorkspaceAdmin, and WorkspaceServices components. For more information about customization, . For more information about customizing the classic Universal Access application, see the *Cúram Universal Access Customization Guide* and the component Javadoc.
Cúram Provider Management	For more information, and the component Javadoc.

Related information

1.5 Compliancy for deprecated functionality

Planned deprecation is used to reduce the impact of change on custom applications. During your development, review and remove any dependencies on deprecated functionality where practical.

For more information about deprecation, .

Notices

Privacy policy

Trademarks