merative™

Cúram 8.1.3

Health Care Reform Developer Guide

Note

Before using this information and the product it supports, read the information in Notices on page 189

Edition

This edition applies to Cúram 8.1, 8.1.1, 8.1.2, and 8.1.3.

© Merative US L.P. 2012, 2024

Merative and the Merative Logo are trademarks of Merative US L.P. in the United States and other countries.

Contents

1 Adding a product to the Cúram solution for Health Care Reform

Add a product to the Cúram solution for Health Care Reform (HCR) so that you can authorize, activate, and reassess the new product in the same way as for other HCR products.

You can configure the Eligibility Viewer, , and annual renewals so that the new product includes the same functions as existing HCR products.

Before you use this information, ensure that you familiar with the following information:

- How to create a new Cúram product. For more information about how to create a product, see *How to Build a Product*.
- The Cúram solution for Health Care Reform. For more information, see the *Cúram Income Support for Medical Assistance (Health Care Reform) business overview*.
- The compliancy guide for Cúram. For more information, see the *Development Compliancy Guide*.

> **Note:** The new State Benefit product that is referred to in the component CustomProduct is for example purposes only.

Related information

1.1 Key business flow for adding a product

The key business flow (KBF) provides an overview for how to add a product to the Cúram solution for HCR.

Click a business flow for a detailed business flow description.
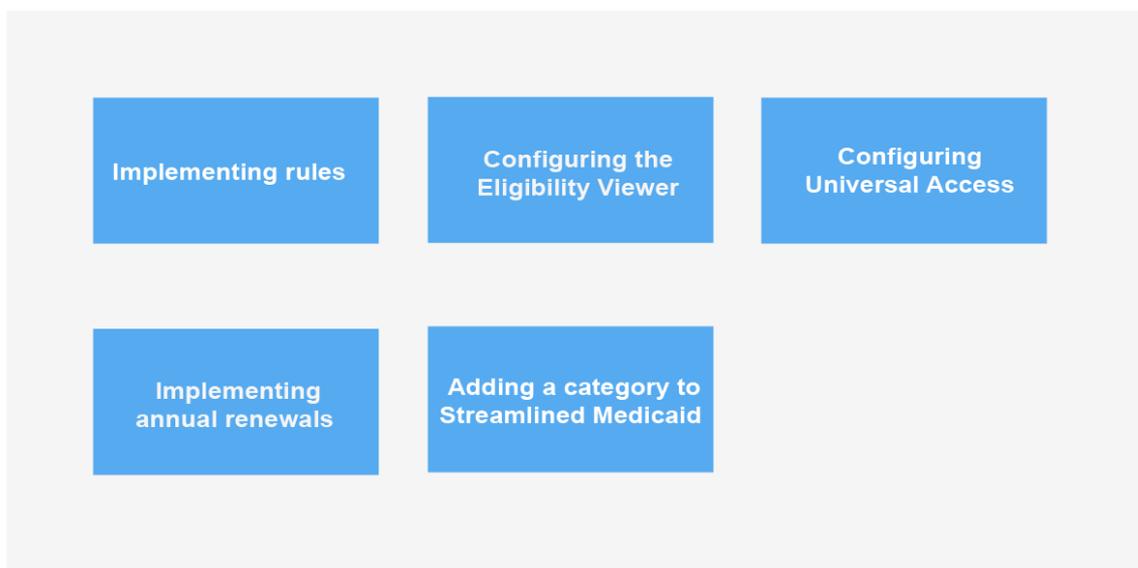
Implementing rules

Configuring the
Eligibility Viewer

Configuring
Universal Access

Implementing
annual renewals

Adding a category to
Streamlined Medicaid

1.2 Implementing rules

When you set up a new product, you must apply generic and specific rules.

When you set up any new product in Cúram, first apply generic rules. The rules apply to all products. For more information about the rules, see *How to Build a Product*.

You must also apply rule updates that are specific to HCR. Use the **Implementing rules** section to apply the required integration with the Cúram solution for HCR Program Group Logic. The Program Group Logic orchestrates the creation, authorization, activation, and reassessment of products.

Related information

Creating a program group

Program group rules indicate to program group logic the products that must be managed.

1. Create `StateBenefitProgramGroupRuleSet` with class StateBenefitProgram that extends `RuleSet AbstractProgramGroupRuleSet` and rule class default program.

```
<RuleSet
  name="StateBenefitProgramRuleSet"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../../../../../../../../catalog/RuleSet.xsd"
>

  <Class
    extends="DefaultProgram"
    extendsRuleSet="AbstractProgramGroupRuleSet"
    name="StateBenefitProgram"
  >
  …
  </Class>
</RuleSet>
```

At a minimum, override the preceding attributes with product-specific logic.

> **Note:** The solution for HCR creates an EligibilityCalculator rule class for each unit to assess. For example, Streamlined Medicaid creates a MemberEligibilityCalculator for each individual before Streamlined Medicaid determines eligibility per individual. This pattern is used because display rules can easily extract decisions for each individual. The units are also beneficial because the units provide the eligible unit list that is required by benefitUnitTimeline, as shown in the proceeding examples. While it is not a requirement, the attribute examples also use the pattern.

- **Product ID**

    - Attribute: `productID`
    - Description: The identification of a product. In the proceeding example, 45000 is the product ID for State Benefit.

```
<Attribute name="productID"/>
        <type>
          <javaclass name="Number"/>
        </type>
        <derivation>
          <Number value="45000"/>
        </derivation>
      </Attribute>
```

- **Product type code**

    - Attribute: `productTypeCode`

- Description: Product type code. In the proceeding example, DPT45001 is the product type code for State Benefit.

```
<Attribute name="productType">
        <type>
          <javaclass name="String"/>
        </type>
        <derivation>
            <String value="DPT45001"/>
        </derivation>
      </Attribute>
```

- **Eligibility**

  - Attribute: `isEligibleForProgramTimeline`
  - Description: The overall eligibility of the product that is combined from any unit eligibility.

```
<Attribute name="isEligibleTimeline">
        <type>
            <javaclass name="curam.creole.value.Timeline">
              <javaclass name="Boolean"/>
            </javaclass>
        </type>
         <derivation>
            <reference attribute="isEligibleTimeline">
              <reference attribute="stateBenefitCase"/>
            </reference>
         </derivation>
      </Attribute>
```

- **Coverage start date**

  - Attribute: `coverageStartDate`
  - Description: The start date of the coverage period.

```
<Attribute name="coverageStartDate">
        <type>
            <javaclass name="curam.util.type.Date"/>
        </type>
        <derivation>
          <reference attribute="coverageStartDate">
            <reference attribute="stateBenefitCase"/>
          </reference>
        </derivation>
      </Attribute>
```

- **Coverage end date**

  - Attribute: `coverageEndDate`
  - Description: The end date of the coverage period.

```
<Attribute name="coverageEndDate"/>
        <type>
            <javaclass name="curam.util.type.Date"/>
        </type>
        <derivation>
          <reference attribute="coverageEndDate">
            <reference attribute="stateBenefitCase"/>
          </reference>
        </derivation>
      </Attribute>
```

- **Product delivery start date**

  - Attribute: `pdCreationCheckStartDate`

- Description: The start date of the product delivery.

```
<Attribute name="pdCreationCheckStartDate">
        <type>
           <javaclass name="curam.util.type.Date"/>
        </type>
        <derivation>
          <reference attribute="coverageStartDate">
            <reference attribute="stateBenefitCase"/>
          </reference>
        </derivation>
     </Attribute>
```

- **Product delivery end date**

  - Attribute: `pdCreationCheckEndDate`
  - Description: The end date of the product delivery.

```
<Attribute name="pdCreationCheckEndDate">
        <type>
           <javaclass name="curam.util.type.Date"/>
        </type>
        <derivation>
          <reference attribute="coverageEndDate">
             <reference attribute="stateBenefitCase"/>
          </reference>
        </derivation>
        </derivation>
     </Attribute>
```

- **Participant eligibility that is based on the `benefitUnitTimeline` eligibility**

  - Attribute: `benefitUnitCaseParticipantRoleTimeline`
  - Description: Participant eligibility that is based on the `benefitUnitTimeline` eligibility.

```
<Attribute name="benefitUnitCaseParticipantRoleTimeline">
        <type>
           <javaclass name="curam.creole.value.Timeline">
             <javaclass name="List">
               <ruleclass
                 name="ConcernRole"
                 ruleset="ParticipantEntitiesRuleSet"
                 />
             </javaclass>
           </javaclass>
        </type>
        <derivation>
          <timelineoperation>
            <dynamiclist>
              <list>
                <intervalvalue>
                   <reference attribute="benefitUnitTimeline"/>
                </intervalvalue>
              </list>
              <listitemexpression>
                <reference attribute="participantRole">
                   <reference attribute="caseParticipantRoleRecord">
                      <current/>
                   </reference>
                 </reference>
               </listitemexpression>
              </dynamiclist>
             </timelineoperation>
            </derivation>
       </Attribute>
```

- **List of individual eligibility by timeline**

  - Attribute: benefitUnitTimeline
  - Description: List of individual eligibility by timeline.

```
<Attribute name="benefitUnitTimeline">
          <type>
            <javaclass name="curam.creole.value.Timeline">
              <javaclass name="List">
                <ruleclass
                  name="MemberUnit"
                  ruleset="HealthCareRuleSet"
              />
              </javaclass>
            </javaclass>
          </type>
          <derivation>
            <timelineoperation>
              <dynamiclist>
                <list>
                  <intervalvalue>
                    <reference attribute="eligibleUnitsTimeline">
                      <reference attribute="stateBenefitCase"/>
                    </reference>
                  </intervalvalue>
                </list>
                <listitemexpression>
                  <reference attribute="memberUnit">
                    <current/>
                  </reference>
                </listitemexpression>
              </dynamiclist>
            </timelineoperation>
          </derivation>
        </Attribute>
```

In the preceding examples, the StateBenefitCase is used:

```
<Attribute name="stateBenefitCase">
          <type>
            <ruleclass
              name="StateBenefitCase"
              ruleset="StateBenefitEligibilityAndEntitlementRuleSet"
            />
          </type>
          <derivation>
             <create
            ruleclass="StateBenefitCase"
            ruleset="StateBenefitEligibilityAndEntitlementRuleSet"
           >
            <specify attribute="parentIntegratedCaseID">
              <reference attribute="caseID"/>
            </specify>
          </create>
        </derivation>
    </Attribute>
```

As shown in the preceding code snippets, use the implementation to delegate
eligibility logic to the Eligibility and Entitlement rule set. For further examples, see the
StreamlineMedicaidProgramRuleSet.xml.

2. To create the program rule class, modify the *HCRProgramGroupRuleSet.xml* attribute `programListTimeline`.

```
<!-- State Benefit Product -->
      <create
         ruleclass="StateBenefitProgram"
         ruleset="StateBenefitProgramRuleSet"
       >
         <specify attribute="caseID">
           <reference attribute="caseID"/>
         </specify>
         <specify attribute="isInitialEnrollment">
            <reference attribute="isInitialEnrollment"/>
         </specify>
      </create>
```

Modifying eligibility and entitlement rules

Modify the created eligibility and entitlement rules.

The Cúram solution for HCR products does not define objectives. As a result, the `objectiveType` attributes return an empty list. However, custom products can, as required, use objectives.

> **Note:** In the Cúram solution for HCR, the `objectiveType` element is processed in a specific way.

```
<RuleSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" name="StateBenefitEligibilityAndEntitlementRuleSet"
 xsi:noNamespaceSchemaLocation="http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <Class extends="DefaultProduct"
 extendsRuleSet="DefaultProductEligibilityEntitlementRuleSet"
 name="StateBenefitProduct">

    <Attribute name="objectiveTypes">
      <Annotations>
        <Legislation link="http://www.curamsoftware.com/Dummy/
AttributeLegislationWithProtocol.html"/>
      </Annotations>
      <type>
        <javaclass name="List">
          <ruleclass name="AbstractObjectiveType"
ruleset="ProductEligibilityEntitlementRuleSet"/>
        </javaclass>
      </type>
      <derivation>
        <fixedlist>
          <listof>
            <ruleclass name="AbstractObjectiveType"
ruleset="ProductEligibilityEntitlementRuleSet"/>
          </listof>
          <members/>
        </fixedlist>
      </derivation>
    </Attribute>
  </Class>

  <Class extends="InsuranceAffordabilityProduct" extendsRuleSet="HealthCareRuleSet"
 name="StateBenefitCase">

  ...

  </Class>

</RuleSet>
```

The contract provides flexibility in the implementation of the product rules. However, to provide individual eligibility for the update of case groups and the display of the Eligibility Viewer attributes must, at a minimum, be exposed to display rules. If it is a requirement of the new product, you might also need to expose eligibility in the cascade to prevent dual benefits.

Updating cascade logic

In the Cúram solution for HCR, an individual or household is not eligible for more than one product for the same time period. If a new product is not allowed to overlap with existing products, then you must update the cascade logic.

About this task

The cascade in the following diagram has been encoded into rules so that the eligibility periods from the more beneficial products are used as periods of ineligibility for less beneficial products. For example, in the diagram, Streamlined Medicaid is more beneficial than CHIP.

Cascading Eligibility Check

More beneficial

Streamlined Medicaid

↓

HCR CHIP

↓

State Basic Health Plan (configurable)

↓

Insurance Assistance

↓

Unassisted Qualified Health Program
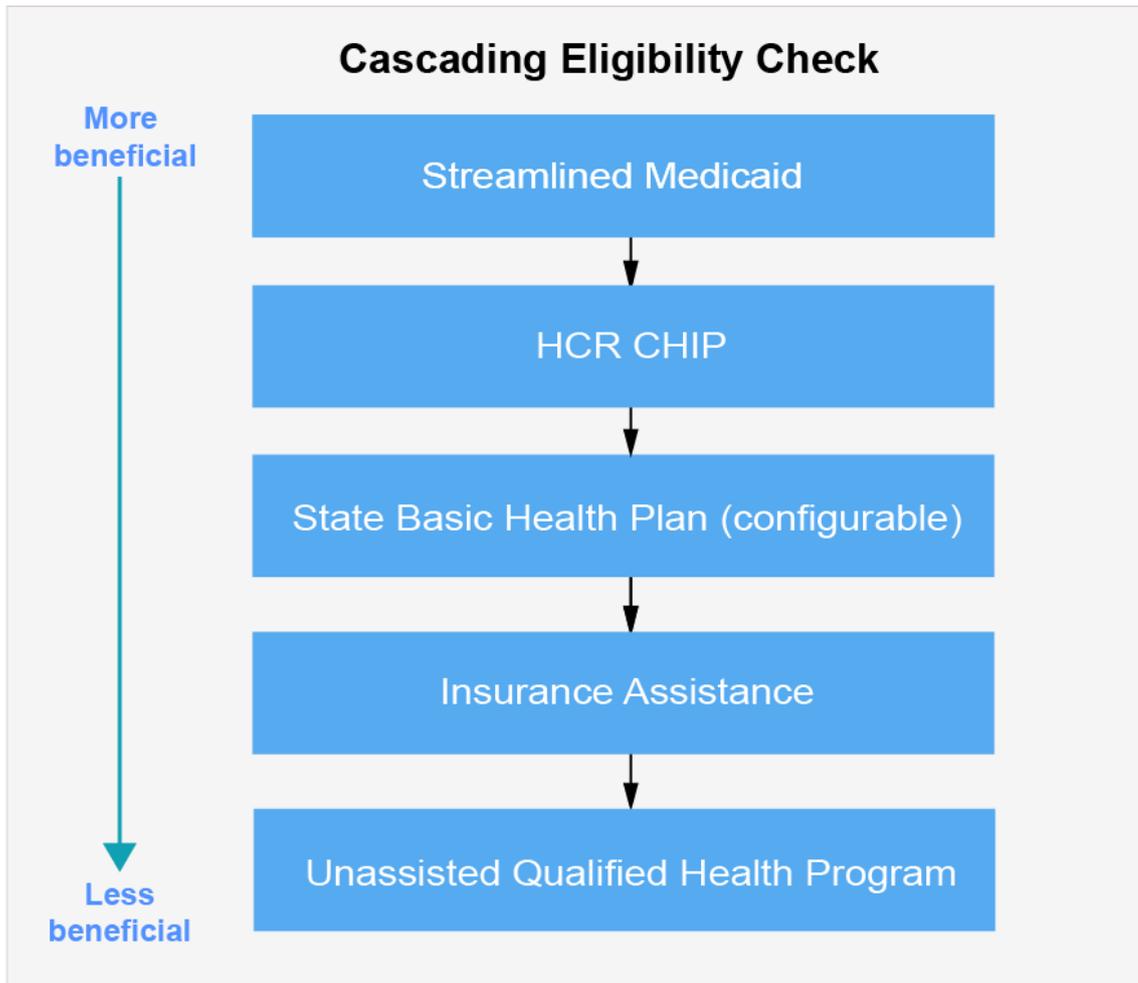
Less beneficial

If a new product is not allowed to overlap, then you must update the cascade logic.

Procedure

1. Adjust the eligibility for the new product so that it creates periods of ineligibility for eligible periods of the more beneficial products. To make the eligibility adjustment, create the rule calculator and obtain its eligibility period as shown in the proceeding example.

```
<Attribute name="hasPassedCascadingEligibilityTimeline">
        <type>
           <javaclass name="curam.creole.value.Timeline">
              <javaclass name="Boolean"/>
           </javaclass>
        </type>
        <derivation>
           <timelineoperation>
              <not>
                 <intervalvalue>
                    <reference attribute="isEligibleForStreamlinedMedicaid"/>
                 </intervalvalue>
              </not>
           </timelineoperation>
        </derivation>
     </Attribute>

     <Attribute name=" isEligibleForStreamlinedMedicaid ">
        <type>
           <javaclass name="curam.creole.value.Timeline">
              <javaclass name="Boolean"/>
           </javaclass>
        </type>
        <derivation>
           <timelineoperation>
              <intervalvalue>
                 <reference attribute="isEligibleTimeline">
                    <create
                      ruleclass="MemberEligibilityCalculator"
                      ruleset="StreamlineMedicaidEligibilityAndEntitlementRuleSet">
                      <specify attribute="caseParticipantRoleRecord">
                         <reference attribute="caseParticipantRoleRecord"/>
                      </specify>
                      <specify attribute="useMonthlyIncomeToDetermineEligibility">
                         <reference attribute="useMonthlyIncomeToDetermineEligibility"/>
                      </specify>
                      <specify attribute="insuranceAffordabilityIntegratedCase">
                         <reference attribute="insuranceAffordabilityIntegratedCase"/>
                      </specify>
                    </create>
                 </reference>
              </intervalvalue>
           </timelineoperation>
        </derivation>
     </Attribute>
```

2. Update the cascade of products that are less beneficial to reflect the eligibility of the new product. Table 1 lists the attributes that must be updated. For example, if the new product is less beneficial than Streamlined Medicaid, but is more beneficial than HCR CHIP, then the cascade rules for the following products must be updated to include reference to the eligibility of the new product:

   - HCR CHIP
   - State Basic
   - Insurance Assistance
   - Unassisted Qualified Health Program

Product	Cascade rule `(RuleSet.RuleClass.attribute)`
Streamlined Medicaid	Not applicable as it is the lowest in the cascade.

Product	Cascade rule (RuleSet.RuleClass.attribute)
HCR CHIP	• CHIPEligibilityAndEntitlementRuleset • MemberEligibilityCalculator.isEligibleMemberTimeline
State Basic	• StateBasicPlanEligibilityAndEntitlementRuleSet • CascadingEligibilityCalculator • hasPassedCascadingEligibilityTimeline
Insurance Assistance	• InsuranceAssistanceEligibilityAndEntitlementRuleSet • CascadingEligibilityCalculator • hasPassedCascadingEligibilityTimeline
Unassisted Qualified Health Program	• UnassistedQualifiedHealthPlanRuleSet • MemberEligibilityCalculator • isEligibleMemberTimeline

The products are listed in declining order from most to least beneficial.

3. Modify the affected attributes to reference, as applicable, for either the eligibility of the individuals or the eligibility of the units of the new product. Use the proceeding example to calculate the appropriate rules to determine eligibility.

```
<Attribute name="isEligibleForStateBenefitTimeline">
        <type>
          <javaclass name="curam.creole.value.Timeline">
            <javaclass name="Boolean"/>
          </javaclass>
        </type>
        <derivation>
          <timelineoperation>
            <intervalvalue>
              <reference attribute="isEligibleTimeline">
                <create
                  ruleclass="MemberEligibilityCalculator"
                  ruleset="StateBenefitEligibilityAndEntitlementRuleSet">
                  <specify attribute="caseParticipantRoleRecord">
                    <reference attribute="caseParticipantRoleRecord"/>
                  </specify>
                  <specify attribute="useMonthlyIncomeToDetermineEligibility">
                    <reference attribute="useMonthlyIncomeToDetermineEligibility"/
>
                  </specify>
                  <specify attribute="insuranceAffordabilityIntegratedCase">
                    <reference attribute="insuranceAffordabilityIntegratedCase"/>
                  </specify>
                </create>
              </reference>
            </intervalvalue>
          </timelineoperation>
        </derivation>
      </Attribute>
```

Display rules

Standard product guidelines apply to display rules.

There are no specific requirements to integrate with the Cúram solution for HCR. However, to ensure that you provide caseworkers with a consistent experience review the existing products.

1.3 Product management examples

Each product type must include its own product manager that is bound against its code table type.

The concept of a product manager was introduced in the Cúram solution for HCR. Use the product manager to create products, modify products, and update case groups. Each product type must include its own product manager so that the different ways that products are created and modified can be managed. For example, for the Streamlined Medicaid implementation, all eligible members must be added to the same product. In contrast, for the Insurance Assistance implementation, a product must be created for each eligible unit.

Example 1

The example shows the `HealthCareProductManager` interface that the new product implements.

```
      /**
       * HealthCareProductManager interface to be implemented per HCR product type.
       */
      @AccessLevel(AccessLevelType.EXTERNAL)
      @Implementable
      public interface HealthCareProductManager {

        /**
         * Responsible for the creation and modification (such as adding a new
participant)
         * of a product based on the <code>programDetails</code> passed.
         *
         * <p/>
         * This method should return details of any new products created using
         * {@link ProcessedProgramDetails #getNewPDCaseList()}.
         *
         * <p/>
         * This method should return details of any change to existing product
         * certification period in
         * {@link ProcessedProgramDetails #getPdCaseIDToAdjustedCertPeriod()}.
         *
         * @param programDetails {@link ProgramDetails}
         * @return {@link ProcessedProgramDetails}
         *
         * @throws AppException
         * @throws InformationalException
         */
        ProcessedProgramDetails manageProgram(final ProgramDetails programDetails)
          throws AppException, InformationalException;

        /**
         * Responsible for maintaining <code>casegroups</code> for this product type.
         *
         * <p/>
         * Store the <code>casegroups</code> details for this product from the latest
         * determination read with the {@link CaseGroupsDetails #getCaseKey()}
         * provided.
         *
         * @param caseGroupDetails {@link CaseGroupsDetails}
         * @throws AppException
         * @throws InformationalException
         */
        void manageCaseGroup(CaseGroupsDetails caseGroupDetails)
          throws AppException, InformationalException;
      }
```

Example 2

The example shows how to bind the implementation against its product type.

```
final MapBinder<String, HealthCareProductManager> healthCareManagerMapbinder =
            MapBinder.newMapBinder(binder(), String.class,
                HealthCareProductManager.class);

        healthCareManagerMapbinder.addBinding(PRODUCTTYPE.STATEBENEFIT)
            .to(StateBenefitProductManager.class);
```

Example 3

The `HealthCareProductManagerHelper` provides methods that are common to `HealthCareProductManager` implementations.

```java
/**
 * Create a new Product with the values passed.
 *
 * @param key CaseKey Integrated Case ID.
 * @param programID Long ID.
 * @param primaryClient Long ID.
 * @param programDetails {@link ProgramDetails}
 * @param certificationStartDate Date start.
 * @param certificationEndDate Date end.
 * @return CaseKey Product Delivery CaseKey.
 *
 * @throws AppException
 * @throws InformationalException
 */
public CaseKey createProductDelivery(final CaseKey key,
  final long programID, final long primaryClient,
  final ProgramDetails programDetails, final Date certificationStartDate,
  final Date certificationEndDate)
  throws AppException, InformationalException

/**
 * Update the existing product delivery adding new participants to the product
 * as identified by {@link ProgramDetails #getBenefitMemberConcernRoleList()}
 * that are not currently participants.
 *
 * @param programDetails {@link ProgramDetails}
 * @return List of product delivery case IDs that have had participants added.
 *
 * @throws AppException
 * @throws InformationalException
 */
public List<Long>
  updateExistingProductDelivery(final ProgramDetails programDetails)
    throws AppException, InformationalException

/**
 * Updates the <code>CaseGroups</code> table for CG1 (Benefit) and CG3
 * (Member), by reading display rule values from the latest decision.
 * Where the product type is Streamlined Medicaid, casegroups of type
 * CG4 (Category) will also be updated. The approach taken is to remove
 * existing case group entries for this product and insert new entries
 * reflecting the latest decision.
 *
 * @param caseGroupsDetails {@link CaseGroupsDetails} populated with caseKey.
 *
 * <p/>
 * <strong>IMPORTANT</strong><br/>
 * This method requires an implementation of
 * {@link HCRCaseGroupsRuleReference} bound to the product type identified by
 * the <code>caseKey</code> passed. If no such binding is found then this
 * method will immediately return without doing anything.
 *
 * @throws AppException
 * @throws InformationalException
 */
public void maintainCaseGroups(final CaseGroupsDetails caseGroupsDetails)
  throws AppException, InformationalException
```

Example 4

The `HealthCareProductManagerHelper.maintainCaseGroups` requires an implementation of `HCRCaseGroupsRuleReference` to identify the rule attributes to read when it populates case group records.

```
public interface HCRCaseGroupsRuleReference {

  /**
   * @return String. The reference for the display rules as defined in
   * CREOLEProductDecisionDispCat.categoryRef.
   */
  String getCategoryReference();

  /**
   * @return String. The name of the display rule class that contains the
   * attributes that determine the member and benefit groups for a given
   * decision.
   */
  String getDisplayRuleClassName();

  /**
   * @return String. The name of the display rule attribute within
   * displayRuleClassName, that defines the benefit group for a given decision.
   * The attribute type must be a Timeline List of Number. Each period being the
   * List of Concern Role IDs for the members in receipt of benefit for that
   * period of time.
   */
  String getBenefitGroupAttributeName();

  /**
   * @return String. The name of the display rule attribute within
   * displayRuleClassName, that defines the member group for a given decision.
   * The attribute type must be a Timeline List of Number. Each period being the
   * List of Concern Role IDs for the product members for that period of time.
   */
  String getMemberGroupAttributeName();
}
```

Ensure to bind the implementation against the `ProductType` code. The following code shows an example of binding the implementation against the `ProductType` code:

```
final MapBinder<String, HCRCaseGroupsRuleReference> caseGroupsRuleReferenceMapBinder =
      MapBinder.newMapBinder(binder(), String.class,
        HCRCaseGroupsRuleReference.class);

    caseGroupsRuleReferenceMapBinder.addBinding(PRODUCTTYPE.STATEBENEFIT)
      .to(StateBenefitCaseGroupsRuleReference.class);
```

1.4 Configuring the Eligibility Viewer

Configure the Eligibility Viewer for a new product in the same way that you configure the Eligibility Viewer for other products. For more information, see *Configuring the Eligibility Viewer*.

You do not have to configure the Eligibility Viewer specifically for HCR.

Related concepts

1.5 Configuring Universal Access

Use the Universal Access **Results** page to determine eligibility for the new product. For more information, see *Configuring the Motivation Results Page* in the *Universal Access Configuration Guide*.

Displaying information on the results page

Display information on the Merative ™ Cúram Universal Access results page to determine eligibility for the new product.

Procedure

1. Add an EligibilityCategory code table entry for the product for the Universal Access flow.

```
<codetable
        java_identifier="ELIGIBILITYCATEGORY"
        name="EligibilityCategory"
     >
     <displaynames>
       <locale language="en">Eligibility Category</locale>
     </displaynames>
     <code
       default="false"
       java_identifier="STATEBENEFIT"
       status="ENABLED"
       value="EC26163"
     >
       <locale
          language="en"
          sort_order="0"
       >
         <description>State Benefit</description>
         <annotation/>
       </locale>
     </code>
   </codetable>
```

2. Add the eligibility check to the Universal Access rules inside the *HealthCareDisplayRuleSet.xml* category. Adjust the eligibility of other programs that are referenced if the other programs are affected by the new program.

```xml
<Attribute name="eligibilityResultList">
            <type>
              <javaclass name="List">
                <ruleclass name="EligibilityResult"/>
              </javaclass>
            </type>
            <derivation>
              <joinlists>
                <fixedlist>
                  <listof>
                    <javaclass name="List">
                      <ruleclass name="EligibilityResult"/>
                    </javaclass>
                  </listof>
                  <members>
                    <reference attribute="chipOnly"/>
                    <reference attribute="stateBasicHealthPlanResult"/>
                    <reference attribute="iaTest"/>
                    <reference attribute="unassistedQualifiedHealthCare"/>
                    <reference attribute="medicaidTest"/>
                    <reference attribute="neTest"/>
                    <reference attribute="emergencyMedicaidTest"/>
                    <reference attribute="stateBenefitTest"/>
                  </members>
                </fixedlist>
              </joinlists>
            </derivation>
          </Attribute>
```

3. To implement the rules for the eligibility test, create an EligibilityResult for each eligible member or group.

   Note that the following code sample shows only the high-level attribute implementation. Ensure that eligibility is referenced from the `entitlementRules` attribute. The

`entitlementRules` attribute is the EligibilityDeterminationCalculator in the HealthCareReformEligibilityRuleset.

```
<Attribute name="stateBenefitTest">
            <type>
              <javaclass name="List">
                <ruleclass name="EligibilityResult"/>
              </javaclass>
            </type>
            <derivation>
              <choose>
                <type>
                  <javaclass name="List">
                    <ruleclass name="EligibilityResult"/>
                  </javaclass>
                </type>
                <when>
                  <condition>
                    <reference attribute="anyoneEligibleForStateBenefit"/>
                  </condition>
                  <value>
                    <fixedlist>
                      <listof>
                        <ruleclass name="EligibilityResult"/>
                      </listof>
                      <members>
                        <create ruleclass="EligibilityResult">
                          <specify attribute="eligibilityCategory">
                            <Code table="EligibilityCategory">
                                <String value="EC26163"/>
                            </Code>
                          </specify>
                          <specify attribute="benefitGroup">
                            <reference attribute="stateBenefitMembers"/>
                          </specify>
                          <specify attribute="objectivesList">
                            <reference attribute="combinedStateBenefitObjectives"/
>
                          </specify>
                          <specify
 attribute="satisfiesSpecialEnrollmentQualifyingEvent">
                              <true/>
                          </specify>
                          <specify attribute="isCOCSpecialEnrollment">
                            <false/>
                          </specify>
                        </create>
                      </members>
                    </fixedlist>
                  </value>
                </when>
                <otherwise>
                    <value>
                     <fixedlist>
                      <listof>
                        <ruleclass name="EligibilityResult"/>
                      </listof>
                      <members/>
                     </fixedlist>
                    </value>
                </otherwise>
              </choose>
            </derivation>
        </Attribute>
```

4. To reference the eligibility, add logic to the Category result in the HealthCareDisplayRuleSet.

```xml
<Attribute name="result">
            <Annotations>
              <Motivation_Display_Attribute/>
            </Annotations>
            <type>
              <javaclass name="List">
                <ruleclass name="Result"/>
              </javaclass>
            </type>
            <derivation>
            ...
              <when>
                <condition>
                  <equals>
                    <reference attribute="eligibilityCategory">
                      <current/>
                    </reference>
                    <Code table="EligibilityCategory">
                      <String value="EC26163"/>
                    </Code>
                  </equals>
                </condition>
                <value>
                  <fixedlist>
                    <listof>
                      <ruleclass name="Result"/>
                    </listof>
                    <members>
                      <create ruleclass="Result">
                        <specify attribute="type">
                          <String value="SB"/>
                        </specify>
                        <specify attribute="resultID">
                          <String value="SB"/>
                        </specify>
                        <specify attribute="context">
                        <fixedlist>
                          <listof>
                            <ruleclass name="HealthCareResultContext"/>
                          </listof>
                          <members/>
                        </fixedlist>
                        </specify>
                        <specify attribute="status">
                          <choose>
                            <type>
                              <javaclass name="String"/>
                            </type>
                            <when>
                             <condition>
                                <reference attribute="isResultActionVisible">
                                  <current/>
                                </reference>
                             </condition>
                             <value>
                               <String value="RPA26323"/>
                             </value>
                            </when>
                            <otherwise>
                             <value>
                               <String value="RPA26324"/>
                             </value>
                            </otherwise>
                          </choose>
                        </specify>
                        <specify attribute="members">
                          <reference attribute="benefitGroup">
                            <current/>
                          </reference>
                        </specify>
                        <specify
  attribute="satisfiesSpecialEnrollmentQualifyingEvent">
                          <true/>
                        </specify>
                        <specify attribute="benefitList">
                          <reference attribute="objectivesList">
                            <current/>
                          </reference>
                        </specify>
                      </create>
                    </members>
                  </fixedlist>
                </value>
              </when>
```

5. Add logic in the resultDescription attribute in the Result rule class in the HealthCareDisplayRuleSet to display the appropriate resource message for the result added in step 4.

```xml
<Attribute name="resultDescription">
            <Annotations>
              <Motivation_Display_Attribute/>
            </Annotations>
            <type>
              <javaclass name="curam.creole.value.Message"/>
            </type>
            <derivation>
            ...
              <when>
                  <condition>
                    <equals>
                      <reference attribute="resultID"/>
                      <String value="SB"/>
                    </equals>
                  </condition>
                  <value>
                    <ResourceMessage
                      key="Result.description.StateBenefit"
                      resourceBundle="curam.healthcare.HealthCareDisplay"
                    >
                      <reference attribute="memberNamesDisplay">
                        <reference attribute="householdPersonsDisplay"/>
                      </reference>
                    </ResourceMessage>
                  </value>
              </when>
              ...
            </derivation>
        </Attribute>
```

6. To enable the results page to display more detailed information, link a UIM fragment with the product code in MotivationResultsProgramDetailsURLMapping.properties, as shown in the following code sample:

```
IA=HealthCare_iaBenefitFragment1
        CHIP=HealthCare_chipBenefitFragment1
        ESCELIGIBLE=HealthCare_availableEmployerPlansFragment1
        ESCINELIGIBLE=HealthCare_availableEmployerPlansFragment1
        NE=HealthCare_ineligibleToPurchasePlansFragment
        SB=HealthCare_stateBenefitFragment
```

Adjust the flow based on the projected eligibility

Within the Cúram solution for HCR IEG Merative ™ Cúram Universal Access script, the flow varies depending on projected eligibility.

If you need to modify or reuse the approach in a different part of the script, the eligibility rules are run within CustomFunctionDetermineEligibility. CustomFunctionDetermineEligibility references rules in the HealthCareReformEligibilityRuleSet EligibilityDeterminator. Use the details in Table 1 to modify or reuse the approach in a different part of the script. The call to CustomFunctionDetermineEligibility makes sense only when sufficient information is gathered to make the result of the rules execution meaningful.

Table 1 lists the mapping between the rules attributes that are read from HealthCareReformEligibilityRuleSet EligibilityDeterminator and the data store entities that are updated based on those values.

Table 1:

Rule attribute	Data store	Condition
eligibleMedicaidMembers	• **Entity: Household** Attribute: isAnyMemberEligibleForMedicaidOrChip • **Entity: Person** Attributes: • isMedicaidCHIPEligible • isEligibleForSBHP • isIAEligible • isEligibleForUQHP • isNotEligible	If the list is not empty, set isAnyMemberEligibleForMedicaidOrChip to **true**. For each person in the list, set isMedicaidCHIPEligible to **true** and set the others to **false**.
eligibleChildrenForCHIP	• **Entity: Household** Attribute: isAnyMemberEligibleForMedicaidOrChip • **Entity: Person** Attributes: • isMedicaidCHIPEligible • isEligibleForSBHP • isIAEligible • isEligibleForUQHP • isNotEligible	If the list is not empty, set isAnyMemberEligibleForMedicaidOrChip to **true**. For each person in the list, set isMedicaidCHIPEligible to **true** and set the others to **false**.
allEligibleIAMembers	• **Entity: Household** Attribute: isAnyMemberEligibleForIA • **Entity: Person** Attributes: • isMedicaidCHIPEligible • isEligibleForSBHP • isIAEligible • isEligibleForUQHP • isNotEligible	If the list is not empty, set isAnyMemberEligibleForIA to **true**. For each person in the list, set isIAEligible to **true** and set the others to **false**.
allEligibleUQHPMembers	• **Entity: Household** Attribute: isAnyMemberEligibleForUQHP • **Entity: Person** Attributes: • isMedicaidCHIPEligible • isEligibleForSBHP • isIAEligible • isEligibleForUQHP • isNotEligible	If the list is not empty, set isAnyMemberEligibleForUQHP to **true**. For each person in the list, set set isEligibleForUQHP to **true** and set the others to **false**.

Rule attribute	Data store	Condition

nonEligibleMembers	**Entity: Household**
	Attribute: `isAnyMemberNotEligible`
	
	• **Entity: Person**
	Attributes:
	
	  • `isMedicaidCHIPEligible`
	  • `isEligibleForSBHP`
	  • `isIAEligible`
	  • `isEligibleForUQHP`
	  • `isNotEligible`
		If the list is not empty, set `isAnyMemberNotEligible` to **true**.
		
		For each person in the list, set `isNotEligible` to **true** and set the others to **false**.

1.6 Implementing annual renewals

For Qualified Health Plan products, you can run the batches for annual renewals and periodic data matching by implementing the renewal logic within manageProgram of the HealthCareProductManager interface.

You cannot use the Streamlined Medicaid renewal batch because there is a restriction on the product type code.

For more information about annual renewals and periodic data matching, see *Implementing periodic data matching and annual renewals*.

Related tasks

Implementing periodic data matching and annual renewals on page 97
From a technical perspective, annual renewals is a specific use case of periodic data matching, with some specific annual renewal requirements. The shared technical infrastructure that is provided for periodic data matching and annual renewals contains the required configuration, customization, and extension points for you to implement your custom solution.

1.7 Adding a category to Streamlined Medicaid example

To add a category to Streamlined Medicaid requires that you add appropriate code tables, coverage rules, eligibility rules, and display rules. The topics in this section provide an example with samples.

> **Note:** For example purposes only, the following topics reference a category for 65 years and older in the CustomProduct component.

Code table samples

The code table samples in this topic outline two code types. The sample eligibility and display rules use the two code types in the subsequent topics in this Streamlined Medicaid category example.

CT_CoverageCategory.ctx

```
<codetables package="curam.healthcare.planmanagement.codetable">
          <codetable
            java_identifier="COVERAGECATEGORY"
            name="CoverageCategory"
            >
            <code
              default="false"
              java_identifier="SIXTYFIVEOROLDER"
              status="ENABLED"
              value="CC6"
            >
              <locale
                language="en"
                sort_order="0"
              >
                <description>Sixty Five or Older</description>
                <annotation/>
              </locale>
            </code>
          </codetable>
        </codetables>
```

CT_MedicaidCoverageType.ctx

```
<codetables package="curam.codetable">
          <codetable
            java_identifier="MEDICAIDCOVERAGETYPE"
            name="MedicaidCoverageType"
        >
            <code
              default="false"
              java_identifier="SIXTYFIVEOROLDER"
              status="ENABLED"
              value="MCT26013"
           >
            <locale
              language="en"
               sort_order="0"
            >
                <description>Sixty Five or Older</description>
                <annotation/>
            </locale>
            </code>
          </codetable>
        </codetables>
```

Coverage rules samples

Use a Boolean timeline attribute to define the coverage rules for this Streamlined Medicaid category example.

To ensure consistency with the other categories, add the attribute to the MedicaidCoverageCategoryCalculator rule class in the CoverageCategoryRuleSet.

```
<Attribute name="is65OrOlderCategoryTimeLine">
        <type>
          <javaclass name="curam.creole.value.Timeline">
            <javaclass name="Boolean"/>
          </javaclass>
        </type>
        <derivation>
          <reference attribute="isInAgeRangeTimeline">
              <create
                ruleclass="AgeRangeCalculator"
                ruleset="HealthCareRuleSet"
              >
                <specify attribute="personRecord">
                  <reference attribute="personRecord">
                    <reference attribute="memberUnit"/>
                  </reference>
                </specify>
                <specify attribute="minimumAgeLimit">
                  <Number value="65"/>
                </specify>
                <specify attribute="maximumAgeLimit">
                  <Number value="200"/>
                </specify>
              </create>
          </reference>
        </derivation>
      </Attribute>
```

In the existing `coverageCategoryHelperList` rule attribute in the InsuranceAffordabilityIntegratedCase rule class in HealthCareRuleSet, add an entry to the existing list that represents the new category. In the example, the list is used during eligibility processing.

```
<create ruleclass="CoverageCategoryHelper">
        <specify attribute="coverageCategory">
          <Code table="MedicaidCoverageType">
            <String value="MCT26013"/>
          </Code>
        </specify>
        <specify attribute="fplLimit">
          <reference attribute="medicaidUpperLimitFor65OrOlder"/>
        </specify>
      </create>
      ..


      <Attribute name="medicaidUpperLimitFor65OrOlder">
        <type>
          <javaclass name="Number"/>
        </type>
        <derivation>
          <Number value="200"/>
        </derivation>
      </Attribute>
```

Eligibility rules samples

The eligibility rules samples in this topic use the code table types that are defined in the code table samples previously in this Streamlined Medicaid category example. Update the Streamlined

Medicaid financial calculations so that appropriate logic is applied while qualified for the new category in StreamlineMedicaidEligibilityAndEntitlementRuleSet FinancialsCPRCalculator.

- **hasPassedFinancialRulesTimelineWithDisregard**
  The rule determines whether the person passes financial rules with 5% income disregard. The rules vary from category to category. The new category must be added to the existing logic.

```
…
        <when>
         <condition>
           <all>
             <fixedlist>
               <listof>
                 <javaclass name="Boolean"/>
               </listof>
               <members>
                 <equals>
                   <reference attribute="coverageCategory">
                     <current/>
                   </reference>
                   <Code table="MedicaidCoverageType">
                     <String value="MCT26013"/>
                   </Code>
                 </equals>
                 <intervalvalue>
                   <reference attribute="is65OrOlderCategoryTimeLine"/>
                 </intervalvalue>
               </members>
             </fixedlist>
           </all>
         </condition>
         <value>
           <compare comparison="&lt;=">
             <intervalvalue>
               <reference attribute="incomePercentageAsFPL"/>
             </intervalvalue>
             <reference attribute="fplLimit">
               <current/>
             </reference>
           </compare>
         </value>
        </when>
…
<Attribute name="is65OrOlderCategoryTimeLine">
<type>
  <javaclass name="curam.creole.value.Timeline">
    <javaclass name="Boolean"/>
  </javaclass>
</type>
<derivation>
  <timelineoperation>
    <all>
      <fixedlist>
        <listof>
          <javaclass name="Boolean"/>
        </listof>
        <members>
          <intervalvalue>
            <reference attribute="coveragePeriodTimeline"/>
          </intervalvalue>
          <intervalvalue>
            <reference attribute="is65OrOlderCategoryTimeLine">
              <reference attribute="coverageCategory"/>
            </reference>
          </intervalvalue>
        </members>
      </fixedlist>
    </all>
  </timelineoperation>
</derivation>
</Attribute>
```

- **hasPassedFinancialRulesTimelineWODisregard**
  This rule determines whether the person passes financial rules without any income disregard.
  The rules vary from category to category. The new category must be added to the existing
  logic.

```
…
        <when>
          <condition>
            <intervalvalue>
              <reference attribute="is65OrOlderCategoryTimeLine">
                <reference attribute="coverageCategory"/>
              </reference>
            </intervalvalue>
          </condition>
          <value>
            <compare comparison="&lt;=">
              <intervalvalue>
                <reference attribute="incomePercentageAsFPL"/>
              </intervalvalue>
              <reference attribute="medicaidUpperLimitFor65OrOlder">
                <reference attribute="insuranceAffordabilityIntegratedCase">
                      <reference attribute="memberUnit"/>
                </reference>
              </reference>
            </compare>
          </value>
        </when>
        …
```

- **eligibleCoverageCategory**
  The rule must be used to call the coverage rules for the new category to determine the correct coverage type.

```
…
        <when>
         <condition>
           <all>
             <fixedlist>
               <listof>
                 <javaclass name="Boolean"/>
               </listof>
               <members>
                 <intervalvalue>
                   <reference attribute="is65OrOlderCategoryTimeLine">
                     <reference attribute="coverageCategory"/>
                   </reference>
                 </intervalvalue>
                 <intervalvalue>
                   <reference attribute="medicaidPercentageFPL65OrOlderTimeline"/>
                 </intervalvalue>
               </members>
             </fixedlist>
           </all>
         </condition>
         <value>
           <Code table="MedicaidCoverageType">
             <String value="MCT26013"/>
           </Code>
         </value>
        </when>
        …
        <Attribute name="medicaidPercentageFPL65OrOlderTimeline">
        <type>
          <javaclass name="curam.creole.value.Timeline">
            <javaclass name="Boolean"/>
          </javaclass>
        </type>
        <derivation>
          <timelineoperation>
            <compare comparison="&lt;=">
              <intervalvalue>
                <reference attribute="incomePercentageAsFPL"/>
              </intervalvalue>
              <reference attribute="medicaidUpperLimitFor65OrOlder">
                <reference attribute="insuranceAffordabilityIntegratedCase">
                 <reference attribute="memberUnit"/>
                </reference>
              </reference>
            </compare>
          </timelineoperation>
        </derivation>
        </Attribute>
```

Display rules samples

The display rules samples in this topic use the code table types that are defined in the code table samples previously in this Streamlined Medicaid category example. For the sample category that

is being added to Streamlined Medicaid, modify the display rules to show the new category in the determination.

- **StreamlineMedicaidSummaryCategory membersCategoryDetails**

```xml
<when>
  <condition>
    <equals>
      <intervalvalue>
        <reference attribute="adverseActionsEligibleCoverageCategory">
          <reference attribute="financialsCPRCalculator">
            <current alias="eligibiltyUnit"/>
          </reference>
        </reference>
      </intervalvalue>
      <Code table="MedicaidCoverageType">
        <String value="MCT26013"/>
      </Code>
    </equals>
  </condition>
  <value>
    <ResourceMessage
      key="Summary.CoverageCategory.Is65OrOlderCategory"
      resourceBundle="curam.hcrcase.rules.StreamlineMedicaidDisplay"
    />
  </value>
</when>
```

- **StreamlineMedicaidSummaryCategory membersCategoryType**

```xml
<when>
  <condition>
    <equals>
      <intervalvalue>
        <reference attribute="adverseActionsEligibleCoverageCategory">
          <reference attribute="financialsCPRCalculator">
            <current alias="eligibiltyUnit"/>
          </reference>
        </reference>
      </intervalvalue>
      <Code table="MedicaidCoverageType">
        <String value="MCT26013"/>
      </Code>
    </equals>
  </condition>
  <value>
    <String value="MCT26013"/>
  </value>
</when>
```

- **StreamlineMedicaidSummaryCategorySubScreen membersEligibilityCriteria**

```xml
<when>
  <condition>
    <intervalvalue>
      <reference attribute="is65OrOlderCategoryTimeLine">
        <reference attribute="coverageCategory">
          <reference attribute="financialsCPRCalculator">
          </reference>
        </reference>
      </reference>
    </intervalvalue>
  </condition>
  <value>
    <ResourceMessage
      key="Summary.EligibilityCriteria.Is65OrOlderCategoryTimeLine"
      resourceBundle="curam.hcrcase.rules.StreamlineMedicaidDisplay"
    />
  </value>
</when>
```

- **StreamlineMedicaidDisplay.properties**

```
Summary.CoverageCategory.Is65OrOlderCategory=65 or Older
Summary.EligibilityCriteria.Is65OrOlderCategoryTimeLine=65 or older
```

2 Configuring Health Care Reform

Complete some or all of the following tasks to configure your Health Care Reform implementation.

2.1 Configuring appeal requests

If you plan to use Cúram Appeals with HCR, you must ensure that application programs of any status can be appealed.

Procedure

1. Log in to the Cúram Administration application as a user with administrator permissions.
2. Select **Universal Access** > **Application Cases**.
3. Select the application case name.
4. In the **Appeals Processing** section, set the **Appeal All Programs** check box to true.

Appeal case types and appeals process configuration

If you are licensed for the Appeals application module, you can install Appeals with HCR. By default in the HCR application, each of the HCR case types are appealable and the appeals process is configured.

By default, the 'appealable' indicator is set to true on all of the HCR case types. Caseworkers can appeal any denied application, product delivery case or product delivery determination, regardless of its status.

By default, the Appeals process is set up as follows:

- 1. Stage 1 = 'Any'

  - The HCR Evidentiary Hearing, which is normally the first stage in the process, maps to the Hearing type. However, it is possible that the HHS Appeal (which maps to a Hearing Review) can happen first, so to support this, the type for stage 1 should be 'Any'.
  2. Stage 2 = 'Hearing Review'

  - This stage maps to the HHS Appeal as described in the federal rules.
  3. Stage 3 = 'Judicial Review'

  - This stage maps to the Judicial Review as described in the federal rules.

2.2 Configuring the resilient option for the process intake application workflow

After you install version 6.0.5.5 or later, ensure that you set the resilient option for the process intake application workflow, which enables a more granular workflow with better error handling.

Procedure

1. Log in to the Cúram Administration application as a user with Administrator privileges.

2. Set the curam.intake.use.resilience configuration property to true.

Related concepts
HCR application intake process overview on page 173
Use this information to understand how HCR applications are processed, from the submission of an application to the creation of product delivery cases.

2.3 Configuring rules to determine the coverage period start date

You can configure how the effective date of a change of circumstance is used to determine the coverage period start date.

The `curam.hcr.effectivedate` environmental property indicates whether to use the Evidence Effective Date to indicate when the reported change takes effect. It determines whether the change of circumstance takes effect based on the date the change happened (Effective Date) or based on when the evidence change was reported (Date Received). The property is used for all programs.

When the property value is set to the default value, `YES`, the coverage period start date is based on the received date of the applicable change in evidence, and the effective date of change of the evidence is not used. When the property value is set to `NO`, the coverage period start date is based on the effective date of change for the applicable change in evidence, and the received date is not used.

2.4 Configuring the Eligibility Viewer

The Eligibility Viewer is an enhancement to caseworker functionality that gives caseworkers a holistic view of eligibility for either a person or an integrated case. System administrators can customize the products that are displayed and the appearance of the eligibility viewer. System administrators can also configure the key events messages that are displayed by the eligibility viewer.

The product implementations are based on Java and might read the display rules for additional information to display to the user. If you update the display rules, you might also need to update the product implementations.

Configuring key events

Key events are changes to an integrated case that might affect eligibility. Key changes can be based on either evidence-based changes or non-evidence-based changes. An icon is displayed on the Eligibility Viewer below the month in which the key event occurs. Caseworkers can review key events to quickly determine what has changed on the case and the reason for a change in eligibility.

For information about configuring key events, see the related link.

Related tasks

Configuring a product for the Eligibility Viewer

The Eligibility Viewer has been configured to display eligibility information for all products that are in the `Insurance Affordability (CT26301)` integrated case. You can either configure a new product to be displayed on the Eligibility Viewer, or replace an existing product. You can also change the color that is displayed for a product on the Eligibility Viewer.

About this task

A data retrieval class has been implemented for each product in the `Insurance Affordability (CT26301)` integrated case. The implementation is configured based on a `PRODUCTTYPE` code table value, for example, `PT26304=Insurance Assistance`. Colors in the Eligibility Viewer are configured based on `PRODUCTTYPE`.

To add or replace a product implementation, or change the color that is displayed for a product in the Eligibility Viewer, use the following procedure. For more information, see the related link.

> **Note:** System administrators must maintain the performance and scalability of any custom bindings that they add.

Procedure

- To either add or replace a product implementation, use the following substeps:

  a) Create a class that implements the following:

  ```
  curam.hcr.eligibilitytimeline.productdataretriever.impl.EligibilityProductDataRetriever
  ```

  b) Configure Guice to use your implementation for the needed product, as shown in the following examples:

    - To set up a product data retrieval map, which is keyed by product type, use the following sample code:

    ```
    final MapBinder<String, EligibilityProductDataRetriever>
    eligibilityProductDataRetrieverMap =
        MapBinder.newMapBinder(binder(), String.class,
          EligibilityProductDataRetriever.class);
    ```

- To replace insurance assistance, use the following sample code:

```
eligibilityProductDataRetrieverMap.addBinding(
    PRODUCTTYPE.INSURANCEASSISTANCE).to(
  InsuranceAssistanceProductDataRetriever.class);
```

- To change the color that is displayed for a product in the Eligibility Viewer, update the application resource `TimelineCalendar.properties` and specify the hex color code for the product that you want to change.

Related tasks

Configuring look back and look forward periods

In a default installation, all Health Care Reform data retrieval implementations are configured to look back and look forward a maximum period of one year from the year that is displayed in the viewer. Limiting the look back and look forward periods improves the performance when eligibility information is retrieved for cases that extend over a long period time.

About this task

System administrators can configure the look back and look forward periods that determine the periods for which data is returned and displayed in the Eligibility Viewer. For more information about configuring the look back and look forward periods, see the related link.

> **Note:** The look back and forward periods that you configure also affect Income Support products, including Traditional Medicaid, Food Assistance, and Cash Assistance.

Procedure

To configure the look back and forward periods, update the following properties in the `TimelineCalendar.properties` application resource.

```
num.years.to.look.forward
num.years.to.look.back
```

Related tasks

2.5 Configuring primary client display settings for Health Care Reform

Organizations can configure application properties to hide references to primary client on various pages.

Use the following application properties to configure whether references to primary client are removed from display on various pages.

Customizing the default implementation

Display Case Clients on Case List Pages

Application property	Description
Property	`primaryclient.display.caselistpages.caseclients`
Default value	**False**
Description	Use the property to hide the Primary Client column and display a Client column in the following clusters: • My Recently Assigned Cases • My Recently Approved Cases • Recently Viewed Cases • My Cases • New Case Query results • Case Search results • Saved Case Query results When the Clients column is configured to be displayed, the following case members are listed: • All active case members with a case participant role of 'Member' or 'Primary Client', including those with an end date, for Insurance Affordability integrated cases and application cases. • All case members that are active members of the Case Group of type 'Member', including those with an end date, for Insurance Affordability product delivery cases. **Note:** Case member names are sorted in alphabetical order and the case type determines the case members displayed. Case member names do not include a hyperlink. If the value is set to **True**, multiple case members are displayed.

Display PD Case Clients on IC Home Pages

Application property	Description
Property	`primaryclient.display.integratedcasehomepage.caseclients`
Default value	**False**
Description	Determines whether: • Multiple product delivery case members are displayed in lists of product delivery cases that are displayed on the Insurance Affordability integrated case home pages OR • Only the primary client of the child product delivery cases is displayed. Use the property to perform the following action: • Remove the Primary Client column and replace it with a Clients column in the cases cluster. If the value is set to **True**, all case members of the product delivery case are displayed.

Display Participant Role on Context Panel

Application property	Description
Property	`primaryclient.display.contextpanel.displayroles`
Default value	**True**

Application property	Description
Description	Determines whether a participant's role, for example, a role of primary client or spouse, is displayed in the integrated case Context panel in Insurance Affordability integrated cases. Use the property to hide the following fields within the context panels, including within the context panels that are displayed in expanded views of lists of cases: • The Role in the Context panel of an Insurance Affordability integrated case. • The Relationship column in the Insurance Affordability integrated case Context panel when the List view is selected. If the value is set to **True**, the integrated case Context panels display the participant's role.

Display Primary Client Role Type in Case Search Case Participants Modal

Application property	Description
Property	`primaryclient.display.casesearch.caseparticipantmodal.roletype`
Default value	**True**
Description	Determines whether: • A role of primary client is displayed for the primary client of a case when the caseworker is viewing a list of case participants from the Participants column within the Case Search results cluster OR • A role of member is displayed instead of a role of primary client. If the value is set to **True**, a role of primary client is displayed. This property is associated with all of the pages that are listed in `primaryclient.display.caselistpages.caseclients`.

Hide Primary Client Name on Insurance Affordability Integrated Case Tab and Tab Header

Application property	Description
Property	`primaryclient.integratedcase.display.person.name.tab.title.header`
Default value	**False**
Description	Determines whether the name of the primary client is displayed in the tab title and tab header of the Insurance Affordability integrated case. If the value set to **True**, the name of the primary client is not displayed.

Display Primary Client on various Integrated Case pages

Application property	Description
Property	`primaryclient.display.integratedcase`
Default value	**True**

Application property	Description
Description	Determines whether references to the role of primary client are displayed on various Integrated Case pages. Use the property to perform the following action: • Change the display of the role of primary client to member on the Case Participants list page within the Insurance Affordability Integrated Case. If the value set to **True**, the role of primary client is displayed on the various Integrated Case pages.

Display Primary Client on various Person pages

Application property	Description
Property	`primaryclient.display.person`
Default value	**True**
Description	Determines whether visual references to the role of primary client are displayed on various **Person** pages. Use the property to perform the following action: • Change the display of the role of primary client to member on the **Cases** page under the **Care and Protection** tab on the Person record. If the value is set to **True**, the role of primary client is displayed on various Person pages.

Enabling hook points

If the application property `primaryclient.display.caselistpages.caseclients` is set to **True**, you can use the hook point `curam.core.hook.impl.CaseClientsPopulationHook` to customize the list of clients to display in the following case list pages:

• My Recently Assigned Cases.
• My Recently Approved Cases.
• My Recently Viewed Cases.
• My Cases.
• New Case Query results.
• Case Searches results.
• Saved Cases results.

You can enable the hook point through the standard Guice dependency injection mechanism.

When you implement `curam.core.hook.impl.CaseClientsPopulationHook`, you can customize per case type to return:

• The `CaseParticipantRoles` of type 'Member' or 'Primary Client'
• The members of the `CaseGroups` of type 'Member' that are associated with the case.

For Heath Care Reform, the following defaults apply:

• `CaseParticipantRoles` of type 'Member' or 'Primary Client' are returned for integrated case types and application cases.

- `Members CaseGroups` of type 'Member' are returned for the product delivery case types.

> **Note:**
>
> You can override the default CaseClientsPopulationHook implementation by creating a custom class that extends the ISCaseClientsPopulationHookImpl class. Bind the custom class to the custom class in a Guice module.

Configuring Smart Navigator for Health Care Reform

Smart Navigator allows caseworkers to quickly navigate to predefined persons, pages, or tabs. Income Support for Medical Assistance has been configured to use for Smart Navigator.

Organizations can enable or disable Smart Navigator for each application role and also customize your own search targets and keywords.

Refer to Using Cúram Smart Navigator to learn more about how caseworkers can perform searches using the Smart Navigator and Customizing Cúram Smart Navigator for information about how to enable and customize additional search targets and keywords.

3 Customizing the Health Care Reform portal

You can customize the Health Care Reform portal, which consists of a custom Citizen Portal for Health Care Reform.

3.1 Customizing the Health Care Reform Motivations

The Health Care Reform portal uses the Merative ™ Cúram Universal Access Motivation infrastructure for the online application processes required by ACA legislation.

Each Health Care Reform motivation is associated with an IEG script, a data store schema, and a display rule set. The following Health Care Reform motivations are available by default:

- Find Assistance
- Browse for plans
- Quick Shopping
- Employer Sponsored Coverage
- Apply for an exemption

For more information about motivations, see the *Merative ™ Cúram Universal AccessCustomization Guide*.

IEG scripts customization

The default Health Care Reform portal IEG scripts are in the HCROnline component. You can customize the default IEG scripts by creating a custom copy.

For more information about customizing IEG scripts, see the *Authoring Scripts Using Intelligent Evidence Gathering (IEG)* guide.

Eligibility Display Rules customization

When an IEG script completes, the eligibility results page is displayed according to the eligibility results display rules. You can write custom display rules to customize eligibility calculations for the eligibility results page. In addition, Health Care Reform provides several other mechanisms for customizing rule sets.

The default display rules reference the default eligibility rule sets to determine eligibility. The `curam.healthcare.eligibility.ruleset.name` property points to the name of this rule set. You must update this property if custom eligibility rules are to be used.

For information about configuring properties, see "Configuring Application Properties" in the *Cúram System Configuration Guide*.

For information about customizing rule sets in a compliant manner, see the *Curam Express Rules Reference Manual* and the *Cúram Development Compliancy Guide*.

There are several areas in the script rules where you can provide a custom implementation as follows.

Customizing the conditional display of IRS income information

You can customize the eligibility rules that determine the display of retrieved income from the IRS.

About this task

IRS income data that is retrieved for members in a tax household is not displayed if any of the following conditions are true:

- There is more than one financial household within the overall household.
- There are any American Indians or Alaskan Natives in the household.
- The household income is below the Medicaid or CHIP threshold for any of the applicants in the household.

These rules are implemented by the *IRSIncomeDisplayDeterminator* rule class available in the default *HealthCareReformEligibilityRuleset* rule set.

Procedure

1. Create a custom rule class that adheres to the default structure provided in the Abstract Eligibility rule set, *AbstractEligibilityRuleset.IRSIncomeDisplayDeterminator*

   This custom rule class must ultimately extend the *AbstractEligibilityRuleset.DefaultIRSIncomeDisplayDeterminator* rule class.

2. Update the *curam.healthcare.displayirsincome.invoking.ruleclass.name* property to point to the fully qualified name of the custom rule class.
   For example, *MyRuleSet.MyRuleClass.*

Customizing the conditional display of specific questions for Medicaid, CHIP, or IA

You can override the default eligibility rules that determine which specific questions are asked based on eligibility for Medicaid, CHIP, or IA.

About this task

Certain eligibility rules are run as the citizen progresses through the script. These rules control the flow of the script according to the citizen's eligibility for certain programs. When the user enters income information for the household, these rules run. The results of these rules allow the script to ask intelligent questions pertinent to the program for which a household member is considered eligible.

Procedure

1. Create a custom rule class that adheres to the default structure provided in the *AbstractEligibilityRuleset.EligibilityDeterminationCalculator* rule class.

    This custom rule class must ultimately extend the *AbstractEligibilityRuleset.DefaultEligibilityDetermination* rule class.

2. Update the *curam.healthcare.eligibility.invoking.ruleclass.name* property to point to the fully qualified name of the custom rule class.
    For example, *MyRuleSet.MyRuleClass*.

Customizing the determination of projected annual income for a citizen

You can override the default eligibility rules that determine the projected annual income for a client.

About this task

Income calculation rules are run after you capture a household member's complete income details, including any deductions or exclusions. The projected annual income is then calculated by rules that are based on these details. The citizen can choose to attest to the determined projected annual income or chose to enter a different value. If the customer enters a different value, the rules take this value into consideration for calculating final eligibility.

Projected annual income is determined by invoking MemberIncomeCalculator available in the default HealthCareReformEligibilityRuleset. The property is "curam.healthcare.memberincome.invoking.ruleclass.name" and is set to a default implementation of the *HealthCareReformEligibilityRuleset.MemberIncomeCalculator* rule class.

Procedure

1. Create a custom rule class that adheres to the default structure provided in the Abstract Eligibility rule set *AbstractEligibilityRuleset.MemberIncomeCalculator*.

    This custom rule class must ultimately extend the *AbstractEligibilityRuleset.DefaultMemberIncomeCalculator* rule class.

2. Update the *curam.healthcare.memberincome.invoking.ruleclass.name* property to point to the fully qualified name of the custom rule class.
    For example, *MyRuleSet.MyRuleClass*.

3.2 Moving the Log Out button to improve usability

The Log Out button from the Welcome Person menu is now available in the Universal Access mega-menu for improved accessibility. For solutions that use a custom Citizen Portal, such

as Cúram Income Support for Medical Assistance (Health Care Reform), you can manually implement this improvement.

About this task

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

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

Procedure

1. Edit the custom *CITWSAPP.app* file for the new Log Out button and enter the following code:

```
<ac:banner-menu type="logout" title="logout.title" />
```

2. Edit the custom *CITWSAPP.properties* file for the new Log Out button and enter any required properties.

3.3 Enabling the Notices tab

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

About this task

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

Procedure

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

```
<nc:navigation-page id="mycommunications"
    page-id="CitizenAccount_communications"  title="leaf.title.communications"
    icon="CitizenAccount.communications.notices.leftnav.icon"
    description="CitizenAccount.desc"/>
```

2. Edit the custom *CitizenAccount.properties* file and enter any required properties.

4 Integration with external systems

The Health Care Reform solution can call external systems at certain points to gather information necessary for application processing. For example, a call can be made to the Federal Hub to verify SSN and citizenship status for a citizen.

The customization and configuration options for these integration points are as follows:

4.1 Customizing the external system implementations

By default, Health Care Reform provides several interfaces and corresponding implementations for integrating with external systems. Customers are free to provide their own implementations for these integration points.

About this task

> **Note:** You might want to customize the default Federal Hub implementation by using the provided customization points.

The following table lists the default external system interfaces, default implementations, and Federal Hub implementations.

Interface	Default Implementation	Federal Hub Implementation
curam.hcr.verification.service.impl. SSACompositeBusinessService	curam.hcr.verification.service.impl. SSAVerificationServiceImpl	curam.hcr.verification.service.impl. FederalSSACompositeServiceImpl
curam.hcr.verification.service.impl. AnnualIncomeDataService	curam.hcr.verification.service.impl. AnnualIncomeDataServiceImpl	curam.hcr.verification.service.impl. FederalAnnualIncomeVerificationServiceImpl
curam.hcr.verification.service.impl. IRSHouseholdDataService	curam.hcr.verification.service.impl. IRSHouseholdDataServiceImpl	No service available
curam.hcr.verification.service.impl. LawfulPresenceVerificationService	curam.hcr.verification.service.impl. LawfulPresenceVerificationServImpl	curam.hcr.verification.service.impl. FederalLawfulPresenceVerificationServiceImpl
curam.hcr.verification.service.impl. MECVerificationService	curam.hcr.verification.service.impl. MECVerificationServiceImpl	curam.hcr.verification.service.impl. FederalMECVerificationServiceImpl
curam.hcr.verification.service.impl. ResidencyVerificationService	curam.hcr.verification.service.impl. ResidencyVerificationServiceImpl	No service available
curam.hcr.verification.service.impl. IncomeDataService	curam.hcr.verification.service.impl. IncomeDataServiceImpl	curam.hcr.verification.service.impl. FederalCurrentIncomeVerificationServiceImpl
curam.hcr.verification.service.impl. CloseDHSCaseService	curam.hcr.verification.service.impl. CloseDHSCaseServiceImpl	curam.hcr.verification.service.impl. FederalCloseDHSCaseService
curam.hcr.verification.service.impl. ESIVerificationService	curam.hcr.verification.service.impl. ESIVerificationServiceImpl	curam.hcr.verification.service.impl. FederalESIVerificationServiceImpl
curam.hcr.verification.service.ridp. fars.impl.FARSVerificationService	curam.hcr.verification.service.ridp. fars.impl.FARSVerificationService	curam.hcr.verification.service.ridp. fars.impl.FederalFARSServiceImpl
curam.hcr.verification.service.ridp. primary.impl. RIDPPrimaryRequestVerificationService	curam.hcr.verification.service.ridp. primary.impl. RIDPPrimaryRequestVerificationService	curam.hcr.verification.service.ridp. primary.impl. FederalRIDPPrimaryRequestServiceImpl

Interface	Default Implementation	Federal Hub Implementation
curam.hcr.verification.service.ridp. secondary.impl. RIDPSecondaryRequestVerificationService	curam.hcr.verification.service.ridp. secondary.impl. RIDPSecondaryRequestVerificationServiceImpl	curam.hcr.verification.service.ridp. secondary.impl. FederalRIDPSecondaryRequestServiceImpl

Procedure

1. To create a custom implementation, write a new class that extends one of the external system default implementations.

2. Bind the custom implementation to the corresponding interface by using a Guice module. For example:

```
public class CustomModule extends AbstractModule {
@Override
protected void configure() {

 binder().bind(IncomeDataService.class).to(CustomIncomeDataService.class);
   }

}
```

3. Ensure that the module is added to a custom Module Class Name *.DMX* file. For example:

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

Customizing request or response fields for external system calls

You can customize the request and response fields that are used by the external system interfaces by extending the respective request or response classes. You can then use the updated request or response classes in the custom implementation of the external system interface.

Procedure

1. Extend the request or response classes. For example:

```
CustomCitizenshipVerificationRequestDetails
extends CitizenshipVerificationRequestDetails {
//Define custom attributes
//Define getter and Setter methods
}
CustomCitizenshipVerificationResponseDetails
extends CitizenshipVerificationResponseDetails {
      //Define custom attributes
      //Define custom getter and setter methods
}
```

2. Use the updated request or response classes in the custom implementation of the external system interface.

For example:

```
CustomCitizenshipVerificationServiceImpl
implements CitizenshipVerificationService {

CustomCitizenshipVerificationResponseDetails
verify(CustomCitizenshipVerificationRequestDetails requestDetails){
}
}
```

4.2 External system processors

During the application process, external system Java classes that are called processors callout to external systems and store the information received in the data store. For example, a call is made to the Federal Hub to verify SSN and citizenship by using the CombinedSSAServiceViewProcessor processor. The response is stored in the data store by the processor and can be used later to facilitate the electronic verification process. By default, the following processors are available:

- `curam.hcr.verification.datastore.impl.CombinedSSAServiceViewProcessor`
- `curam.hcr.verification.datastore.impl.AnnualIncomeViewProcessor`
- `curam.hcr.verification.datastore.impl.CurrentIncomeViewProcessor`
- `curam.hcr.verification.datastore.impl.LawfulPresenceViewProcessor`
- `curam.hcr.verification.datastore.impl.MECViewProcessor`
- `curam.hcr.verification.datastore.impl.RIDPFARSViewProcessor`
- `curam.hcr.verification.datastore.impl.RIDPPrimaryViewProcessor`
- `curam.hcr.verification.datastore.impl.RIDPSecondaryViewProcessor`

4.3 Configuring the Federal Hub implementation

By default, all external system calls are routed to the default (empty) implementations for the external system interfaces. Complete the following steps to route the external system calls to the Federal Hub implementations. You must restart the server after you update these values.

About this task

For information about configuring properties, see "Configuring Application Properties" in the *Cúram System Configuration Guide*.

Procedure

1. Set the *curam.healthcare.test.registerMockExternalSystems* property to false.
2. Set the *curam.fed.hub.verification.system.name* property to the Federal Hub system name.
3. Set the *curam.fed.hub.verification.system.registered* property to true.
4. Restart the server.

4.4 Configuring a State systems implementation

You might want to implement a custom implementation to call State systems as well as calling the Federal Hub.

About this task

For example, you might want to retrieve Current Income from the State Quarterly Wages system, and to fall back on the corresponding Federal Hub service only if the information is not available.

Procedure

Create a custom implementation for the service that first calls the State system, and then calls the Federal Hub implementation.

4.5 Customizing electronic verifications

External systems are also used for electronic verification of information that is provided in the application. Health Care Reform provides support for integrating with external systems such as state systems or third-party commercial applications that are identified by states as data sources. You can also customize electronic verification.

By default, Health Care Reform provides processing for Electronic Verification of data such as Citizenship, Residency, or SSN. The framework for Electronic Verification supports adding implementations for custom verification processing for data elements that are either not covered by default processing or those data elements that are added as part of the custom implementation. Also, it is possible to override the default Verification Processing, if needed.

Default verification processors

Seven verification processors are available by default.

- **The seven default verification processors explained**

  The following list outlines the verification processors that are available by default:

  - **curam.hcr.verification.online.impl.ResidencyVerificationProcessor**
    The verification processor considers that the Residency is verified where it was indicated (isStateResident attribute of the Person data store entity or has address with the state to be configured state) that a Person was a state resident or where the Person was indicated as a state resident and the information that is retrieved about the person from the external system, which is stored in the ExternalSystemResidencyInformation data store entity, also indicates that the person is a state resident.
    The processing is completed for all the persons who are marked as applicant (isApplicant attribute of the Person data store entity) on the case.
  - **curam.hcr.verification.online.impl.CitizenshipVerificationProcessor**
    The verification processor considers that the Citizenship is verified where it was indicated (isUSCitizen or isUSNational or lawfullyPresent attribute of the Person data store

entity) that a Person was a US citizen or US Nation or Lawfully Present alien and the information that is retrieved about the person from the external system, which is stored in the ExternalSystemCitizenshipInformation data store entity, also indicates that the person citizenship is verified.

The processing is completed for all the persons who are marked as applicant (isApplicant attribute of the Person data store entity) on the case.

- **curam.hcr.verification.online.impl.IncarcerationVerificationProcessor**
The verification processor considers that the Incarceration status is verified where it was indicated (isIncarcerated attribute of the Person data store entity) that a Person is incarcerated or where the Person was indicated as not incarcerated or incarcerated pending disposition and the information that is retrieved about the person from the external system, which is stored in the RetrievedPersonInformation data store entity, indicates the same.
The processing is completed for all the persons who are marked as applicant (isApplicant attribute of the Person data store entity) on the case.

- **curam.hcr.verification.online.impl.HouseholdSSNVerificationProcessor**
The verification process considers that the SSN is verified where the SSN was provided (ssn attribute of the Person data store entity) and the information that is retrieved about the person from the external system, which is stored in the ExternalSystemSSNInformation data store entity, indicates that the SSN was verified.
This processing is completed for all the persons who are marked as applicant (isApplicant attribute of the Person data store entity) on the case.

- **curam.hcr.verification.online.impl.IncomeVerificationProcessor**
The verification processor considers that the Income data is verified where the Income was provided (IncomeItem data store entity has records) and the information that is retrieved about the person from the external system, which is stored in the IRSAnnualTaxReturn or ExternalSystemIncome data store entity, are reasonably compatible or are E-verified.
The processing is completed for all the persons.

- **curam.hcr.verification.online.impl.MECVerificationProcessor**
The verification processor considers that the MEC is verified where the person indicated to not receive benefits (isReceivingBenefits attribute of the Person data store entity) and the information that is retrieved about the person from the external system, which is stored in the ExternalSystemMECDetails data store entity, indicates the same.
The processing is completed for all the persons.

- **curam.hcr.verification.online.impl.ESIVerificationProcessor**
The verification processor considers that the ESI status is verified where the person indicated that they have Employer Sponsored Insurance and the information that is retrieved from the external system is electronically verified.
The processing is completed for all the persons.

Adding custom verification processing

Complete the following steps to add custom verification processing.

Procedure

1. Edit *CT_VerificationItemType.ctx* to add an entry to the VerificationItemType code table.

2. Create an implementation of the
   curam.hcr.verification.online.impl.VerificationProcessorinterface. Ensure that the
   `getVerificationType()` API returns the code table code you added.

3. Install the custom implementation by using a custom Guice module. The custom Verification
   Processing implementation can be bound by using a Guice Set MultiBinder.
   For example:

```
public class CustomModule extends AbstractModule {
  @Override
  protected void configure() {
    Multibinder<VerificationProcessor> binder = Multibinder.newSetBinder(
      binder(), VerificationProcessor.class);
      binder.addBinding().to(CustomVerificationProcessor.class);
  }
}
```

4. Add an entry that contains the custom Guice module name to a `.DMX` file for
   ModuleClassName entity.

Overriding the default verification processing

Complete the following steps to override the default verification processing.

About this task

Each entry in the VerificationItemType represents a kind of data item, such as Citizenship.

Procedure

1. Review CT_VerificationItemType.ctx to identify the code for the data item type for which the
   default processing must be overridden.

2. Create an implementation of the
   curam.hcr.verification.online.impl.VerificationProcessorinterface. Ensure that the
   `getVerificationType()` API returns the code table code you identified.

3. Install the custom implementation by using a custom Guice module. The custom Verification
   Processing implementation can be bound by using a Guice Set MultiBinder.
   For example:

```
public class CustomModule extends AbstractModule {
  @Override
  protected void configure() {
    Multibinder<VerificationProcessor> binder = Multibinder.newSetBinder(
      binder(), VerificationProcessor.class);
      binder.addBinding().to(CustomVerificationProcessor.class);
  }
}
```

4. Add an entry that contains the custom Guice module name to a `.DMX` file for
   ModuleClassName entity.

4.6 Supported federal hub verification services

HCR supports a number of federal hub verification services that are used to verify evidence. Only specific versions of these services are supported.

Table 2: Supported federal hub verification services

Federal hub verification service	Supported version
Remote ID Proofing	Sprint 14
SSA Composite	Sprint 15
Verify Lawful Presence (VLP)	Sprint 15
Verify Employer Sponsored Insurance (ESI)	Sprint 14
Verify non-ESI Minimum Essential Coverage	Sprint 15
Verify Current Income	Sprint 14
Verify Annual Income	Sprint 15

5 Customizing case management

You can customize Health Care Reform case management artifacts such as dynamic evidence, eligibility rule sets, and conditional verifications.

5.1 Dynamic evidence customization

Health Care Reform provides a number of dynamic evidence configurations in the HCR component. The Health Care Reform dynamic evidence configurations model information that is captured and maintained for the various ACA programs.

For information about customizing dynamic evidence, see the *Cúram Dynamic Evidence Configuration Guide*.

5.2 Eligibility rules customization

Health Care Reform provides a default set of eligibility rule sets in the HCR component. You can customize these eligibility rules for your custom requirements.

For information about customizing eligibility rules, see the *Inside Cúram Eligibility and Entitlement Using Cúram Express Rules guide*.

For information about compliantly customizing the default rule sets, see the *Cúram Development Compliancy Guide*.

Customizing non-financial rules failure reasons

You can customize non-financial rules failure reasons by updating the relevant failure reason in the *CT_HCIneligibleReason.ctx* code table file.

About this task
Changing non-financial display rule descriptions does not affect the eligibility rule interface contract, and thus does not require any changes to the actual display rules. Failure reasons are picked up automatically.

Procedure

1. Open *EJBServer\components\HCR\codetable \CT_HCIneligibleReason.ctx*
2. Modify the non-financial failure reason description for the failure reason you want to change.

5.3 Conditional verifications customization

Health Care Reform application cases and integrated cases are configured to use the verification framework. You can customize conditional verification rule sets in the same way as other rule sets.

For more information about configuration of verifications and conditional verifications, see the *Cúram Verification Guide*.

For information about compliantly customizing the default rule sets, see the *Cúram Development Compliancy Guide*.

6 Customizing plan management

Complete the following tasks to customize the default plan management implementation.

6.1 Integration with Plan Management

When a citizen applies for insurance affordability assistance through Cúram, they must go to a plan management vendor's website to view and purchase plans. To facilitate this access, you must integrate a plan management vendor with the Cúram application. You can integrate with the plan management vendor of your choice.

> **Important:** Cúram implements a vendor-agnostic approach to plan management integration and does not include an implementation of the plan management adapter in the product. Each project is responsible for implementing their own integration between the Cúram system and the plan management system of choice.

Plan management integration is accomplished with a combination of both user interface and web services integration.

A plan management vendor's user interface is shown in an inline frame on a Cúram page.

Information is exchanged between Cúram and the plan management vendor through two categories of web services:

- Web services that are owned by Cúram (inbound)
- Web services that are owned by the plan management vendor (outbound)

This approach allows the citizen to enroll on a plan on the plan management vendor's system with the eligibility information that is determined on the Cúram side. In addition, Cúram can query the plan management vendor's web services to read and store any plans in which a citizen enrolls.

6.2 The plan management adapter interface

A plan management interface is provided which customers must implement. The custom implementation allows customers to communicate with their chosen plan management vendor through web services.

The methods in the interface are called at different points during processing. For example, the getEnrollmentDetails() method is called to determine the plan details after a citizen successfully enrolls on a plan in the plan management system.

A default
curam.planmanagement.adapter.impl.PlanManagementAdapterDefault
implementation of the plan management adapter interface is provided. To provide some insulation from future changes, extend this class instead of directly implementing the interface.

`curam.planmanagement.adapter.impl.PlanManagementAdapter`

- `getBenchmarkPlanDetails()`

  Retrieves the benchmark plan amount and essential health benefit premium amount from a plan management vendor.

- `getEnrollmentDetails()`

  Retrieves the enrollment details for a completed enrollment. For example, the enrolled plan details. You can customize the Enrollment evidence mapping by using the following mechanisms:

  - Provide a custom implementation of EnrollmentEventProcessor, and optionally EnrollmentSanitizer
  - Override the event ENROLLMENT.ENROLLMENT_ADDED

- `getAvailableEmployerPlanDetails()`

  Retrieves the available employer insurance plans for an employee.

- `getBenchmarkPlanDetailsForBenefitMembers()`

  Retrieves the benchmark plan amount and essential health benefit premium amount from a plan management vendor.

- `updateEntitlementDetails()`

  Informs the plan management vendor of a change in entitlement for a specific enrollment.

- `getPlanUpdates()`

  Retrieves any updates to plans for an enrollment, typically called during re-enrollment.

- `continueEnrollment()`

  Informs the plan management vendor that an existing enrollment on a plan is to be continued, typically called during the re-enrollment period.

- `getPolicyID()`

  Retrieves the policy identifier for a specific enrollment.

- `getEmployerOpenEnrollmentDetails()`

  Retrieves the open enrollment details for an employer.

> **Note:** For more information about the plan management adapter interface, see the Javadoc in the *HCR* component.

For more information, see "Events and Event Handlers" in the *Server Developer's Guide*.

Configuring the plan management adapter

The custom plan management adapter typically communicates with a plan management vendor over a web service with stubs generated from the plan management vendor's WSDL file.

Procedure

1. Create a directory that is named axis in a custom component.

2. Add a `ws_outbound.xml` file to this directory. This file must reference the WSDL file that is provided by a plan management vendor. ,
   For example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<services>
<service
location="components/CustomComponent/axis/PlanMgmtWebService/
PlanManagementVendor.wsdl"
    name="PlanManagementVendor"
  />
</services>
```

3. From a command prompt under the *EJBServer* directory, run `build wsconnector2` to generate the stubs to the *build* directory.
   These stubs are now available to call in the custom PlanManagementAdapter implementation.

4. Create an implementation of the plan management adapter interface and bind it using a Guice module.
   For example:

```java
@Override
protected void configure() {

  bind(PlanManagementAdapter.class).to(CustomPlanManagementAdapter.class);
}
```

   For more information about bindings in Guice, see the *Persistence Cookbook*.

5. Code the custom implementation of the plan management adapter by using the generated stubs.

   For more information about web services in Cúram, see the *Cúram Web Services Guide*.

6.3 Plan management web services provided by Cúram

A plan management vendor must call Cúram web services to be able to populate their screens and for plan management processing.

For example, when a household is enrolling on a plan in the plan management vendor's system, the vendor requires details about the household such as names, date of births, address, and eligibility information. Cúram provides the retrieveDemographicsAndEligibilityDetails() web service for this purpose.

The following web services are provided:

`curam.planmanagement.adapter.intf.HealthCareWebService`

- `retrieveDemographicsAndEligibilityDetails()`
- `getHouseholdSummaryDetails()`
- `getEntitlementDetails()`
- `policyIDAvailable()`
- `updateEmployerEnrollment()`

For more information about these web services, see the Javadoc in the HCR component.

Related concepts

Health Care Reform web services on page 69

The web services that are available for Health Care Reform.

6.4 Configuration parameters for plan management

The following configuration properties exist for plan management integration.

Property	Description
curam.healthcare.planManagementVendorUrl	The plan management vendor URL for the main find assistance flow.
	A unique enrollment identifier is appended to this URL.
curam.healthcare.planManagementVendorBrowseForPlansUrl	The plan management URL used to allow a citizen to browse for (but not purchase) insurance plans.
	A unique enrollment identifier is appended to this URL.
curam.healthcare.planManagementVendorEmployerCoverageUrl	The plan management URL used to allow employees to shop for insurance plans provided by their employer.
	A unique enrollment identifier is appended to this URL.
curam.healthcare.planManagementVendorAvailable	This property indicates whether a plan management vendor is available. By default, it is set to false to enable testing but must be set to true when integrated with a plan management vendor.

6.5 Callback URLs for plan management

Callback URLs are the URLs that a plan management vendor uses to return control to the Cúram user interface. For example, after an enrollment completes, a callback URL is used to redirect back to the Cúram results page.

The default callback URLs are listed in this table.

Callback URL	Description
https://<host>:<port>/CitizenPortal/en_US/HealthCare_finishEnrollmentPage.do?o3ctx=4096	A plan management vendor redirects to this URL upon successful completion of an enrollment.
https://<host>:<port>/CitizenPortal/en_US/HealthCare_saveAndExitEnrollmentPage.do?o3ctx=4096	A plan management vendor redirects to this URL if a user chooses to save and exit from the plan management vendor's screens. This option would enable a user to resume the enrollment later.
https://<host>:<port>/CitizenPortal/en_US/HealthCare_cancelEnrollmentPage.do?o3ctx=4096	A plan management vendor redirects to this URL if a user chooses to cancel/quit from the plan management vendor's screens.

6.6 Batch processing for plan management

The following plan management batch processes are available.

For more information about batch processes, see the *Cúram Batch Processing Guide*.

Employer enrollment notification batch process

The purpose of this batch process is to generate notifications for employees to indicate that the open enrollment period for their employer is about to begin.

This batch process looks at active EmployerEnrollment records on the database. For each one, it calls out to the plan management vendor by using the curam.planmanagement.adapter.impl.PlanManagementAdapter.getEmployerOpenEnrollmentDetails() API. Using the response from the plan management vendor, a pro-forma communication is generated and stored against each employee returned.

6.7 Plan management web service API reference

The plan management web services that are available for the Cúram Income Support for Medical Assistance and the schema that is used for the data.

Health Care Reform web services

The web services that are available for Health Care Reform.
Related concepts
Plan management web services provided by Cúram on page 67
A plan management vendor must call Cúram web services to be able to populate their screens and for plan management processing.

retrieveDemographicsAndEligibilityDetails
A plan management vendor requests eligibility details for an enrollment. The eligibility details and details for each person in the enrollment are returned from Cúram Income Support for Medical Assistance.

Table 3: Request

Request

Data Member	Type	Description
EnrollmentDetails	EnrollmentDetails	The health care reform retrieve eligibility request that contains the enrollment ID.

Table 4: Response

Response

Data Member	Type	Description
EligibilityAnd DemographicDetails	EligibilityAnd DemographicDetails	Response containing eligibility details, details about each person in the enrollment group, previous enrollments for each person that is being enrolled and details about assistors.

getEntitlementDetails

A plan management vendor calls the Income Support for Medical Assistance (Health Care Reform) solution to get updated entitlement details for an existing enrollment.

Table 5: Request

Data Member	Type	Description
EnrollmentDetails	EnrollmentDetails	The health care reform retrieve eligibility request that contains the enrollment ID.

Table 6: Response

Data Member	Type	Description
EntitlementUpdateDetails	EntitlementUpdateDetails	Response containing the updated tax credit amount

getHouseholdSummaryDetails

A plan management vendor calls the Income Support for Medical Assistance (Health Care Reform) solution to notify any change in the status of an existing enrollment. For example, when a carrier finishes processing the enrollment and made a policy ID available.

Table 7: Request

Data Member	Type	Description
EnrollmentDetails	EnrollmentDetails	Contains the enrollment ID for which the request is being made

Table 8: Response

Data Member	Type	Description
HouseholdSummaryDetails	HouseholdSummaryDetails	Response containing eligibility details, details about each person in the enrollment group, previous enrollments for each person that is being enrolled and details about assistors.

policyIDAvailable

A plan management vendor calls the Income Support for Medical Assistance (Health Care Reform) solution to notify that a carrier has finished processing the enrollment and made a policy ID available.

Table 9: Request

Data Member	Type	Description
EnrollmentDetails	EnrollmentDetails	Contains the ID of the enrollment for which a policy ID is available.

updateEmployerEnrollment

A plan management vendor calls this API to notify the agency that the open enrollment period has begun for a specific employer.

Table 10: Request

Data Member	Type	Description
EmployerEnrollment	EmployerEnrollment	Contains the employerEnrollmentID for the employer with an open enrollment period.

Table 11: Response

Data Member	Type	Description
EmployerEnrollmentReceived	EmployerEnrollmentReceived	An indicator that represents successful receipt and storage of the employer identifier.

Health Care Reform schema elements

The schema that is used for Health Care Reform data.

Table 12: EnrollmentDetails

Data Member	Type	Description
enrollmentID	Long	The enrollment key.

Table 13: EligibilityAndDemographicDetails

Data Member	Type	Description
eligibilityDetails	eligibilityDetails	
persons	persons	
previousEnrollments	previousEnrollments	
assistors	assistors	
employerDetails	employerDetails	

Table 14: eligibilityDetails

Data Member	Type	Description
program	String	Values are as follows: EP1 Insurance Assistance EP2 CHIP EP3 Medicaid EP4 State Basic Plan EP5 None (for when the household is just shopping for plans)
maxPremiumTaxCredit	Double	
maxPremiumTaxCreditAnnual	Double	The amount of premium tax credit that remains for the year.
monthsRemaining	Int	The number of months that remains in the plan year.
costSharingSubsidy	Double	
premiumPayment	Double	
maximumCoPay	Double	
stateSubsidy	Double	
enrollmentPeriod	String	Values are as follows: EPD1 Open EPD2 Special
coverageStartDate	Date	
coverageEndDate	Date	

Table 15: persons

Data Member	Type	Description
person	List of person	

Table 16: person

Data Member	Type	Description
personID	Long	Unique identifier for a person within the exchange
ssn	String	
firstName	String	
middleName	String	
lastName	String	
dateOfBirth	Date	
gender	String	Values are as follows: SX1 Male SX2 Female
tobaccoUser	Boolean	
coverageCategory	String	Values are as follows: CC1 Parent/Caretaker CC2 Pregnant Woman CC3 Adult CC4 Child
address	Address	
phoneNumber	PhoneNumber	
emailAddress	String	
nativeAmerican	Boolean	Indicates whether the person is an American Indian or Alaskan Native.
isPrimaryContact	Boolean	Indicates whether the person is the primary contact for the group that is being enrolled
costSharingEliminated	Boolean	True for AI/NA individual with household income less than or equal to 300% of FPL
subscriberID	Long	Unique identifier of the primary client that is assigned to each member.
taxFilerRelationshipList	TaxFilerRelationshipList	

Table 17: Address

Data Member	Type	Description
addressLine1	String	
addressLine2	String	
city	String	
county	String	
state	String	

Data Member	Type	Description
zip	String	

Table 18: TaxFilerRelationshipList

Data Member	Type	Description
taxFilerRelationships	List of TaxFilerRelationship	

Table 19: TaxFilerRelationship

Data Member	Type	Description
relatedPersonID	Long	
taxFilerRelationshipType	String	Values are as follows:
		TFRT26001 Dependent
		TFRT26002 Spouse
		TFRT26003 Tax Filer

Table 20: previousEnrollments

Data Member	Type	Description
enrollment	List of enrollment objects	

Table 21: enrollment

Data Member	Type	Description
enrollmentID	Long	
planID	String	
policyID	String	
coverageEndDate	Date	
previousPremium	Double	
previousTaxCredit	Double	
previousEnrollees	previousEnrollees	

Table 22: previousEnrollees

Data Member	Type	Description
enrollee	List of enrollee objects	

Table 23: enrollee

Data Member	Type	Description
personID	Long	

Table 24: assistors

Data Member	Type	Description
assistor	List of assistor objects	

Table 25: assistor

Data Member	Type	Description
firstName	String	
lastName	String	
address	Address	
phoneNumber	PhoneNumber	
certificationNumber	String	
assistorType	String	
assistorID	Long	
agencyOrganisationID	Long	

Table 26: PhoneNumber

Data Member	Type	Description
countryCode	String	
areaCode	String	
phoneNumber	String	
Extension	String	

Table 27: EmployerDetails

Data Member	Type	Description
employerID	Long	
coverageStartDate	Date	

Table 28: EntitlementUpdateDetails

Data Member	Type	Description
enrollmentID	Long	
updatedPremiumTaxCredit	Double	

Table 29: HouseholdSummaryDetails

Data Member	Type	Description
effectiveDate	String	
zipCode	String	
personList	PersonList	

Table 30: PersonList

Data Member	Type	Description
persons	List of Person	

Table 31: person

Data Member	Type	Description
dateOfBirth	Date	
tobaccoUser	Boolean	
isPrimaryContact	Boolean	Indicates whether the person is the primary contact for the group being enrolled

Table 32: EmployerEnrollment

Data Member	Type	Description
employerEnrollmentID	String	The employer enrollment identifier.

Table 33: EmployerEnrollmentReceived

Data Member	Type	Description
employerEnrollmentReceived	Boolean	Indicates that the employer enrollment identifier was successfully received and stored.

7 Customizing change of circumstances

To customize change of circumstances for your environment, you must be familiar with the default implementation. Use this information to understand the process flow, and to identify the steps that you must complete to customize your system.

Related tasks

[Customizing inconsistency period processing on page 153](#)
Inconsistency period processing allows a caseworker to give a client a reasonable opportunity period to provide outstanding verifications for evidence that requires verification. Cases can proceed during that period as if outstanding verifications were provided. The default inconsistency period processing infrastructure consists of a batch process, a workflow, and the inconsistency period processing APIs. You can create a custom event handler to customize the default inconsistency period processing.

7.1 Change of Circumstances Process Flow

Use this information to understand how the components work together to handle changes in client circumstances.
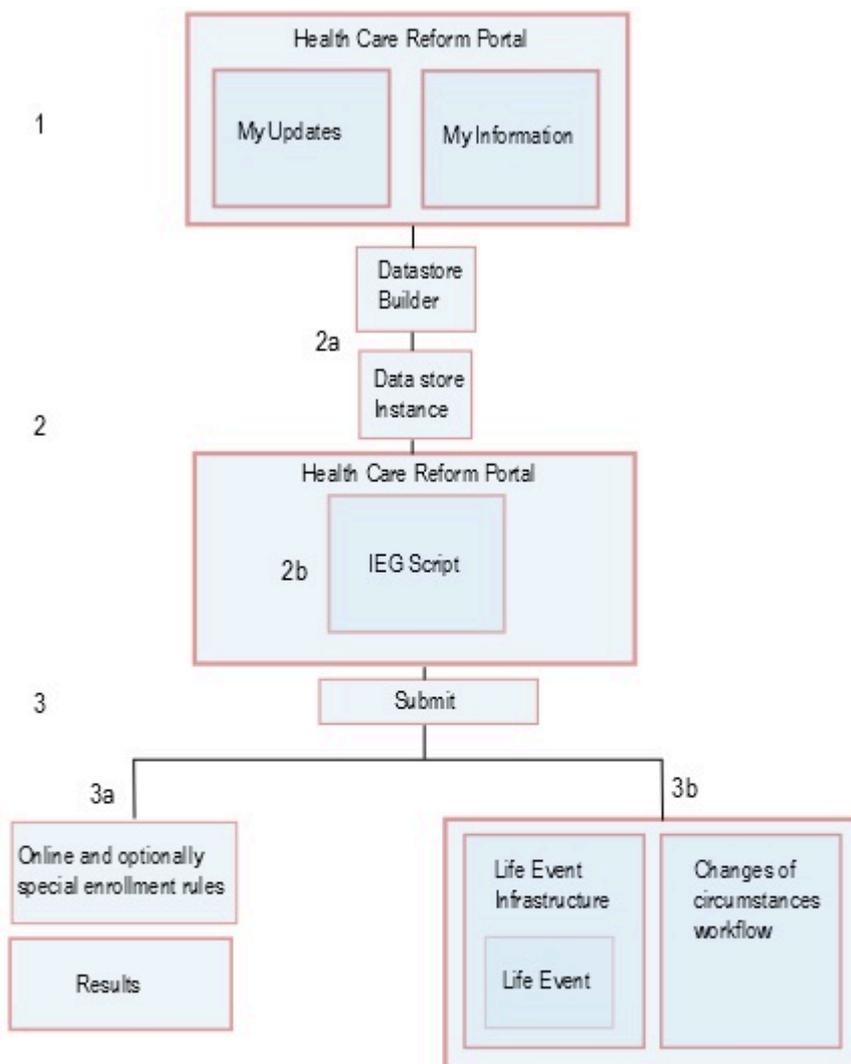
Figure 1: Change of circumstances process flow

- **1 A client with an existing application logs in to the Health Care Reform (HCR) portal**

  They can see a read-only summary of some of their evidence, such as Social Security number (SSN), Address, Household Members, and Income by clicking the **View your information to provide updates** link on the landing page, or the **My Information** menu option. They also can see the history of their submitted Life Events in **My Updates**.

The read-only data that is shown is the most current information for each evidence type, specifically the most recent active evidence. If the client recently created an in-edit version of evidence by a previous change of circumstances, that in-edit version is displayed instead.

- **2 The client decides to update their data**

After they review the information, the client can click the **Update My Information** link to update their information. This link is available only if no outstanding change of circumstances exist for the client. Clicking the link starts the following processing:

2a The change of circumstances Datastore Builder retrieves the evidence from the ongoing Insurance Affordability integrated case and creates a data store instance for the data retrieved. This data store instance becomes the data store used for the change of circumstances IEG script.

2b The change of circumstances intelligent evidence gathering (IEG) script opens with the data pre-populated for the client to make the required changes. The client can add, update, or remove data. Remove refers to end-dating particular evidence types. The client continues through the script and completes their updates.

- **3 The client submits their change of circumstances updates**

When the client clicks submit, the following processing starts:

- **3a Online and special enrollment rules**
  HCR Online rules, and optionally special enrollment rules, are run to generate a results page for the client. Clients can enroll only on Advanced Premium Tax Credit (APTC) plans outside the configured open enrollment period if they meet the special enrollment criteria. A set of special enrollment rules are run to determine whether the reported change qualifies an individual for special enrollment.

  A results page with the outcome of those rules is displayed to the client. Depending on the results, the client can proceed to enrollment.

- **3b Life Event Infrastructure**

  The change of circumstances process uses the Life Event infrastructure as the mechanism for updating the ongoing Insurance Affordability case with the new or modified data that is supplied by the client. When the life event associated with the change of circumstances is submitted, the following processing is triggered:

  1. The state of the life event is updated to pending. The **Update My Information** link is unavailable to prevent the client from triggering a second change of circumstances while there is still one in progress.
  2. The change of circumstances workflow is started.

Change of Circumstances workflow

View the default Change of Circumstances workflow in the **Workflow** section of the Administration Workspace.

The default Change of Circumstances workflow should be viewed in the Process Definition Tool. In the Administration Workspace, navigate to the workflow from the left hand navigation menu: **Workflow** > **Released Processes** > **ChangeOfCircumstances**.

7.2 Customizing the default change of circumstances implementation

Complete the following steps to customize the default change of circumstances implementation to suit your custom environment.

Before you begin

You must complete a full analysis of your requirements, and identify the information that you want citizens to be able to modify.

Procedure

1. Customize the default change of circumstances IEG script and data store schema.

    ```
    /components/HCROnline/data/initial/clob/ChangeOfCircumstance.xml
    /components/HCROnline/data/initial/clob/ChangeOfCircumstance.xsd
    ```

    a) In most cases, you are updating the default change of circumstances IEG script to align it with the existing enrollment and internal caseworker scripts.
        For example, you might want to make one of the following changes:

        *   Create a custom evidence entity for which you want to capture data.
        *   Customize a default evidence entity, typically by adding one or more attributes.
        *   Customize the flow of the script. For example, by modifying control questions.
        *   Customize the script to facilitate adding, updating, or removing evidence for a newly added evidence type.

    b) Depending on the changes to the script, you might need to make parallel changes to the schema that is associated with the script.

2. Configure the change of circumstances life event to call your custom script by overriding the change of circumstances entry in the following files:

    ```
    /components/HCROnline/data/initial/LifeEventContext.dmx
    /components/HCROnline/data/initial/LifeEventType.dmx
    ```

3. To add a custom evidence entity, you must write new prepopulator and updater implementations. Customization of Enrolment evidence differs slightly from customization of other evidence types. For Enrolment evidence, use the prepopulator mechanism that is described here. However, for the updater/mapper for Enrolment Evidence use the plan management adapter interface. **Important:** Customization of Enrolment evidence differs slightly from customization of other evidence types. For Enrolment evidence use the prepopulator mechanism described here. However for the updater/mapper for Enrolment Evidence use the plan management adapter interface.

    a) A prepopulator takes the data from a dynamic evidence instance and puts that data into data store format so that it can be read by the script. For information about configuring a new prepopulator, see the Javadoc of the following class:

    ```
    curam.healthcare.lifeevents.coc.prepopulators.impl.Recertification
    ```

    b) An updater, or mapper, takes the data store information after a change of circumstance and identifies which evidence on the ongoing case must be added, modified, or removed. For information about configuring a new updater, see the Javadoc of the following class:

```
curam.healthcare.lifeevents.coc.mappers.impl.LifeEventDefaultEvidenceMapper
```

4. If you are extending a default entity, you must extend the provided prepopulator and mapper classes that are associated with this type and add the custom code. For a list of all of the prepopulator and mapper classes that can be extended, see the following class:

```
curam.healthcare.lifeevents.impl.Module
```

5. Configure the online and program group logic rules to reflect your custom changes. The existing portal and case management rule sets were updated to cater to change of circumstances.

   You can find the online rules in the following location:

```
./EJBServer/components/HCROnline/CREOLE_Rule_Sets/HealthCareReformEligibilityRuleset.xml
```

   You can find the main program group logic rules in the following location:

```
./EJBServer/HCR/CREOLE_Rule_Sets/HCRProgramGroupRuleSet
```

   You can find a rule set per program in the following location:

```
./EJBServer/HCR/CREOLE_Rule_Sets
```

6. Thoroughly test the custom changes made to the change of circumstances process.

   You must ensure the following results:

   - The correct online results are been achieved.
   - The correct information is being written to the ongoing case.
   - The correct program group logic results are being achieved.

7. Customize the change of circumstances workflow.

Related concepts

[The plan management adapter interface on page 65](#)
A plan management interface is provided which customers must implement. The custom implementation allows customers to communicate with their chosen plan management vendor through web services.

Customizing the change of circumstances IEG script

You can complete one or more of the following tasks to customize the change of circumstances script for your custom environment.

About this task
The change of circumstances script starts with a summary page. From this summary page, all of the necessary change of circumstance actions can be done. For example, add, modify and remove, where remove refers to the end-dating of evidence.

Adding custom entities through the change of circumstances script

To add custom entities through the change of circumstance script, you must make the following changes to the script.

Procedure

1. Provide an **Add** link on the summary page. This link must point at an existing page.
2. Add a new data store entity to reflect the new custom evidence entity.
3. Add an attribute called evidenceCoCStatus to this data store entity.

   This attribute is based on the code-table EVIDENCECOCSTATUS, which contains the following values:

   • ADDED
   • MODIFIED
   • REMOVED

   The default for this data store schema attribute is a blank value.
4. Set the newly added attribute to ADDED after the IEG page that gathers the data for this new entity is submitted. This is achieved through invoking the UpdateEvidenceCoCStatus custom function, which takes the name of the entity as a parameter.
5. This attribute can be used as follows in conditions to display or hide data:

   ```
   "IsRecordAdded() or <MyEntity>.evidenceCoCStatus=="ADDED""
   ```
6. The evidence updater can use the value of this attribute to determine any data store entity that must be added as evidence. The ADDED status can also be deduced by using the localID for the entity in question as this is not set for newly added entities. The localID attribute is used to hold the unique identifier of evidence on the database.

Modifying entities through the change of circumstances script

To modify entities through the change of circumstance script, you must make the following changes to the script.

Procedure

1. Provide a **Change** link on the summary page. This link must point at an existing page.
2. Set the evidenceCoCStatus attribute to MODIFIED. This is achieved by comparing the attributes. New validations are added that call a custom function (HasChanged or HasAttrValueChanged, which always return true). The parameters are the new value and the fully qualified attribute name. This function can deduce if the attribute has changed by looking up its original value.
3. An additional Boolean attribute dataSubmitted, which defaults to false, is added to the schema on the data store entity. It is needed in case other page validations fail. The custom function SetDataSubmitted is called in the last validation, setting the flag to true. This has the effect of resetting evidenceCoCStatus to a blank value if this flag is set. The flag is reset to false in the custom function following the page, UpdateEvidenceCoCStatus.

Removing entities through the change of circumstances script

To remove entities through the change of circumstance script, you must make the following changes to the script.

Procedure

1. Provide a **Change** link on the summary page. This link must point at an existing page.

2. Set the evidenceCoCStatus attribute to REMOVED. This is achieved by comparing the attributes. New validations are added that call a custom function (HasChanged or HasAttrValueChanged, which always return true). The parameters are the new value and the fully qualified attribute name. This function can deduce if the attribute has changed by looking up its original value.

3. An extra Boolean attribute dataSubmitted, which defaults to false, is added to the schema on the data store entity. It is needed in case other page validations fail. The custom function SetDataSubmitted is called in the last validation, setting the flag to true. This has the effect of resetting evidenceCoCStatus to a blank value if this flag is set. The flag is reset to false in the custom function following the page, UpdateEvidenceCoCStatus.

Customizing the Change of Circumstances workflow

You can use the following steps to customize the default Change of Circumstances workflow.

Before you begin

You can find the Change of Circumstances workflow in the following location:

```
./EJBServer/components/HCROnline/workflow/
ChangeOfCircumstances_v3.xml
```

Procedure

1. If you want to or add or remove steps, or to change the flow structure of the existing workflow, create a version of the workflow to make your custom changes.

2. Customize the Change of Circumstances workflow in the standard supported fashion of customizing workflows as follows:

   a) Using the Process Definition Tool, view the latest version of the process definition that requires modification. Create a version of that process definition by using the tool.

   b) Make the changes, validate it and release the workflow.

   c) Export the newly released workflow process definition by using the PDT and place it into the workflow subdirectory of the `..\EJBServer\components\custom` directory.

3. If you are not happy with the structure and the steps in the default workflow, you can implement your own version of each step.

4. To customize the automatic steps in the workflow, use the following hook points to implement your own version of a step. Once those implementations are complete, you can bind them by using Guice to ensure that those customized versions of the automatic steps are called when the workflow is enacted. The automatic steps that are available for customization are as follows:

a) curam.healthcare.lifeevents.coc.sl.impl.PreEvidenceProcessing
b) curam.healthcare.lifeevents.coc.sl.impl.CaseAndParticipantProcessing
c) curam.healthcare.lifeevents.coc.sl.impl.EvidenceUpdater
d) curam.healthcare.lifeevents.coc.sl.impl.PostEvidenceUpdater
e) curam.healthcare.lifeevents.coc.sl.impl.EvidenceActivator
f) curam.healthcare.lifeevents.coc.sl.impl.CompleteCoC

5. To change a manual activity step in the Change of Circumstances workflow, update the process definition metadata with your change.

7.3 Configuring the change of circumstance evidence submission workflow

Use curam.healthcare.coc.auto.activate.evidence to configure the submission stage of the change of circumstance workflow.

About this task

The application property allows the change of circumstance evidence to be automatically activated on the associated integrated case, provided there are no outstanding verifications. When set to True, if there are no outstanding verifications, the changes are automatically activated and the product delivery cases are reassessed. The life event status is then set to complete. If there are outstanding verifications, a task is generated to the user to indicate the need for manual intervention by a caseworker. When the verification is actioned by the caseworker and the changes are manually activated, the life event status is set to complete.

The property can also be configured to allow for change of circumstance evidence to be manually reviewed by a caseworker. When set to false, the workflow is configured for manual review of submitted evidence. The reported changes are mapped to the integrated case in an in-edit state, and the life event status is set to complete. The caseworker must manually activate the evidence to redetermine or reassess the eligibility for the household on the product delivery cases.

The default value for this property is True.

Procedure

1. Log in to the Cúram System Administration application as a user with system administrator permissions.
2. From the left navigation menu, select **Application Data** > **Property Adminstration**
3. Select curam.healthcare.coc.auto.activate.evidence in the Application - Insurance Affordability Settings category.
4. Change the value to true to have the submission process attempt to automatically activate the evidence on submission. Change it to false to configure the workflow for manual review and activation of the submitted evidence.

8 Customizing evidence management wizards

To customize an evidence management wizard, you must be familiar with the specific wizard's customization strategies.

8.1 Customizing the Add a Member wizard

You can use three associated wizards to customize the **Add a Member** implementation for your custom environment.

Customizing wizard evidence mappings

The creation of evidence upon finishing the 'Add a Member' evidence management wizard can be customized.

To create more types of evidence via the 'Add a Member' evidence management wizard, an implementation of an interface is required. This new implementation needs to be bound to the evidence type in a custom Guice module.

The interface is

```
curam.healthcare.guidedchanges.sl.impl.HCRAddMemberEvidenceMapper
```

and it consists of the following four functions

- preMapEvidence - Performs assignments to the struct that represents the evidence.
- shouldCreateEvidence - If this function returns true, call the mapEvidence and postMapEvidence functions. If false, do not.
- mapEvidence - Create the evidence.
- postMapEvidence - Typically for a parent evidence mapper to call the mappers of any configured child evidence types.

Sample Customization

```
/**
 * Adding a new custom evidence type.
 */
public class MyCustomTypeEvidenceMapper implements HCRAddMemberEvidenceMapper {
  // implementation details
}
```

The module binding change to accompany the customization is shown here.

```
final MapBinder<String, HCRAddMemberEvidenceMapper> mapperStrategy =
  MapBinder.newMapBinder(binder(), String.class,
    HCRAddMemberEvidenceMapper.class);

mapperStrategy.addBinding(CASEEVIDENCE.MYCUSTOMTYPE).to(
  MyCustomTypeEvidenceMapper.class);
```

To customize an existing implementation of this mapper interface, customers can extend the out-of-the-box implementation and use a Guice linked binding to bind the out-of-the-box implementation to the custom implementation. The custom implementation is injected in place of the out-of-the-box implementation.

Sample Customization

```
/**
 * Customizing an out-of-the-box evidence mapping implementation.
 */
public class CustomDemographicsEvidenceMapper extends DemographicsEvidenceMapper {

  @Override
  public void preMapEvidence(final AddMemberWizardStoredDetails store,
    final long caseParticipantRoleID, final Date startDate)
    throws AppException, InformationalException {

    // re-use the OOTB functionality
    super.preMapEvidence(store, caseParticipantRoleID, startDate);

    // perform additional, custom assignments
    store.personalDetails.demographicDtls.startDate =
      store.personalDetails.demographicDtls.startDate.addDays(1);
  }

  @Override
  public boolean shouldCreateEvidence(
    final AddMemberWizardStoredDetails store,
    final long caseParticipantRoleID, final Date startDate)
    throws AppException, InformationalException {

    // override the OOTB logic
    return true;
  }

}
```

The module binding change to accompany the customization is shown here.

```
bind(DemographicsEvidenceMapper.class).to(CustomDemographicsEvidenceMapper.class);
```

Customizing wizard evidence mapping order

The order of the creation of evidence upon finishing the 'Add a Member' wizard can be customized.

Submitting the Add a Member evidence management wizard results in creation of evidence via the HCRAddMemberEvidenceMapper implementations. The list of evidence types to iterate over is built up dynamically and includes the evidence types configured via the existing implementation of

```
curam.healthcare.lifeevents.impl.HealthCareCoCStaticEvidence.getStaticEvidenceTypesForUpdaterToProces
```

The list of evidence types is then ordered by a call to the following function.

```
curam.healthcare.guidedchanges.sl.impl.HCREvidenceSelection.selectAndSortEvidenceTypes
```

The following default implementation can be extended if the default behavior needs to be customized.

```
curam.healthcare.guidedchanges.sl.impl.HCREvidenceSelectionImpl
```

To customize the existing implementation of this interface, customers can extend the out-of-the-box implementation and use a Guice linked binding to bind the out-of-the-box implementation to the custom implementation. The custom implementation is injected in place of the out-of-the-box implementation.

Sample Customization

```
bind(HCREvidenceSelectionImpl.class).to(CustomHCREvidenceSelectionImpl.class);
```

Customizing wizard evidence dates

Various dates that are used by the 'Add a Member' wizard can be customized.

The interface

```
curam.healthcare.guidedchanges.sl.impl.HCRAddMemberWizardStartDate
```

is used to customize various dates that are manipulated by the evidence management wizard. The operations on this interface are listed here.

- getDefaultEvidenceStartDate - Gets the default start date for evidence created via the wizard
- getDefaultRelationshipStartDate - Gets the initial value to populate the relationship start dates on the wizard.

The out-of-the-box implementation of this interface is

```
curam.healthcare.guidedchanges.sl.impl.HCRAddMemberWizardStartDateDefaultImpl
```

To customize the existing implementation of this interface, customers can extend the out-of-the-box implementation and use a Guice linked binding to bind the out-of-the-box implementation to the custom implementation. The custom implementation is injected in place of the out-of-the-box implementation.

Sample Customization

```
/**
 * Customizing an out-of-the-box evidence dates implementation.
 */
public class CustomWizardStartDateImpl extends HCRAddMemberWizardStartDateDefaultImpl {

  @Override
  public Date getDefaultEvidenceStartDate(
    final AddMemberWizardStoredDetails store) throws AppException,
    InformationalException {

    // custom implementation
  }

  @Override
  public Date getDefaultRelationshipStartDate(
    final AddMemberWizardStoredDetails store) throws AppException,
    InformationalException {

    // custom implementation
  }

}
```

The module binding change to accompany the customization is shown here.

```
bind(HCRAddMemberWizardStartDateDefaultImpl.class).to(CustomHCRAddMemberWizardStartDateDefaultImpl.cl
```

8.2 Customizing the Re-add a Member function

You can customize the **Re-add a Member** implementation for your custom environment.

For information about the **Re-add a Member** function, see the *Re-adding a member to a case* related link.

Related concepts

Customizing default evidence attributes and post-function processing

You can customize the default values that are stored for each of the evidence attributes as the evidence is being created. You can also customize the processing that applies when the **Re-add a Member** function is complete.

The proceeding table lists the functions and the associated descriptions for the interface `curam.healthcare.guidedchanges.sl.impl.HCRReAddApplicant`.

Table 34: The functions and the associated descriptions for the interface
curam.healthcare.guidedchanges.sl.impl.HCRReAddApplicant.

Function	Description
getReAddedHCRApplicant	Determines the values to store for each evidence attribute as the evidence is being created.
postReAddApplicant	Activates after the member is successfully re-added to the household.

To customize the existing implementation, customers can extend the default implementation and use a Guice-linked binding to bind the custom implementation to the interface `curam.healthcare.guidedchanges.sl.impl.HCRReAddApplicant`. As a result, the custom implementation is injected instead of the default implementation. The following code is a sample customization.

```
public class CustomHCRReAddApplicant extends DefaultHCRReAddApplicant {

  @Override
  public HCRApplicantDtls
    getReAddedHCRApplicant(HCRApplicantDtls wizardValues)
      throws AppException, InformationalException {

    // re-use the OOTB default attributes
    HCRApplicantDtls hcrApplicant = super.getReAddedHCRApplicant(wizardValues);

    // perform addition, custom assignments
    hcrApplicant.applicant = isHCRApplicant(wizardValues.caseParticipantRoleID);
  }

  @Override
  public void postReAddApplicant(WizardKey wizardKey,
    EvidenceKey applicantDetailsEvidence)
    throws AppException, InformationalException {

    // call OOTB functionality
    super.postReAddApplicant(wizardKey, applicantDetailsEvidence);

    // preform additional post processing
    unEndDateReleventEvidence(applicantDetailsEvidence);
  }
}
```

The module binding change to accompany the customization is
`bind(HCRReAddApplicant.class).to(CustomHCRReAddApplicant.class);`.

9 Customizing appeal requests

After you install the application, complete the following steps to customize the default implementation of appeal requests to suit your specific requirements.

Procedure

1. If required, update the Citizen Account for your specific requirements.

    a) Create a custom appeal request IEG script and data store schema for your specific requirements.

        You can copy and modify the default IEG script and schema:

        -   `\EJBServer\components\HCROnline\data\initial\clob`
            `\OnlineAppealsSchema.xsd`
        -   `\EJBServer\components\HCROnline\data\initial\clob`
            `\OnlineAppealsSchema.xml`

    b) Implement curam.citizenworkspace.pageplayer.impl.AppealDatastorePrepopulator to populate the data store.

    c) Confirm that the data store prepopulator returns the appropriate set of potential appellants.

    d) Set the application properties to point to the new custom appeal request IEG script and data store schema.

2. Review the generated PDF to ensure that it meets your specific requirements. If required, create an XSL template to modify the PDF to your requirements.

9.1 Setting the appeals requests IEG script and data store schema

In the administration application, set the appeals requests properties to point to the appropriate appeals requests IEG script and data store schema.

Procedure

1. Log in to the Cúram System Administration application as a user with system administration permissions.

2. Click **System Configurations** > **Application Data** > **Property Administration**

3. In the **Citizen Portal - Online Appeals Configuration**

4. Set the following properties to point to the appeal requests IEG script and data store schema:

    ```
    curam.citizenworkspace.appeals.datastore.schema
    curam.citizenworkspace.appeals.script.id
    ```

9.2 Customizing the appeal request summary PDF document

By default, a PDF that summarizes the information that is entered by a citizen during an appeal request is created when a citizen submits an appeal. You can configure the XSL template of this PDF to change the default PDF document to suit your specific requirements.

About this task

The default XSL template file is *CURAM_DIR\EJBServer\components\HCROnline \data\initial\blob\XSLTEMPLATEINST001.xsl.*

10 Customizing the handling of closed cases

Complete the following tasks to configure and customize how closed cases are handled to your specific requirements.

10.1 Configuring the permanent closure of closed cases

Complete the following steps to configure the closed case reason so that a closed case remains closed permanently. Overriding the default behavior for closed cases ensures that cases that were created in error, or that you want remain closed are not reopened during the routine processing of case changes.

Procedure

To define the closed reasons that will prevent a case from ever being reopened, set the `curam.miscapp.productDeliveryReactivateClosedReason` property to a comma-separated list of the code table values for the closed reasons.

Results
If the case is closed with a reason that is configured in this property, then the case is not reassessed on closure and is never reopened. If the case is closed with a reason that is not configured in this property, then the case is reassessed on closure and can be reopened.

10.2 Configuring the reassessment strategy for closed cases

By default, the reassessment strategy is set to 'Do not reassess closed cases' for HCR product delivery cases. The reassessment strategy for the product delivery cases can be configured with 'Do not reassess closed cases' or 'Automatically reassess all cases'.

Procedure

1. Log in to the Cúram Administration application as a user with administrator permissions.
2. Click **Shortcuts** > **Case** > **Product Delivery Cases**
3. Open the HCR PD Case Home Page.
4. Click **Rule Sets** > **Published** > **Eligibility Determination**
5. Set the value of **Reassessment strategy** to **Do not reassess closed cases** or to **Automatically reassess all cases**.

10.3 Customizing the reassessment implementation for closed cases

By default, product delivery cases that are closed as created in error are not reassessed. Complete the following steps if you want to change the reassessment implementation for closed cases

About this task

For more information, see "Eligibility and Entitlement Engine Hooks" in the *Inside Eligibility and Entitlement Using Cúram Express Rules Guide*.

Procedure

To customize the reassessment implementation for closed cases implement the Eligibility and Entitlement engine hook `curam.core.sl.infrastructure.assessment.impl.ReassessEligiblityHook`

11 Customizing Trigger Points

Customize default Outbound Account Transfer trigger decision.

Default decision to trigger an Outbound Account Transfer can be customized. Custom Event listener must be implemented to customize trigger decision.

```
curam.hcr.fedexchange.eligibility.impl.EligibilityProcessor.EligibilityEvent.sendOutboundApplication(
  currentOutcome, Boolean[] customOutcome, CaseHeader caseHeader, Entity datastoreRoot)
```

is an event that allows a custom listener to perform additional checks before an Account Transfer application is created and sent to the FFM.

Event has following parameters:

currentOutcome - boolean with default outcome. This parameter can not be modified.

customOutcome - boolean array where custom outcome can be set. Only one Boolean will exists in array and it can be updated by the custom listener.

caseHeader - CaseHeader object of a case that eligibility has been determined against.

datastoreRoot - root of the datastore that will be used to create an Outbound Account Transfer payload.

Customize Application Case denial Outbound Account Transfer trigger.

When state initiated application is denied with denial reason other than procedural denial Account Transfer must be triggered. FFM originated application will trigger an Account Transfer regardless of denial reason.

Default denial reasons triggering an Outbound Account Transfer are:

- Client Ineligible value="IPADR1002"
- Already In Receipt Of Program value="IPADR1003"

Both are code table items from the IntakeProgramApplicationDenialReason code table. Custom values can be used by EligibilityProcessorMap.getDenialReasonList() override. Values form List of IntakeProgramApplicationDenialReasonEntry objects.

Default denial reasons can be added by calling super class method as per example below:

```
public List<IntakeProgramApplicationDenialReasonEntry>
        getDenialReasonList() {
        final List<IntakeProgramApplicationDenialReasonEntry> denialReasonList =
          super.getDenialReasonList();
        denialReasonList
        .add(IntakeProgramApplicationDenialReasonEntry.CUSTOMDENYREASON);

        return denialReasonList;
        }
```

12 Implementing periodic data matching and annual renewals

From a technical perspective, annual renewals is a specific use case of periodic data matching, with some specific annual renewal requirements. The shared technical infrastructure that is provided for periodic data matching and annual renewals contains the required configuration, customization, and extension points for you to implement your custom solution.

12.1 Storing all existing program group determinations

Before version 6.0.5.5 interim fix 2, program group determinations were not saved in the database. If you upgrade from an earlier version, you must run the BulkRunProgramGroupEligibility batch process on your system before you run any of the periodic data matching or annual renewals batch processes. The BulkRunProgramGroupEligibility batch process identifies and stores all of the current program group determinations in your system. This once-off task for each system captures information that is required for projected eligibility comparisons.

About this task

Before you run the BulkRunProgramGroupEligibility batch process to store the determinations, you can run SQL commands to identify how many cases will be processed by the batch process (Count A), and the number of stored program group determinations (Count B), which should be zero before you run the batch process. After the batch process is completed, you can run the same SQL to identify the actual number of program group determinations that were stored (Count C).

After the batch process is run, the batch log file shows the number of cases processed (Count D) and the number of cases skipped (Count E). To verify the results, you compare the count values.

Procedure

1. Run the following SQL command to identify all cases that are processed by the batch process (Count A).

   ```
   SELECT COUNT(*) FROM CASEHEADER WHERE CASETYPECODE= 'CT5' AND
   STATUSCODE= 'CS4' AND INTEGRATEDCASETYPE= 'CT26301'
   ```

2. Run the following SQL command to show the number of stored program group determinations, which should be zero before you run the batch process (Count B).

   ```
   SELECT COUNT(*) FROM PROGRAMGROUPDETERMINATION WHERE
   CREOLEPROGGRPDETERMINATIONID  IS NOT NULL AND RECORDSTATUS='RST1';
   ```

3. Run the BulkRunProgramGroupEligibility batch process to store determinations for cases.

4. Run the following SQL command to determine the actual number of program group determinations that were stored (Count C).

   ```
   SELECT COUNT(*) FROM PROGRAMGROUPDETERMINATION WHERE
   CREOLEPROGGRPDETERMINATIONID  IS NOT NULL AND RECORDSTATUS='RST1';
   ```

5. Review the batch log file for any technical issues.

6. Verify your results by getting the remaining count values from the log files and comparing the different count values.

   The batch log file shows the number of cases processed (Count D) and the number of cases skipped (Count E).

   To verify the results, compare following count values:

   - Count B should equal 0
   - Count C should equal Count D
   - Count A should equal (Count D + Count E)

   If Count E>0 then that indicates that the batch encountered an error, review the batch log for details.

Related tasks

[Customizing the storage of program group determinations on page 130](#)
From version 6.0.5.5 interim fix 2 onwards, all program group determinations are stored in the database by default. Over time the number of determinations can become significant and increase the size of the database table. You can use the provided hook point to suppress the storage of identical program group determinations to reduce the size of the database table.

BulkRunProgramGroupEligibility batch process

This once-off batch process runs the program group logic with the current evidence and stores the results in the database.

The determinations are stored in the CreoleProgGrpDetermination and CreoleProgGrpDeterData tables. For each active determination, a row is added and the batch process updates a new CREOLEPROGGRPDETERMINATIONID field on the PROGRAMGROUPDETERMINATION table with the active determination ID.

Class and method

curam.healthcare.sl.intf.BulkRunProgramGroupEligibility.process

12.2 Developer overview of periodic data matching and annual renewals

Use this overview diagram to understand the development tasks required to implement a custom periodic data matching or annuals renewals solution.
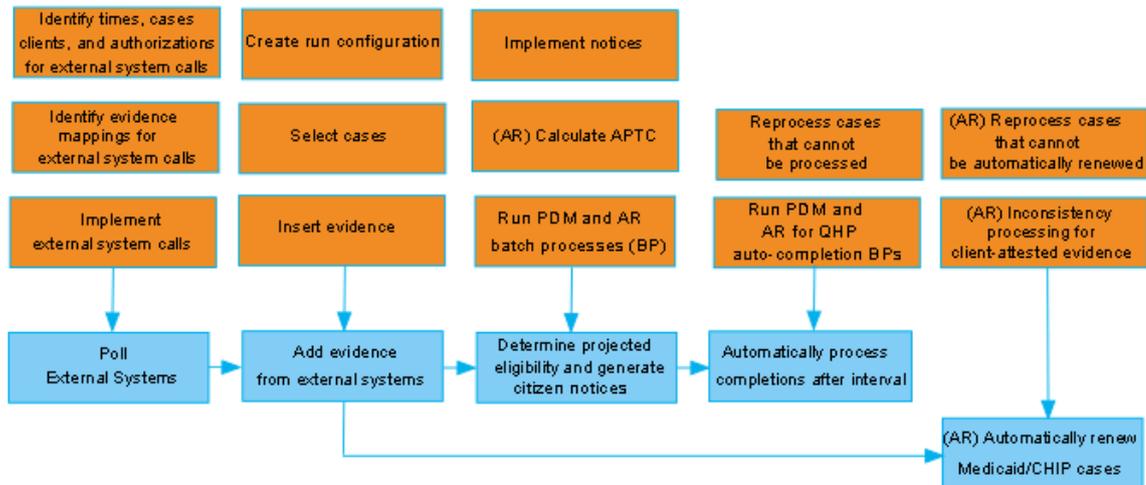
Figure 2: Developer tasks associated with the respective high-level process steps

- **Polling external systems**
  Identify the timing, cases, and clients for which external systems are polled for data and ensure that client authorization is available for accessing their data. Review the mappings of data to the Cúram external evidence types. Customize your implementation to poll the external systems and retrieve the data.
- **Adding evidence from external systems**
  Using a run configuration to track each individual run, write the received data for the selected cases to the Cúram application with the provided API.
- **Determining projected eligibility and generating citizen notices**
  Implement your custom notices, calculating the APTC values for annual renewals only. Run the appropriate batch processes for periodic data matching, or annual renewals for QHP, Medicaid, or CHIP.
- **Automatically process completions after interval**
  Run the automatic completion batch process for periodic data matching or annual renewals for QHP at a defined interval after you run projected eligibility, by default 30 days. External evidence converters convert external evidence to evidence on the case. You must create a

custom implementation for your organization to process the cases that cannot be automatically processed.

- **Automatically renewing Medicaid or CHIP cases**
Annual renewals for Medicaid or CHIP are automatically completed where possible. You must create a custom implementation for your organization to process the cases that cannot be automatically renewed.
For annual renewals, if a client has outstanding verifications against their client-attested evidence, caseworkers can give them a reasonable opportunity period to provide the verifications, allowing the case to temporarily proceed as if verifications had been provided.

12.3 Polling external systems

The goal of periodic data matching is to ensure that an organization has the most up-to-date client information so that the system can correctly assess each citizen for eligibility and provide the correct entitlements. To achieve this goal, an organization can use trusted data source (TDS) services to query external systems for current data. TDS services provide services such as verifying annual tax return information, or verifying a citizens birth and death details or current address.

From the Cúram application perspective, TDS services are just another means of adding evidence to a case, like a caseworker who adds evidence through the caseworker application.

For example, when a client is applying for a program, TDS services are used to verify information that is provided by the prospective client and that the information is stored on the citizens case.

The method of retrieving information from TDS services during periodic data matching is different from the initial intake process. During periodic data matching, the TDS services are typically accessed in a bulk request mode. Information for 10,000's or 100'000's of citizens is acquired through a single request through a dedicated bulk service. The Cúram application does not provide any explicit functions for connecting to TDS services. However, bespoke periodic data matching APIs are available to handle the creation of the large volumes of evidence records that are expected to be returned from TDS bulk services.

12.4 Adding evidence from external systems

Complete the following tasks to write information that was retrieved from external systems to the Cúram system.

Creating a batch run configuration for annual renewals or periodic data matching

A run configuration is mandatory for all annual renewal or periodic data matching batch process runs. Each run configuration contains a unique run ID that you must use to link all of the

individual batch process steps for a particular periodic data matching or annual renewals run. You need a new run configuration for each run.

About this task

To insert evidence for a periodic data matching run, you must create a periodic data matching run configuration before you call the PDMEvidenceMaintanence API. The ID in the run configuration supports linking batch jobs that are all part of the same run. For example, to process the volume of renewals that are needed for annual renewals 2014, you need multiple run iterations over a series of batch windows.

Procedure

1. Log in to the Cúram Administrator application as a user with administrator permissions.
2. Open the **Administration Workplace** tab.
3. Expand the shortcuts pane and select **Health Care Reform** > **PDM Run Configuration**.
4. On the **PDM Run Configuration** page select **New**. The **New PDM Run Configuration** window opens.
5. Identify the run type.

   - If you are creating a run configuration for an annual renewal, select the **Annual Renewal Indicator** check box and select the annual renewal type from the **Renewal for** menu.
   - If you are creating a run configuration for a periodic data match, clear the **Annual Renewal Indicator** check box.

6. Enter a unique run ID.
   For example,

   - QHP_2014
   - CHIP_2014
   - PDM_2014_Q1

   Note the run ID value as you need this value when you run the batch processes.

7. Enter a short name for the run configuration.
8. Click **Save**.

Implementing case selection for a batch run

Before you run a periodic data matching or annual renewal, you must ensure that all of the required cases are selected for processing. The cohort of cases that you select for a run depends on the requirements of your process. Before you add case to a run, you must ensure that the client has consented to have their data polled and eligibility checked.

Before you begin

Before you add cases to a periodic data matching or annual renewal run, you must define a run configuration Attempting to add cases to a non-existent run causes errors.

About this task

For a typical periodic data matching run, you generally need to reassess only those cases for which updated evidence is received. In this case, adding evidence to the run through the PDM Evidence Maintenance API is sufficient to ensure that the cases are processed.

For an annual renewal run, you must ensure that all of the mandatory cases are selected for processing.

If you want to reassess cases regardless of whether updated evidence is received, as in the case of a typical annual renewal run, you must explicitly add cases to the run. You can add cases by first implementing a custom batch streaming process to select all of the cases of interest. Then, use the addCase method of the Run Case Control Manager API to add these cases to the run if they are not present

Example code snippet:

```
public BatchProcessingSkippedRecord processRecord(
final BatchProcessingID batchProcessingID, final YOUR_PROCESS_KEY key)
throws AppException, InformationalException {

final String runID = key.runID;
final long caseID = batchProcessingID.recordID;


if (pdmRunCaseControlManager.getCase(runID, caseID) == null) {
pdmRunCaseControlManager.addCase(runID, caseID);
}


}
```

For more information about the PDMRunCaseControlManager API, see the Javadoc for the API.

Inserting evidence from external systems with the PDMEvidenceMaintenance API

Use the PDMEvidenceMaintenance API to add evidence that you have retrieved from external systems to integrated cases as external evidence by using a bulk update of dynamic or static evidence.

Before you begin

Ensure that current client information was retrieved from external systems through a custom TDS service.

About this task

The PDMEvidenceMaintenance API provides an integration point that hooks external evidence into the annual renewal and periodic data matching processes by creating corresponding periodic data matching run evidence control records for each piece of external evidence. Evidence that is created or updated through this API is recorded as In-Edit evidence and associated with the appropriate periodic data matching run case control record.

If required, you can optionally activate the evidence by using the standard EvidenceControllerInterface. The EvidenceControllerInterface ignores whether the evidence type is static or dynamic. However, evidence activation is not required to drive these processes as projected eligibility and notice generation operates on both In Edit and Active evidence.

For more information, see the Javadoc for the PDMEvidenceMaintenance API.

Procedure

1. Create a periodic data matching run configuration and note the run ID, which you must associate with each of the process steps in the run.

2. Insert evidence by using the PDMEvidenceMaintanence API and passing in the run ID as a parameter.

12.5 Advising caseworkers about income evidence mismatches

Use the sample ARIncomeAdvisorRuleSet Advisor rule set as a example implementation of building advice for evidence mismatches.

A sample Advisor rule set is provided which can be used to advise a caseworker when client attested income evidence that is part of an annual renewal that is submitted by the case worker needs to be reviewed for re-verification.

The Advisor rule set compares client attested income and income evidence that was added as a result of polling the external system. When evidence items are not reasonably compatible with each other, advice is displayed on the Integrated Case home page and Evidence Dashboard page. When a case worker completes an annual renewal, they are asked to confirm that the client attested evidence on the case has been reviewed and verified as necessary. When the case worker confirms that the evidence is reviewed and verified, the advice is no longer displayed.

You can implement advice for other evidence types using Advisor rule sets and including configurations for where this advice should appear. The advice category for this type of annual renewal Advisor rules should be set to *AREVDMIS* in the AdviceCategory code table.

12.6 Implementing citizen notices

Complete the following tasks to implement citizen notices and to configure the load balancing of the XML server.

Implementing citizen notice generation

During the periodic data matching or annual renewals processes, notices must be generated and sent to citizens to inform them of the process and the implications it has for their eligibility, entitlement, and coverage. By default no notice is generated. Use the following information to help you implement notice generation for both periodic data matching and annual renewals.

About this task

To implement citizen notice generation for both periodic data matching and annual renewals, you must decide on the notices that you want to generate and their contents. You must create a custom

XSL template to present the information. You must then create an implementation that retrieves data to populate and call the notice generation.

Citizen notices are generated through the periodic data matching and annual renewals batch processes. These batch processes call the notices infrastructure, which uses the curam.hcr.pdm.notices.impl.PDMNotificationTemplate interface to get the XSL template to generate the notice.

For more information about specific APIs, see the Javadoc for the API.

Procedure

1. Complete a business analysis task to identify the information that you need in each notice.

2. Create a custom XSL template that sets up the layout and data placeholders for the notice.

3. Create a custom implementation that retrieves the identifier of the custom XSL template.

   a) This implementation should extend curam.hcr.pdm.notices.impl.PDMNotificationTemplateImpl for future compatibility as new methods are added over time. Each method represents a point in the business process at which a notice is sent. All methods should return a code table entry from the TemplateIDCode code table. You must add an entry to this code table for each template identifier returned.

   b) Register the custom implementation by using Guice bindings to bind the implementation to the PDMNotificationTemplate interface. You must register new Guice modules adding a row to the ModuleClassName database table.

   - **Example Binding**

```
public class ExampleModule extends AbstractModule {
  public void configure() {
   bind(PDMNotificationTemplate.class).to(ExampleNoticeTemplateImpl.class);
  }
}
```

   - **Example Citizen Notice Template Implementation**

```
   public class ExampleNoticeTemplateImpl extends PDMNotificationTemplateImpl {

  @Override
   public TEMPLATEIDCODEEntry getNotificationTemplate(
     final PDMARNotificationWrapperDetails details)
     throws AppException, InformationalException {

      return TEMPLATEIDCODEEntry.EXAMPLETEMPLATE;
   }

  @Override
   public TEMPLATEIDCODEEntry getRenewalNotificationTemplate(
     final PDMARNotificationWrapperDetails details)
     throws AppException, InformationalException {

      return TEMPLATEIDCODEEntry.EXAMPLERENEWALTEMPLATE;
   }
```

4. Create a custom implementation that retrieves the data for the notice.

   a) This implementation should implement the curam.healthcare.sl.impl.HCRNotificationGenerator interface and provide an

implementation for the generateNotification method. Use this method to populate the data for the notice and call the notice generation.

b) You can retrieve the data for the notice by using the curam.hcr.pdm.notices.impl.PDMNotificationDataRetrieval API. If you want to retrieve extra data with this API, then you must extend the default implementation curam.hcr.pdm.notices.impl.PDMNotificationDataRetrievalImpl.

c) Register the custom implementation by using Guice bindings to bind the implementation to the HCRNotificationGenerator interface. The binding happens on the ID of the template. You must register new Guice modules by adding a row to the ModuleClassName database table.

- **Example Binding**

```
public class SampleModule extends AbstractModule {
  public void configure() {
    final MapBinder<Long, HCRNotificationGenerator> mapbinder =
  MapBinder.newMapBinder(binder(), Long.class, HCRNotificationGenerator.class);
    mapbinder.addBinding(12345).to(ExampleNoticeGenerator.class);
  }
}
```

- **Example Citizen Notice Implementation**

```
public class ExampleNoticeGenerator implements HCRNotificationGenerator {

  public HCRNotificationDetails generateNotification(
    final NotificationGenerationDetails notificationGenerationDetails)
    throws AppException, InformationalException {

    final HCRProFormaDataGenerator<PDMARNotificationDetails>
      documentGenerator =
        new HCRProFormaDataGenerator<PDMARNotificationDetails>();

    final PDMNotificationDataRetrievalImpl pdmNotificationDataRetrival =
      new PDMNotificationDataRetrievalImpl();

 final PDMARNotificationDetails notificationDetails =
      new PDMARNotificationDetails();

    // Example of retrieving data using the PDMNotificationDataRetrievalImpl API
    notificationDetails.primaryClient = pdmNotificationDataRetrival
       .getPrimaryCorrespondentName(notificationGenerationDetails
         .getCaseKey());

     // Invoke notice generation
    final HCRNotificationDetails hcrNotificationDetails =
      documentGenerator.generateHCRNotification(notificationDetails,
        notificationGenerationDetails.getXslTemplateInstanceKey(),
        "SampleNotice");

    return hcrNotificationDetails;
  }
}
```

Implementing the calculation of APTC for inclusion in notices

During annual renewals processes, the Annual Premium Tax Credit (APTC) amount must be calculated for the coming coverage period and included in the notification sent to customers. Use the following information to help you implement the calculation.

About this task

To implement the inclusion of the APTC amount, create a new type of evidence that is identical to BenchmarkPlan. Create an evidence handler that converts that evidence into a

BenchmarkPlan RuleObject to be used in the APTC calculation. You must implement an event that retrieves the BenchmarkPlanDetails and creates your new evidence by using these details. Within the event, you must then run executeProgramGroupProjectedEligibility and from the returned ProgramGroupProjectedEligibility calculate the APTC amount. The original ProgramGroupProjectedEligibility must be updated to reflect this change.

The APTC is calculated through the annual renewals batch processes.

Procedure

1. Create a type of evidence that is identical to the BenchmarkPlan evidence. For example, ProjectedBenchmarkPlan.

2. Create an Evidence handler that converts the new evidence type into an in-memory RuleObject for use in the APTC calculation.

   a) This event implementation implements curam.healthcare.sl.impl.ProjectedEligibilityEvidencehandler and provides an implementation for the defineInMemoryRuleClasses and createRuleObjects methods. The defineInMemoryRuleClasses method returns a list of fully qualified rule classes to be created in memory by the Evidence Handler. By returning a rule class, the handler prevents rule objects for this rule class from being loaded from the database. The createRuleObjects method uses the evidence above to create the specified in memory RuleObject. For example, BenchmarkPlan.

   b) Register the evidence to the Evidence Handler by using Guice bindings. Register new Guice modules by adding a row to the ModuleClassName database table.

   - **Example defineInMemoryRuleClasses method**

```
public Set<String> defineInMemoryRuleClasses(final CaseKey caseKey, final Session session,
      final EvidenceDescriptorDtlsList evidenceDescriptorDtlsList,
      final PROJECTEDELIGIBILITYTYPEEntry projectedEligibilityType) {
      return new HashSet<String>() {
   {
         add("BenchmarkPlanDataRuleSet.BenchmarkPlan");
      }
      };
```

   - **Example Binding**

```
public class SampleModule extends AbstractModule {

      protected void configure() {

      // final Bind ProjectedBenchmarkPlan evidence to
      // the evidence handler
final MapBinder<String, ProjectedEligibilityEvidenceHandler> projectedEligibilityRules =
         MapBinder.newMapBinder(binder(), String.class,
          ProjectedEligibilityEvidenceHandler.class);

      projectedEligibilityRules.addBinding(
      CASEEVIDENCE.PROJECTEDBENCHMARKPLAN)
         .to(ProjectedBenchmarkPlanEvidencehandlerImpl.class);
      }
}
```

3. Create an event implementation that retrieves the BenchmarkPlan details, adds the custom BenchmarkPlan evidence to the case, recalculates the ProgGrpProjectedEligibility and the APTC amount.

a) This event implementation implements curam.hcr.pdm.sl.impl.PDMBatchEvents and provide an implementation of postProjectedEligibility method from which the APTC amount is calculated.

b) Read the HCRProgramGroup RuleObject from the ProgGrpProjectedEligibility method parameter.

- **Example reading RuleObject from ProgGrpProjectedEligibility**

```
// Get the HCRProgramGroup RuleObject from the Snapshot
        final RuleObject hcrProgramRuleObject =
      programGroupProjectedEligibility.getDetermination()
.getRuleObject();
```

c) Get the list of Eligible Programs (RuleObjects) from the HCRProgramGroup. Loop through each IA program.

1. Read the "eligibleProgramsTimeline" for the HCRProgramGroup RuleObject.
2. Loop through each interval of the eligible programs timeline.
3. Loop through the programs from each interval.
4. Return a list of programs whose "pdCreationCheckStartDate" & "pdCreationCheckEndDate" overlap with the interval start and end date.

d) Loop through each program of type Insurance Assistance. Retrieve the coverage start date and create a list of benefit members.

1. Retrieve the "benefitUnitTimeline" from the program
2. Read the value of this timeline on the program's coverage start date.
3. Iterate through each of the RuleObjects previously returned in the point and read the "caseParticipantRoleRecord" RuleObjects.
4. Return a list of "caseParticipantRoleRecord" numbers that are retrieved from each case participant role records above.

e) Create the list of BenchmarkPlanApplicantDetails with the list of benefit members that you previously created.

f) Create a web service call to get BenchmarkPlan Details, passing the BenchmarkPlan Applicant Details that you previously created and the enrollment type, which is Annual Renewals in this case.

- **Example BenchmarkDetails web service call**

```
@Inject
      private PlanManagementAdapter planManagementAdapter;

    final BenchmarkPlanDetails benchmarkPlanDetails =
          planManagementAdapter.getBenchmarkPlanDetailsForBenefitMembers(
            benchmarkPlanApplicantDetailsList, enrollmentType);
```

g) Create your custom evidence and apply the changes to the case. For example, ProjectedBenchmarkPlan evidence.

- **Example getBenchmarkApplicantDetailsList method**

```
@Inject
      private PlanManagementAdapter planManagementAdapter;

    final BenchmarkPlanDetails benchmarkPlanDetails =
          planManagementAdapter.getBenchmarkPlanDetailsForBenefitMembers(
            benchmarkPlanApplicantDetailsList, enrollmentType);
```

h) Add the EvidenceDescriptorDtls related to the ProjectedBenchmarkPlan
    evidence that you previously created to the EvidenceDescriptorDtlsList
    (postProjectedEligibility method parameter). Run
    ProgramGroupProjectedEligibilityManager.executeProgramGroupProjectedEligibility
    passing in the CaseKey, EvidenceDescriptorDtlsList, and ProjectedEligiblity Type,
    which recalculates the ProgramGroupProjectedEligibility.

i) Read back the HCRProgramGroup RuleObject from this new
    ProgramGroupProjectedEligibility as done in point 4b. Loop through every eligible
    program of type IA and Calculate the APTC amount by call getValue on its RuleObject
    attribute.

- **Example Calculating APTC by calling its Rule's value**

```
final RuleObject eligibilityCalculator =
    (RuleObject) program.getAttributeValue(
        HCRCaseConst.kEligibilityCalculator).getValue();

    final RuleObject financialsCPRCalculator =
        (RuleObject) eligibilityCalculator.getAttributeValue(
          "financialsCPRCalculator").getValue();

    // Calculate APTC
   financialsCPRCalculator.getAttributeValue("maximumPremiumTaxCredit")
       .getValue();
```

j) Update the ProgramGroupProjectedEligibility on the database with the
    CREOLEProgramGroupDetermination from the new ProgramGroupProjectedEligibility
    obtained in point 4i.

- **Example Updating ProgramGroupProjectedEligibility**

```
// Get new creoleProgramGroupDetermination
final CREOLEProgramGroupDetermination creoleProgramGroupDetermination =
     programGroupProjectedEligibility2.getDetermination();

    // update programGroupProjectedEligibility with new
    // creoleProgramGroupDetermination
    programGroupProjectedEligibility
      .setDetermination(creoleProgramGroupDetermination);
    programGroupProjectedEligibility.modify();
```

k) Register the custom event by using Guice binding to bind the implantation to the
    PDMBatchEvents interface.

- **Example Binding**

```
// PDMBatchEvent binding to PDMBatchEventImpl
    final Multibinder<PDMBatchEvents> pdmBatchEventsListener =
      Multibinder.newSetBinder(binder(), PDMBatchEvents.class);

    pdmBatchEventsListener.addBinding().to(PDMBatchEventsImpl.class);
```

Configuring XML server load balancing for notices

If you plan to generate large volumes of notices by using the XML Server, ensure that the load is shared among a number of servers. Set the curam.xmlserver.host and curam.xmlserver.port properties to specify the appropriate ports and servers for load balancing and failover.

About this task

The curam.xmlserver.host property specifies the names of the computers that host the XML Server as a forward-slash (/) separated list of host names.

The curam.xmlserver.port property specifies the ports on which the XML Server is running as a forward-slash (/) separated list of entries.

There is a one-to-one mapping between the servers and ports that are specified.

Procedure

You can specify the XML servers in one of two ways as follows:

a) In the Administration application, set the curam.xmlserver.host and curam.xmlserver.port properties.
   For example:

```
curam.xmlserver.host="host1/host2"
curam.xmlserver.port="port1/port2"
```

a) When you run a batch process, specify the -Dcuram.xmlserver.host and -Dcuram.xmlserver.port parameters:
   For example:

```
-Djava.jvmargs="-Dcuram.xmlserver.host=host1/host2"
-Djava.extra.jvmargs="-Dcuram.xmlserver.port=port1/port2"
```

12.7 Overview of the periodic data match batch process flow

The periodic data match batch flow consists of a number of discrete but interrelated batch processes. There are four steps to the complete flow.

To complete the end to end PDM process in batch, you must implement a number of custom batch jobs that work with the provided PDMProjectedEligibility and PDMProcessAutoCompletions batch jobs. An overview of the PDM batch flow is shown in the following figure. Steps 1 and 2 are custom batch jobs that you must implement.
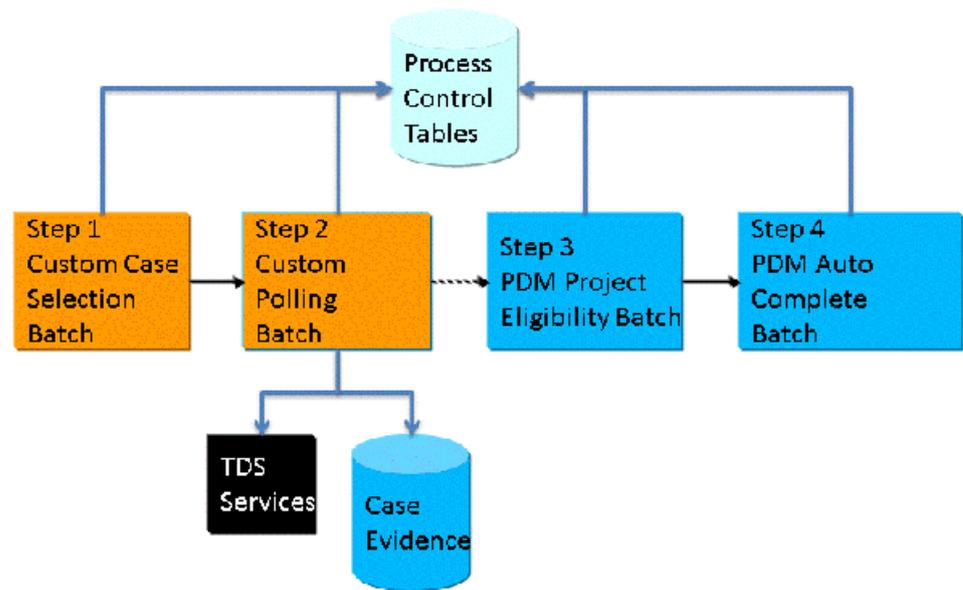
Figure 3: PDM batch process flow

- **Custom Case Selection and Custom Polling batch processes**

  These custom processes should identify cases to be processed for each PDM run. At a minimum, they should identify the cases to be processed and write this information to the PDMRunCaseControl table.

  They should also do the following:

  - Poll for information from external systems

- Populate the PDM process control tables
- Populate evidence tables with the retrieved case information
- **PDM Projected Eligibility batch**

  The PDMProjectedEligibility batch runs the projected eligibility and notice generation process for a specified list of cases (stored on the PDMRunCaseControl table).

  For each case that is successfully processed, the corresponding record on the process control table is updated with the following:

  - ID of the projected eligibility record
  - ID of the generated notice record
  - Status is set to 'Notice Generated (RCCS26002)'
- **PDM Auto Complete batch**

  The PDMProcessAutoCompletions batch process completes the periodic data match after a configured period.

  A citizen can contest the updated evidence by the end of the specified period via their citizen account or in person through a caseworker. If the updated evidence is not contested, the batch process redetermines eligibility with the evidence in the projected eligibility notice. To run this batch process at the command line, pass the value of the RunID to the batch process.

  For each case that is successfully processed, the corresponding record on the process control table is updated with the status set to 'Automatically Completed (RCCS26003)'.

12.8 Running the periodic data matching batch processes

Complete the following tasks to run the periodic data matching batch processes. Periodic data matching is split between two batch processes that you must schedule independently.

Before you begin

> **Important:** Ensure that the current client information was retrieved from external systems through your custom TDS service and was written to the integrated cases as external evidence by using the PDMEvidenceMaintenance API.

Before running a periodic data matching run, ensure that no other period data matching or annual renewals runs are in process. It is recommended that period data matching or annual renewal runs do not overlap.

About this task

The first batch process runs projected eligibility and calls the custom generate notices implementation to send the appropriate citizen notices.

The second batch process picks up cases based on the notice generation date, completes the processing, and notifies the client of any changes to their coverage.

You must run this batch process approximately 31 days from the date that the notice generation occurred. The number of days is dependent on the date the notice was sent to the user. If you run the first batch process late at night, some notices can be generated a day later than others. By default, the batch process picks up cases where the notification was sent 30+1 days prior.

Procedure

1. Create or identify a run configuration in the Cúram Administration application.
2. Run the PDMProjectedEligibility batch process, passing in the run ID as a parameter. For example, to run this batch at the command line:`ant -f app_batchlauncher.sample.xml -Dbatch.username=superuser -Dbatch.program= curam.hcr.pdm.sl.intf.PDMProjectedEligibility.process -Dbatch.parameters=runID=PDM_Q1_14`
3. Run the PDMProcessAutoCompletions batch process, passing in the run ID as a parameter.

PDMProjectedEligibility batch process

This batch process is used to run the projected eligibility and notice generation processes for a specified list of cases as part of periodic data matching.

For periodic data matching runs, inserting evidence by using the PDMEvidenceMaintanence API automatically ensures that the case is processed by periodic data matching projected eligibility and notice generation processes.

Class and method

curam.hcr.pdm.sl.intf.PDMProjectedEligibility.process

Parameters

Parameter	Description	Default value
runID	Mandatory. The run ID value that you specified in the periodic data matching run configuration in the Administration application.	A different value is required for each run.
instanceID	Optional. An instance ID can be used by the batch infrastructure to stop or restart a batch process.	BPN26007
processingDate	Optional.	The current date.
runInstanceID	Do not set. An internal batch field for communicating the Run Instance ID between the batch main and batch stream components.	Not applicable

PDMProcessAutoCompletions batch process

This batch process completes periodic data matching processing after a configurable period. If a citizen has not contested or confirmed the changed information at the end of the specified period,

either online or through a caseworker, this batch process redetermines their eligibility as per the evidence in the projected eligibility notice.

You must schedule the batch process to run after a period of curam.citizenaccount.periodicdatamatch.expiry.days +1 days. All cases that are in the "Notice Generated" state for that period are processed. By default, the value of curam.citizenaccount.periodicdatamatch.expiry.days is 30 days.

Class and method

curam.hcr.pdm.sl.intf.PDMProcessAutoCompletions.process

Parameters

Parameter	Description	Default value
runID	Mandatory. The run ID value that you specified in the periodic data matching run configuration in the Cúram Administration application.	A different value is required for each run.
instanceID	Optional. An instance ID can be used by the batch infrastructure to stop or restart a batch process.	BPN26007
processingDate	Optional.	The current date.
runInstanceID	Do not set. An internal batch field for communicating the Run Instance ID between the batch main and batch stream components.	Not applicable

12.9 Configuring automatic completion intervals for periodic data matching

You can modify the number of days that are allowed for citizens to respond to changes that result from periodic data matching. Complete the following steps to modify the default expiry intervals for your requirements.

About this task
The default expiry period for periodic data matching is 30 days.

Procedure

1. Log in to the Cúram application as a user with administrator permissions.
2. Modify the value of the curam.citizenaccount.periodicdatamatch.expiry.days property.

12.10 Configuring and running the annual renewals batch processes

Complete the following tasks to configure and run annual renewals batch processes for Qualified Health Plans (QHPs), Medicaid, and Children's Health Insurance Plan (CHIP).

Configuring automatic completion intervals for annual renewals

The number of days that are allowed for citizens to respond to changes that result from annual renewals are set out by legislation and are subject to change. Complete the following steps to modify the default expiry intervals for your requirements.

About this task
The default expiry period for annual renewals is 30 days.

Procedure

1. Log in to the Cúram application as a user with administrator permissions.
2. Modify the value of the curam.citizenaccount.annualrenewal.expiry.days property.

Overview of the QHP annual renewal batch process flow

The QHP annual renewal batch flow consists of a number of discrete but interrelated batch processes. There are four steps to the complete flow.

To complete the end to end QHP annual renewal process in batch, you must implement a number of custom batch jobs that work with the provided QHPProjectedEligibility and QHPProcessAutoCompletions batch jobs. An overview of the QHP batch flow is shown in the following figure. Steps 1 and 2 are custom batch jobs that you must implement.
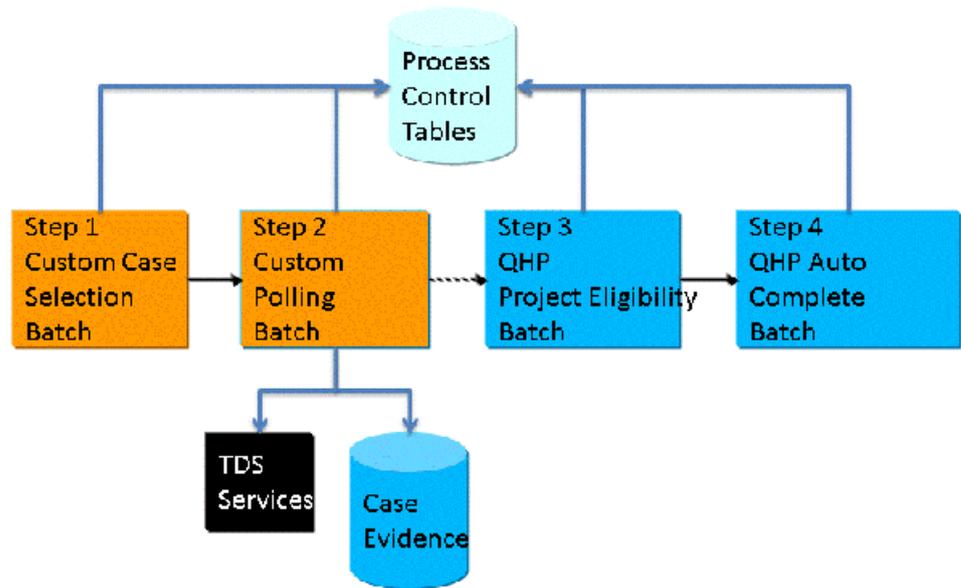
Figure 4: QHP annual renewal batch process flow

- **Custom Case Selection and Custom Polling batch**

  These custom batch processes should identify cases to be processed for each QHP run. At a minimum, they should identify the cases to be processed and write this information to the PDMRunCaseControl table. The custom batch processes should also:

  - Poll for information from external systems
  - Populate the annual renewal process control tables

- Populate evidence tables with the retrieved case information
- **QHP Project Eligibility batch**

  The QHPProjectedEligibility batch process performs projected eligibility for every case in the run. If the projection results in no change in eligibility, then the renewal is completed automatically and the case is renewed for a further 12 months.

  For each case successfully processed, the corresponding record on the process control table is updated with:

  - ID of the projected eligibility record
  - ID of the generated notice record
  - Status is set to Notice Generated (RCCS26002).
- **QHP Auto Complete batch**

  The QHPProcessAutoCompletions batch process completes the QHP annual renewals after a configured period.

  A citizen can contest the updated evidence by the end of the specified period via their citizen account or in person through a caseworker. If the updated evidence is not contested, the batch process redetermines eligibility with the evidence in the projected eligibility notice.

  For each case that is successfully processed, the corresponding record on the process control table is updated with the status set to 'Automatically Completed (RCCS26003)'.

Running the annual renewals for QHP batch processes

Complete the following tasks to run the annual renewals for QHP batch processes. The annual renewals for QHPs process is split between two batch processes that you must schedule independently

Before you begin

> **Important:** Ensure that the current client information was retrieved from external systems through your custom TDS service and was written to the integrated cases as external evidence by using the PDMEvidenceMaintenance API.

Before running an annual renewals run, ensure that no other period data matching or annual renewals runs are in process. It is recommended that period data matching or annual renewal runs do not overlap.

About this task

The first batch process runs projected eligibility and calls the custom generate notices implementation to send the appropriate citizen notices.

The second batch process picks up cases based on the notice generation date, completes the processing, and notifies the client of any changes to their coverage.

You must run this batch process approximately 31 days from the date that the notice generation occurred. The number of days is dependent on the date the notice was sent to the user. If you run

the first batch process late at night, some notices can be generated a day later than others. By default, the batch process picks up cases where the notification was sent 30+1 days prior.

Procedure

1. Create or identify a run configuration in the Cúram Administration application.
2. Run the QHPProjectdEligibility batch process, passing in the run ID as a parameter. For example, to run this batch at the command line:`ant -f app_batchlauncher.sample.xml -Dbatch.username=superuser - Dbatch.program= curam.hcr.pdm.sl.intf.QHPProjectedEligibility.process -Dbatch.parameters=runID=QHP_Q1_14`
3. Run the QHPProcessAutoCompletions batch process, passing in the run ID as a parameter.

QHPProjectedEligibility batch process

This batch process is used to run the projected eligibility and notice generation processes for a specified list of cases as part of annual renewals for Qualified Health Plans.

Inserting evidence by using the PDMEvidenceMaintanence API automatically ensures that the case is processed by annual renewal projected eligibility and notice generation processes. There can also be cases for which no evidence was received, but still must be processed as part of an annual renewal.

Class and method

curam.hcr.pdm.sl.intf.QHPProjectedEligibility.process

Parameters

Parameter	Description	Default value
runID	Mandatory. The run ID value that you specified in the periodic data matching run configuration in the Administration application.	A different value is required for each run.
instanceID	Optional. An instance ID can be used by the batch infrastructure to stop or restart a batch process.	BPN26008
processingDate	Optional.	The current date.
runInstanceID	Do not set. An internal batch field for communicating the Run Instance ID between the batch main and batch stream components.	Not applicable

QHPProcessAutoCompletions batch process

This batch process completes annual renewals for QHP after a configurable period. If a citizen has not submitted an annual renewal at the end of the specified period, either online or through a caseworker, this batch process redetermines their eligibility as per the evidence in the projected eligibility notice.

You must schedule the batch process to run after a period of curam.citizenaccount.annualrenewal.expiry.days +1 days. All cases that are in the "Notice

Generated" state for that period are processed. By default, the value of automatic-completion curam.citizenaccount.annualrenewal.expiry.days is 30 days.

Class and method

curam.hcr.pdm.sl.intf.QHPProcessAutoCompletions.process

Parameters

Parameter	Description	Default value
runID	Mandatory. The run ID value that you specified in the periodic data matching run configuration in the Administration application.	A different value is required for each run.
instanceID	Optional. An instance ID can be used by the batch infrastructure to stop or restart a batch process.	BPN26008
processingDate	Optional.	The current date.
runInstanceID	Do not set. An internal batch field for communicating the Run Instance ID between the batch main and batch stream components.	Not applicable

Overview of the Medicaid annual renewal batch process flow

The Medicaid annual batch flow consists of a number of discrete but interrelated batch processes. There are three steps to the complete flow.

To complete the end to end Medicaid annual renewal process in batch, you must implement a number of custom batch jobs that work with the provided MedicaidProcessAnnualRenewals batch job. An overview of the Medicaid batch flow is shown in the following figure. Steps 1 and 2 are custom batch jobs that you must implement.
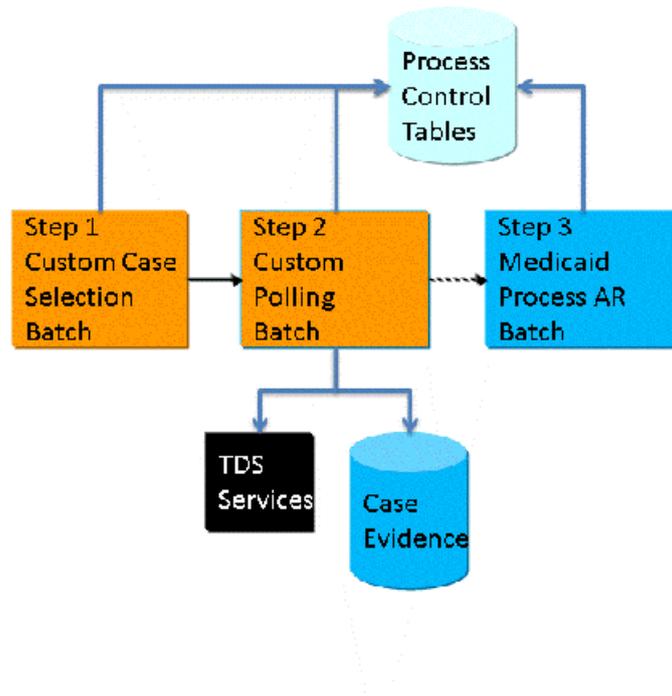
Figure 5: Medicaid annual renewal batch process flow

- **Custom Case Selection and Custom Polling batch**

    These custom batch processes should identify cases to be processed for each Medicaid run. At a minimum, they should identify the cases to be processed and write this information to the PDMRunCaseControl table. the custom batch processes should also:

    - Poll for information from external systems
    - Populate the annual renewal process control tables

- Populate evidence tables with the retrieved case information
- **Medicaid Process AR batch**
  The MedicaidProcessAnnualRenewals batch process performs projected eligibility for every case in the run. If the projection results in no change in eligibility, then the renewal is completed automatically and the case is renewed for a further 12 months.
  To summarize, the batch does the following:

  - Runs projected eligibility
  - Calls the custom generate notices implementation to send the appropriate citizen notices
  - Automatically renews Medicaid for eligible citizens where eligibility is unchanged
  - For each case that is successfully processed, updates the corresponding record on the process control table to 'Automatically Completed (RCCS26003)'.

Running the annual renewals for Medicaid batch process

Complete the following tasks to run the annual renewals for Medicaid batch. This batch process runs projected eligibility, calls the custom generate notices implementation to send the appropriate citizen notices, and automatically renews Medicaid for eligible citizens where eligibility is unchanged.

Before you begin

> **Important:** Ensure that the current client information was retrieved from external systems through your custom TDS service and was written to the integrated cases as external evidence by using the PDMEvidenceMaintenance API.

Before running an annual renewals run, ensure that no other period data matching or annual renewals runs are in process. It is recommended that period data matching or annual renewal runs do not overlap.

About this task

This batch process renews Medicaid cases only where the eligibility is unchanged. The batch process renews Medicaid cases and progresses the run case control record to the Automatically Completed state. For all other cases, the case run control record is set to failed. As part of your implementation, you must decide what further custom processing you can apply to these cases.

Procedure

1. Create or identify a run configuration in the Cúram Administration application.
2. Run the MedicaidProcessAnnualRenewals batch process, passing in the run ID as a parameter. For example, to run this batch at the command line:`ant -f app_batchlauncher.sample.xml -Dbatch.username=superuser - Dbatch.program= curam.hcr.pdm.sl.intf.MedicaidProcessAnnualRenewals.process – Dbatch.parameters=runID=MA_2014`

MedicaidProcessAnnualRenewals batch process

This batch process runs projected eligibility, calls the custom generate notices implementation to send the appropriate citizen notices, and automatically renews Medicaid for eligible citizens where eligibility is unchanged.

Class and method

curam.hcr.pdm.sl.intf.MedicaidProcessAnnualRenewals.process

Parameters

Parameter	Description	Default value
runID	Mandatory. The run ID value that you specified in the periodic data matching run configuration in the Administration application.	A different value is required for each run.
instanceID	Optional. An optional instance ID that used by the batch infrastructure to stop or restart a batch process.	BPN26008
processingDate	Optional.	The current date.
runInstanceID	Do not set. An internal batch field for communicating the Run Instance ID between the batch main and batch stream components.	Not applicable

Overview of the CHIP annual renewal batch process flow

The CHIP annual batch flow consists of a number of discrete but interrelated batch processes. There are three steps to the complete flow.

To complete the end to end CHIP annual renewal process in batch, you must implement a number of custom batch jobs that work with the provided CHIPProcessAnnualRenewals batch job. An overview of the CHIP batch flow is shown in the following figure. Steps 1 and 2 are custom batch jobs that you must implement.
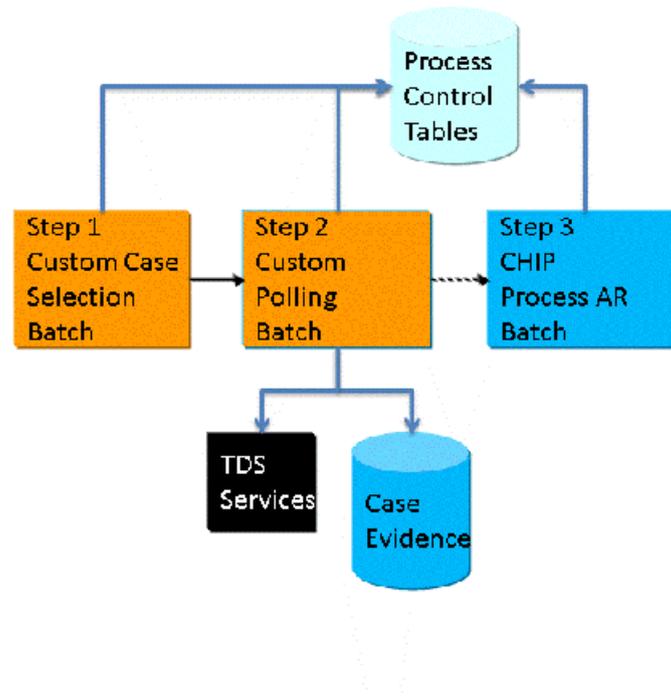
Figure 6: CHIP annual renewal batch process flow

- **Custom Case Selection and Custom Polling batch**

  These custom batch processes should identify cases to be processed for each CHIP run. At a minimum, they should identify the cases to be processed and write this information to the PDMRunCaseControl table. the custom batch processes should also:

  - Poll for information from external systems
  - Populate the annual renewal process control tables

- Populate evidence tables with the retrieved case information
- **CHIP Process AR batch**
  The CHIPProcessAnnualRenewals batch process performs projected eligibility for every case in the run. If the projection results in no change in eligibility, then the renewal is completed automatically and the case is renewed for a further 12 months.
  To summarize, the batch does the following:

  - Runs projected eligibility
  - Calls the custom generate notices implementation to send the appropriate citizen notices
  - Automatically renews CHIP for eligible citizens where eligibility is unchanged
  - For each case that is successfully processed, updates the corresponding record on the process control table to 'Automatically Completed (RCCS26003)'.

Running the CHIP annual renewals batch process

Complete the following steps to run annual renewals for the Children's Health Insurance Program (CHIP) batch process. This batch process runs projected eligibility, calls the custom generate notices implementation to send the appropriate citizen notices, and automatically renews CHIP for eligible citizens.

Before you begin

> **Important:** Ensure that the current client information was retrieved from external systems through your custom TDS service and was written to the integrated cases as external evidence by using the PDMEvidenceMaintenance API.

Before running an annual renewals run, ensure that no other period data matching or annual renewals runs are in process. It is recommended that period data matching or annual renewal runs do not overlap.

About this task

This batch process renews CHIP cases only where the eligibility is unchanged. The batch process renews CHIP cases and progresses the run case control record to the Automatically Completed state. For all other cases, the case run control record is set to failed. You can use an API to query this table for the list of non-renewed cases. As part of your implementation, you must decide what further custom processing you can apply to these cases.

Procedure

1. Create or identify a run configuration in the Cúram Administration application.
2. Run the CHIPProcessAnnualRenewals batch process, passing in the run ID as a parameter. For example, `ant -f app_batchlauncher.sample.xml - Dbatch.username=superuser - Dbatch.program= curam.hcr.pdm.sl.intf. ChipProcessAnnualRenewals.process -Dbatch.parameters=runID=CHIP_2014`

CHIPProcessAnnualRenewals batch process

This batch process runs projected eligibility, calls the custom generate notices implementation to send the appropriate citizen notices, and automatically renews CHIP for eligible citizens where eligibility is unchanged.

Class and method

curam.hcr.pdm.sl.intf.CHIPProcessAnnualRenewals.process

Parameters

Parameter	Description	Default value
runID	Mandatory. The run ID value that you specified in the periodic data matching run configuration in the Cúram Administration application.	A different value is required for each run.
instanceID	Optional. An optional instance ID that used by the batch infrastructure to stop or restart a batch process.	BPN26008
processingDate	Optional.	The current date.
runInstanceID	Do not set. An internal batch field for communicating the Run Instance ID between the batch main and batch stream components.	Not applicable

12.11 Triaging periodic data matching and annual renewal batch process errors

There are three main reasons why an error occurs in a PDM and AR batch process. The batch process can fail completely, an error can occur processing an individual Annual Renewal or PDM record, or an error can occur in the DB-to-JMS processing.

- **The batch fails completely**
  The batch launcher returns an integer value to the operating system when it completes a batch process. A non-zero return code is returned when an error occurs. Consult the logs for the details of the batch process error.
- **Error occurs processing individual Annual Renewal or PDM record**
  Errors can occur even when the batch process completes successfully. These errors can be caused by a number of reasons. PDM and AR batch processes write to a PDMRunCaseControlFailure failure table when a case cannot be processed during a batch job. You can use the PDMRunCaseControlManager API to build batch process error reports.
- **Error occurs in DB-to-JMS processing**
  DB-to-JMS errors can occur after the batch process is finished.

Checking for batch processing errors and reprocessing failed cases

After you run periodic data matching or annual renewal batch processes, you must identify any batch processing errors and reprocess any failed cases. Reprocessing and following up on failed cases is a custom development task

About this task

You can use the PDMRunCaseControlManager API to help you with tasks related to the following 4 main error processing scenarios. Typically, you can complete these tasks by implementing a custom batch or batch-streaming process.

- Capturing a list of cases that failed to process for reporting purposes.
- Reviewing the reasons for technical case processing failures.
- Resetting cases that failed for technical reasons so they can be reprocessed when the issues are resolved.
- Listing the cases that failed for a business reason to follow up with a caseworker.

For more information about the PDMRunCaseControlManager API, see the Javadoc for the API.

The procedure for implementing a batch-streaming process involves 2 main steps:

1. Determining the list of cases to review.
2. Determining what work you need to do on each case.

Procedure

1. Determine the list of cases to review. You can use the default APIs to determine the list of cases as follows.

   a) The list of all failed cases:

```
final BatchStreamHelper batchStreamHelper = new BatchStreamHelper(); batchStreamHelper.setStartTime();
    batchStreamHelper.runChunkMain(key.instanceID, key, <YourBatchMainWrapper>,
 pdmRunCaseControlManager.listCasesByRunIDAndState(key.runID,
PDMRUNCASECONTROLSTATUSEntry.FAILURE, chunkMainParameters, <YourBatchStreamWrapper>);
```

   b) The list of all cases that failed for a technical reason:

```
final BatchStreamHelper batchStreamHelper = new BatchStreamHelper(); batchStreamHelper.setStartTime();
    batchStreamHelper.runChunkMain(key.instanceID, key, <YourBatchMainWrapper>,
    pdmRunCaseControlManager.listFailedCasesByRunIDAndFailureType(key.runID,
 PDMRUNCASEFAILURETYPECODEEntry.TECHNICAL),
chunkMainParameters, <YourBatchStreamWrapper>);
```

   c) The list of all cases that failed for a business reason:

```
final BatchStreamHelper batchStreamHelper = new BatchStreamHelper(); batchStreamHelper.setStartTime();
    batchStreamHelper.runChunkMain(key.instanceID, key, <YourBatchMainWrapper>,
    pdmRunCaseControlManager.listFailedCasesByRunIDAndFailureType(key.runID,
 PDMRUNCASEFAILURETYPECODEEntry.BUSINESS),
chunkMainParameters, <YourBatchStreamWrapper>);
```

2. Determine the work that you need to do on each case. The PDMRunCaseControlManager APIs helps you to complete the following tasks:

a) Reviewing the reasons for technical case processing failures.

```
final PDMRunCaseControlExt pdmRunCaseControlExt = pdmRunCaseControlManager.getCase(key.runID,
 batchProcessingID.recordID)
final PDMRunCaseControlFailureExt failureDetails = pdmRunCaseControlExt.getCurrentFailureDetails();
   log.logFailedCaseAndFailureDetails(pdmRunCaseControlExt, failureDetails.getDateTime(),
 failureDetails.getReasonCode(),
failureDetails.getMessage(), failureDetails.getDetails());
```

b) Resetting cases that failed for technical reasons so they can be reprocessed when the issues are resolved.

```
final PDMRunCaseControlExt pdmRunCaseControlExt = pdmRunCaseControlManager.getCase(key.runID,
 batchProcessingID.recordID);
pdmRunCaseControlExt.resetCase();
```

c) Listing the cases that failed for a business reason to follow up with a caseworker.

```
final PDMRunCaseControlExt pdmRunCaseControlExt = pdmRunCaseControlManager.getCase(key.runID,
 batchProcessingID.recordID);
final PDMRunCaseControlFailureExt failureDetails = pdmRunCaseControlExt.getCurrentFailureDetails();
if (failureDetails.getDateTime().before(<TimeLimitForFollowup>))
 { log.sendCaseWorkerAndSupervisorFollowupNote(pdmRunCaseControlExt); }
```

Identifying Medicaid or CHIP cases that were not automatically renewed

After you run annual renewals for Medicaid or CHIP, you must identify each of the cases that were not automatically renewed and reprocess the cases. Reprocessing and following up on failed cases is a custom development task

About this task

Medicaid or CHIP cases typically cannot be renewed for one of two reasons:

- The citizen is ineligible. Someone on the case is not eligible for Medicaid or CHIP for the next 12 months as determined by their projected eligibility.
- A technical problem prevented the determination of projected eligibility, typically if corrupted or bad data cannot be constructed as evidence.

You can use the PDMRunCaseControlManager API to help you to identify the cases. For more information about the PDMRunCaseControlManager API, see the Javadoc for the API.

Diagnosing PDM and AR batch run failures

All PDM and AR batch processes write to a PDMRunCaseControlFailure failure table when a case cannot be processed during a batch job. You can also see the list of cases that failed by looking at the status of the PDMRunCaseControl table.

To find all the failed cases for a run, use the following query:

```
SELECT
 caseID, runCaseControlID
FROM
 PDMRunCaseControl
WHERE
 runID = <the value of the RunID>
AND
status = 'RCCS26030';
```

To find the details of a specific failure:

```
SELECT
 reasonCode, message
FROM
 PDMRunCaseControlFailure
WHERE
 runCaseControlID = <the value returned from the previous selection>
```

Reason codes provide context to issues entered on the PDMRunCaseControlFailure table. These codes are listed in the PDMRunCaseFailureReason code table.

Table 35: PDMRunCaseControlFailure codetable

Code	Description (en locale)	Java Identifier
FRC001	Projected eligibility failure	PROJECTEDELIGIBILTY
FRC002	Projected comparison failure	ELIGIBILTYCOMPARISON
FRC003	Evidence copy failure	EVIDENCECOPY
FRC004	Evidence activation failure	EVIDENCEACTIVATION
FRC005	Outstanding verifications failure	OUTSTANDINGVERIFICATIONS
FRC006	Notice send failure	NOTICESEND
FRC007	Primary client or tax filer deceased	PRIMARYCLIENTORTAXFILERDECEASED
FRC020	Unexpected exception failure	UNKNOWNCAUSE

PDM and AR batch job failure reasons

Describes the PDM and AR batch job failure reasons and codes as listed in the PDMRunCaseControlFailure code table.

- **PROJECTEDELIGIBILTY (FRC001)**

  Raised when corrupted evidence is found. This error occurs when the projected eligibility batch processes try to construct the required rule objects from external evidence. This error does not occur if the evidence is validated before it is inserted into the PDM control tables as part of the polling and retrieval of data.

  To recover from this error, validate and correct the evidence to make sure that it is well-formed. The case control record can then be reset by the PDMRunCaseControlManager API. When the error is corrected, it is picked up by the next projected eligibility batch run.

- **NOTICESEND (FRC006)**

  Raised by QHPProjectedEligibility or PDMProjectedEligibility when an error occurs during projected eligibility processing when the notice is generated. This error can occur when the template for the notice is not available or some issue exists with the data that is being merged with the template.

  When the issue is corrected, the failed case control records can be reset by the PDMRunCaseControlManager API. You can then run the batches again to pick up the corrections.

- **UNKNOWNCAUSE (FRC020)**

  Raised when a general error of a technical nature occurs and the exact cause cannot be determined. The failure table contains a stack trace of the error to aid troubleshooting. When the cause of the error is corrected, the records can be reset and processed by the batch that caused the error to occur.

The following batch failures are processed as tasks that are routed to the caseworker to correct the underlying issues. The caseworker can then manually complete the renewal or PDM process, or the failure can be reset by a developer and reprocessed again.

- **ELIGIBILTYCOMPARISON (FRC002)**

  Raised by the MedicaidProcessAnnualRenewals or CHIPProcessAnnualRenewals batch processes when the projected eligibility differs from the existing eligibility decision on the case. When this error occurs, the batch cannot complete the renewal automatically. These failures must be corrected before further processing can continue.

- **EVIDENCECOPY (FRC003)**

  Raised by the QHPProcessAutoCompletions batch process when an error occurs following a renewal where the caseworker **Complete**action or the citizen **Sign and submit** actions have been used.

  This error is usually raised when there is pre-existing in-edit evidence for the same type as that being copied by the batch process. This error must be resolved by the caseworker before completing the renewal manually. The caseworker must decide on the correct evidence to use and ensure that it is correctly recorded.

- **EVIDENCEACTIVATION (FRC004)**

  Raised by the QHPProcessAutoCompletions and PDMProcessAutoCompletions batch processes when external evidence is copied successfully with no raised verifications, but the new internal evidence activation is unsuccessful.

- **OUTSTANDINGVERIFICATIONS (FRC005)**

  Raised by the QHPProcessAutoCompletions and PDMProcessAutoCompletions batch process when external evidence is copied successfully but there are outstanding verifications for the new internal evidence. When the caseworker resolves the verifications, the renewal is completed manually by the caseworker.

- **PRIMARYCLIENTORTAXFILERDECEASED (FRC007)**

  Raised by PDMProjectedEligibility during projected eligibility processing if the external evidence indicates that the primary client or tax filer on the case is deceased. This error stops a projected eligibility from being carried out and raises a task to the caseworker to select a new primary client. The caseworker then completes the renewal manually.

12.12 Extracting rule objects snapshots to SessionDoc style HTML

You can extract the rule objects snapshot for a program group determination to SessionDoc style HTML by running a build target from the runtime directory for the rule objects snapshot of a program group determination.

About this task

This build target extracts an active or superseded program group determination by referencing the *CREOLEPROGGRPDETERMINATIONID* field from the PROGRAMGROUPDETERMINATION table. You can also extract a projected eligibility program group determination by referencing the *CREOLEPROGGRPDETERMINATIONID* field from the PROGGRPPROJECTEDELIGIBILITY table.

The following input parameters are used by the `creole.extract.programgroupruleobjects` build target:

- `outputDir` The folder where the HTML output pages are placed by the tool. Ensure the folder is writable. This is a mandatory parameter.
- `programGroupDeterminationID` The unique identifier of the program group determination for which you are extracting the rule objects snapshot. This is the *CreoleProgGrpDetermination.creoleProgGrpDeterminationId* field on the database. This is a mandatory parameter.

Procedure

Run the following command from the runtime directory: **build creole.extract.programruleobjects -DoutputDir={outputDir value} -DprogramGroupDeterminationID={programGroupDetermainationID value}**

12.13 Customizing periodic data matching and annual renewals

Use the following information to help you to customize your periodic data matching or annual renewals implementation to your requirements.

Related tasks

Customizing inconsistency period processing on page 153

Inconsistency period processing allows a caseworker to give a client a reasonable opportunity period to provide outstanding verifications for evidence that requires verification. Cases can proceed during that period as if outstanding verifications were provided. The default inconsistency period processing infrastructure consists of a batch process, a workflow, and the inconsistency period processing APIs. You can create a custom event handler to customize the default inconsistency period processing.

Customizing the storage of program group determinations

From version 6.0.5.5 interim fix 2 onwards, all program group determinations are stored in the database by default. Over time the number of determinations can become significant and increase the size of the database table. You can use the provided hook point to suppress the storage of identical program group determinations to reduce the size of the database table.

About this task

If you create an implementation that can identify if the new eligibility determination is the same as the current active eligibility determination, you can use the hook to suppress the storage of the new determination. Suppressing the storage of the determination means that no information is inserted into the PROGRAMGROUPDETERMINATION, CREOLEPROGGRPDETERMINATION and CREOLEPROGGRPDETERDATA entities.

> **Note:** You must ensure that your implementation correctly stores a determination when required. For example, if an updated determination is not stored, then future comparisons for this value can be affected.

Procedure

1. Create a class that implements curam.healthcare.sl.impl.ProgramGroupDeterminationStorageHook, which contains the single ProgramGroupDeterminationStorageHook.storeDetermination(ProgramGroupDeterminationDetails) method with the following options.

   - Returns true to indicate to store the program group determination because it is different from the current active program group determination.
   - Returns false to indicate not to store the program group determination because it is considered equivalent to the current active program group determination.

2. Add a Guice link binding to the existing custom module class, linking the implementation that you created for curam.healthcare.sl.impl.ProgramGroupDeterminationStorageHook to the ProgramGroupDeterminationStorageHook interface.
   For example:

```
protected void configure() {

    // Existing bindings

    // Guice Link binding linking custom implementation of ProgramGroupDeterminationStorageHook
    //(e.g. CustomProgramGroupDeterminationStorageHookImpl) to
    // the ProgramGroupDeterminationStorageHook interface

 bind(ProgramGroupDeterminationStorageHook.class).to(CustomProgramGroupDeterminationStorageHookImpl.class);
}
```

Related tasks

Storing all existing program group determinations on page 97
Before version 6.0.5.5 interim fix 2, program group determinations were not saved in the database. If you upgrade from an earlier version, you must run the BulkRunProgramGroupEligibility batch process on your system before you run any of the periodic data matching or annual renewals batch processes. The BulkRunProgramGroupEligibility batch process identifies and stores all of the current program

group determinations in your system. This once-off task for each system captures information that is required for projected eligibility comparisons.

Customizing projected eligibility for periodic data matching and annual renewals

Complete the following tasks to customize the default projected eligibility implementation to include extra external evidence types. By default, the Death Status, Minimum Essential Coverage, Income Details, and Annual Tax Return external evidence types are supported.

Procedure

1. Update your TDS services implementation to handle the new evidence types.
2. Configure dynamic evidence entities for the new external evidence types.
3. Customize the projected eligibility evidence handlers to handle the new evidence types.
4. Where necessary, customize the citizen account to include the new evidence types.

Customizing projected eligibility evidence handlers

Use this information to customize evidence handlers if you want to include an extra external data evidence type in your eligibility projections. You can also modify or replace the default evidence handler mapping of external data to internal rule objects.

Before you begin

> **Important:** Implementing a new evidence handler or replacing an existing handler is a non-trivial task. An incorrectly implemented error handler that encounters errors or incorrectly constructs rule objects can cause the projected eligibility rules to encounter errors at run time or can cause incorrect eligibility determinations. It is important to thoroughly test all custom evidence handlers before deployment in a live environment.

Projected eligibility

Projected eligibility is the process where the HCR Program Group Rules are run in a mode that uses the active data on the case, supplemented by data that is obtained from external sources, to determine and inform a citizen of the affect that the external data would have if it was applied to their case.

Depending on the projection type, the projection can be for the current period or for an eligibility period in the future, such as the next enrollment period.

> **Important:** External evidence must not be directly referenced by eligibility and entitlement rule sets as this will lead to case redeterminations each time external evidence is added to a case. Redetermining cases as part of evidence polling might lead to performance issues and removes the ability to generate eligibility projections that use the external data.

Projected eligibility evidence handlers

Projected eligibility evidence handlers enable HCR rules to use external data in eligibility projections. These evidence handlers convert external data into rule objects that are used when the

rules run. Then, the data that was sourced from external sources and added as evidence to the case can be used to determine eligibility and entitlement in the case.

You can replace or disable the default evidence handlers. You can also add custom evidence handlers for external evidence types that not supported by default.

You use the ProjectedEligibilityEvidenceHandler API to implement a projected eligibility evidence handler. For more information about the ProjectedEligibilityEvidenceHandler API, see the Javadoc for the API.

External evidence

External evidence is evidence that is not provided by the citizen but acquired from trusted data sources, such as the Social Security Administration, or the Internal Revenue Service. It is applied to the integrated case to be used only in eligibility projections.

External evidence can be associated with verifications that ensure that the values are compatible with client-attested values. For example, client-reported yearly income must be reasonably compatible with the external evidence type of 'Annual Tax Return', otherwise the client must provide proof that the evidence from the trusted data source is incorrect.

> **Important:** External evidence must not be directly referenced by eligibility and entitlement rule sets as this causes case redeterminations each time external evidence is added to a case. Redetermining cases as part of evidence polling might lead to performance issues, and removes the ability to generate eligibility projections that use the external data.

Implementing a new evidence handler

Use this information to customize evidence handlers if you want to include an extra external data evidence type in your eligibility projections. You can also modify or replace the default evidence handler mapping of external data to internal rule objects.

What to do next

> **Important:** Implementing a new evidence handler or replacing an existing handler is a non-trivial task. An incorrectly implemented error handler that encounters errors or incorrectly constructs rule objects can cause the projected eligibility rules to encounter errors at run time or can cause incorrect eligibility determinations. It is important to thoroughly test all custom evidence handlers before deployment in a live environment.

Identifying rule classes for the evidence handler

Complete the following analysis to identify which rule classes must be returned by the defineInMemoryRuleObjects() method.

About this task

Typically, but not always, an evidence handler only creates rule objects of a single type.

You must follow the chain from External Evidence to Data Rule. The link between External Evidence and Rule Class is a multi-step chain as shown:

External Evidence > **Evidence** > **Evidence Propagator** > **Data Rule Set** > **Data Rule Class** > **Rule Objects of the Rule Class type**

The following procedure illustrates how to navigate the chain by using an example. To determine which rule class must be returned for a new evidence type, you must repeat this procedure for each new handler that you create.

Procedure

1. Identify the evidence type associated with the external evidence type.

   A business analyst should be able to identify the client attested evidence type which is associated with an external evidence type.

   Annual Tax Return data that is obtained from external data sources is stored on the 'Annual Tax Return' external evidence type. This external evidence maps to the 'Income' evidence on the case that is used by the HCR eligibility rules. For your custom handler, you must determine to which evidence type the external evidence is mapped before proceeding to the next step in the chain analysis.

   > **Note:** In some instances, multiple external evidence types map to the same evidence type, for example 'Income Details' and 'Annual Tax Return' both map to 'Income'. In this scenario, the same evidence handler converts the data from both external evidence types into rule objects.

2. Identify the evidence propagator associated with the evidence type.

   After you identify the evidence type, there should be only one Active Succession Propagator Configuration, type "ROPT2005", associated with this evidence type. The 'Income' evidence type code is "DET0026030". Use this evidence type code to search all of the propagation configuration files to identify the associated active succession set propagation configuration file. In this example, searching all of files on the system for the text `'<propagator type="ROPT2005">'` and `'<evidence type="DET0026030">'` yields the *IncomePropagatorConfiguration.xml* file with the following contents:

   ```xml
   <?xml version="1.0" encoding="UTF-8"?>
   <propagator type="ROPT2005">
     <configuration>
       <evidence type="DET0026030">
         <ruleset name="IncomeDataRuleSet"/>
       </evidence>
     </configuration>
   </propagator>
   ```

   For your custom handler, you must determine the propagator configuration file to which the evidence type is mapped before proceeding to the next step in the chain analysis.

   > **Note:** If the file search results in multiple active succession set propagator configuration files, then the evidence may be linked to multiple data rule sets. In this scenario, you must do the next step in the chain analysis for each propagator configuration that you identified.

3. Identify the data rule set associated with the evidence propagator.

After you identify the Evidence Propagator Configuration file, then this will contain the name of the data rule set. For example, the *IncomePropagatorConfiguration* file is associated with the IncomeDataRuleSet as indicated in the highlighted XML.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<propagator type="ROPT2005">
  <configuration>
    <evidence type="DET0026030">
      <ruleset name="IncomeDataRuleSet"/>
    </evidence>
  </configuration>
</propagator>
```

For your custom handler, you must determine the data rule set the evidence propagator is mapped to before proceeding to the next step in the chain analysis.

4. Identify the data rule class associated with the data rule set.

Once you have identified the Data Rule Set file, then examination of this file should allow you to locate the associated Data Rule Class. In our example, the *IncomeDataRuleSet.xml* rule set contains one rule class 'Income' as indicated in the highlighted XML.

```xml
<RuleSet name="IncomeDataRuleSet">
  <Class
    extends="ActiveSuccessionSet"
    extendsRuleSet="PropagatorRuleSet"
    name="Income"
  >
```

For your custom handler, you need to determine the rule class the data rule set is mapped to.

5. Determine the fully qualified rule class name.

Finally, the fully qualified rule class name is obtained by concatenating the rule set name with the rule class name. For example, the fully qualified rule class name is 'IncomeDataRuleSet.Income'. For your custom handler, you must determine the fully qualified rule class name.

This fully qualified rule class name is to be returned by the defineInMemoryRuleObjects() method when you create a custom evidence handler.

What to do next
The initial analysis is complete and you now have enough information to start implementation of the custom evidence handler.
External evidence to qualified rule class name mappings
The default mappings of external evidence to qualified rule class name.

Table 36: Default mappings of external evidence to qualified rule class name

External Evidence Type	Mapped Evidence Type	Mapped Data Rule Set	Mapped Rule Class Name	Fully Qualified Rule Class Name
Death Status	Application Details Birth and Death Details	HCRApplicantDataRuleSet PDCBirthAndDeathDataRuleSet	HCRApplicant PDCBirthAndDeath	HCRApplicantDataRuleSet.HCRApplicant PDCBirthAndDeathDataRuleSet.PDCBirthAndDeath
Minimum Essential Coverage	Benefit	HCRBenefitDataRuleSet	HCRBenefit	HCRBenefitDataRuleSet.HCRBenefit
Income Details	Income	IncomeDataRuleSet	Income	IncomeDataRuleSet.Income

External Evidence Type	Mapped Evidence Type	Mapped Data Rule Set	Mapped Rule Class Name	Fully Qualified Rule Class Name
Annual Tax Return	Income	IncomeDataRuleSet	Income	IncomeDataRuleSet.Income

Rule objects in projected eligibility

Rule objects are created before running rules. For active eligibility and entitlement determinations, a rule object converter reads underlying business tables to obtain the appropriate data and populate rule objects in memory. For Projected Eligibility, the normal rule object converter is bypassed and instead responsibility for creating all rule objects is delegated to the projected eligibility evidence handler for the rule class types that are specified by the defineInMemoryRuleClasses() method. These rule objects are created by the createRuleObject() method, which is implemented by each evidence handler.

Each evidence handler must replicate the work that is normally done by the rule object converter. Before you can start creating rule objects, you need to determine which rule object creation strategy the handler should use. The default evidence handlers use three different rule object creation strategies. However, you can choose to implement a new strategy.

Creating a custom evidence handler

Complete the following steps to create the custom evidence handler implementation.

Before you begin

Use the following naming convention for custom projected eligibility evidence handlers:

```
{Custom Identifier}{Evidence Name}ProjectedEligibilityEvidenceHandlerImpl
```

For example, you might call a custom version of an Income evidence handler CustomIncomeProjectedEligibilityEvidenceHandlerImpl.

Procedure

1. Using the identified the rule class for the rule objects that are created by the handler, implement the defineInMemoryRuleObjects() method.

The default income evidence handler implementation of the defineInMemoryRuleObjects()
method is shown. This method returns a list with containing the one rule class that is created
by the handler 'IncomeDataRuleSet.Income'

```
public class IncomeProjectedEligibilityEvidenceHandlerImpl implements
  ProjectedEligibilityEvidenceHandler {

public class SampleIncomeProjectedEligibilityEvidenceHandlerImpl implements
  ProjectedEligibilityEvidenceHandler {

  /**
   * {@inheritDoc}
   */
  @Override
  public Set<String> defineInMemoryRuleClasses(final CaseKey caseKey,
    final Session session,
    final EvidenceDescriptorDtlsList evidenceDescriptorDtlsList,
    final PROJECTEDELIGIBILITYTYPEEntry projectedEligibilityType) {

    return new HashSet<String>() {

      {
        add("IncomeDataRuleSet.Income");
      }
    };
}
```

2. To complete the implementation of the custom evidence handler, you must implement
   the createRuleObjects() method to create rule objects for projected eligibility. The
   createRuleObjects() method is defined as follows:

```
void createRuleObjects(final CaseKey caseKey, final Session session,
final EvidenceDescriptorDtlsList evidenceDescriptorDtlsList,
final PROJECTEDELIGIBILITYTYPEEntry projectedEligibilityType)
throws AppException, InformationalException;
```

Adding logging to custom evidence handlers
Logging can be added to custom evidence handlers by invoking the log() method on the
ProgramGroupProjectedEligiblityHelper class. Adding logging to custom evidence handlers can
help in the investigation of unexpected projected eligibility results.

Procedure

Use the following code to add logging to a custom evidence handler:

```
ProgramGroupProjectedEligibilityHelper helper =
  new ProgramGroupProjectedEligibilityHelper();
helper.log(Level.INFO, message);
```

Please refer to the ProgramGroupProjectedEligibilityHelper API for more information on the
log() method.

Implementing the creation of rule objects for projected eligibility
Completing the implementation of a custom evidence handler requires implementing the
createRuleObjects() method. This is by far the most complex step in creating a custom evidence
handler. The createRuleObject() method is responsible for creating all rule objects for the rule
class types that are specified by the defineInMemoryRuleClasses() method. Therefore business
logic is required by the createRuleObjects() to determine which data to use when constructing the
rule object.

About this task

There are many different sources of data or strategies that this method could use when creating the rule objects. For example the method could:

- Exclusively use only the active evidence on the case; not necessarily useful as this would result in the same decision as the current active decision.
- Exclusively use the external evidence on the case; what happens if there are missing fields on the external evidence that must be specified on the rule object?
- Use all the active evidence on the case and all the external evidence; what happens if this results in duplicated rule objects resulting in an ineligible decision because for example income was double counted?

When implementing the default evidence handlers, business analysts defined three different strategies that were required so that the evidence handlers used the correct data when constructing the rule objects. These are discussed in detail in the following topic as they may be applicable to custom evidence handlers. When implementing a new evidence handler, identifying the correct data to use when constructing the rule object is critical and will probably require business analyst input.

Rule object creation strategies

The default evidence handlers use three different rule object creation strategies in their createRuleObjects() implementations to correctly construct rule objects that represent the active client attested evidence and the external evidence. However, you can choose to implement a new strategy if you prefer.

- **Load active internal evidence and external evidence as rule objects**
  The first strategy is to create rule objects that represent all the active internal evidence and to supplement this evidence by creating additional rule objects for the external data. This strategy can be used when there is no overlap between evidence that is on the case and evidence that was sourced externally, in other words the external evidence is orthogonal to the evidence already used in the determination.
- **Load active internal evidence as rule objects, clone rule objects and modify attributes**
  The second strategy is to load rule objects that represent all the active internal evidence, however rather than creating rule objects for the external evidence, the loaded rule objects are cloned, which certain rule attribute values being replaced with the external data. Use this strategy if the external evidence attributes only maps to a subset of attributes on the internal evidence.
- **Create rule objects by overlaying active evidence with external evidence**
  This is the most complex rule object creation strategy because for those external evidence records that are deemed equivalent by a business-defined evidence-matching algorithm, you need to effective date or overlay the external evidence information with the active evidence. For equivalent evidence that needs to be effective dated, the rule objects that are created from the active evidence details are end dated while the external evidence records have a start date, which ensures the rule objects created are never processed by the rules for a common date. For active and external evidence records that are not deemed equivalent by the evidence matching algorithm, this strategy follows the 'Load active internal evidence and external evidence as rule objects' strategy.
  An example of this strategy is used in the default Income evidence handler. The income evidence handler merges active Income evidence with two external evidence types – Annual

Tax Return and Income Details to create rule objects that represent data from the three separate sources. An income merging rule object algorithm was developed to combine active income evidence with the external evidence. The algorithm takes the form:

- If active income evidence is equivalent to external evidence, based on an income type and participant match algorithm:

  - Load the income rule object, and clone and end-date income rule object.
  - Create a rule object by using the external evidence and start date, effectively

- If income is not equivalent to the external income:

  - Load the income rule object that represents active income evidence.
  - Create new rule objects that represent the external evidence.

Techniques for filtering evidence and creating rule objects

Each of the default rule object creation strategies uses the following techniques to filter evidence and create rule objects. You can use the same techniques in a custom handler.

Filtering external evidence by evidence type

When each evidence handler's createRuleObject() method is invoked by the projected eligibility manager, the method is passed a list of evidence descriptor records associated with the projection (EvidenceDescriptorDtlsList evidenceDescriptorDtlsList).

Procedure

Each handler should filter this list of evidence by evidence type and before converting the filtered list to rule objects as per the creation strategy being implemented. To filter the evidence the following code can be added to a custom handler:

```
// Filter external MEC evidence details
    final EvidenceDescriptorDtlsList filteredEvidenceDescriptorList =
      new EvidenceDescriptorDtlsList();

    filteredEvidenceDescriptorList.dtls
      .addAll(evidenceDescriptorDtlsList.dtls);

    // Filter input evidence details list by type
    org.apache.commons.collections.CollectionUtils.filter
                  (filteredEvidenceDescriptorList.dtls,
      new org.apache.commons.collections.Predicate() {

        @Override
        public boolean evaluate(final Object input) {

          return ((EvidenceDescriptorDtls) input).evidenceType
            .equals(CASEEVIDENCE.MEC);
          // update to evidence type associated with the custom handler
        }
      });
```

Creating rule objects from active evidence

Active internal evidence can be converted into rule objects by invoking the default rule object converter for the rule class.

About this task

Using the associated Data Rule Set and Data Rule Class identified in the 'Identifying rule classes for the evidence handler' task and specifying the case id in the search criteria, the converter will convert the active evidence on the case into rule objects. Once the existing rule objects have been loaded, they must be cloned, creating new rule objects in the current session that are used by the session when the rules execute.

Procedure

Add a code equivalent to the following example to the custom handler, updating the rule set and
rule class as appropriate.

```
// Load rule objects based on active internal evidence
   final RuleSet ruleSet =
     ruleSetManager.readRuleSet("HCRBenefitDataRuleSet");
   final RuleClass ruleClass = ruleSet.findClass("HCRBenefit");

   final SingleAttributeMatch ruleObjectSearchCriteria =
     new SingleAttributeMatch(ruleClass, "caseID", caseKey.caseID);

   final RuleClass soughtRuleClass = ruleObjectSearchCriteria.ruleClass();

   final RuleObjectConverter ruleObjectConverter =
     ruleClassConverterMapper.getRuleObjectConverter(soughtRuleClass);

   final List<RuleObject> ruleObjectList =
     ruleObjectConverter.convert(session, ruleObjectSearchCriteria);

   // Clone rule objects, creating new rule objects in the current session
   for (final RuleObject ruleObjectItem : ruleObjectList) {

     final RuleObject ruleObject = session.createRuleObject(ruleClass);

     for (final RuleAttribute ruleAttribute : ruleClass.allAttributes()) {

       ruleObject.getAttributeValue(ruleAttribute.name()).specifyValue(
         ruleObjectItem.getAttributeValue(ruleAttribute.name()).getValue());
     }
   }
```

Creating rule objects from external evidence

The filtered external evidence descriptor list can be used to create new rule objects in memory representing the external data using the following technique.

About this task

Modify the rule object attributes and external evidence attributes in the example to match the data rule class and external evidence that is associated with the custom handler.

```
// Looping the filtered evidence records,
// create new rule objects for external evidence

    for (final EvidenceDescriptorDtls
            activeEvidenceDtls : filteredEvidenceDescriptorList.dtls) {

      final RuleObject ruleObject = session.createRuleObject(ruleClass);

      ruleObject.getAttributeValue("caseID").specifyValue(caseKey.caseID);

      final EvidenceTypeKey evType = new EvidenceTypeKey();
      evType.evidenceType = activeEvidenceDtls.evidenceType;

      final EvidenceCaseKey evidenceCaseKey = new EvidenceCaseKey();
      evidenceCaseKey.evidenceKey.evidenceID = activeEvidenceDtls.relatedID;
      evidenceCaseKey.evidenceKey.evType = activeEvidenceDtls.evidenceType;

      // read external evidence
      final DynamicEvidenceObjectInf dynamicEvidenceObject =
        dynamicEvidenceMaintenanceExt.readEvidence(evidenceCaseKey);

      // set rule object values using external evidence details
      final long caseParticipantRoleID =
        (Long) dynamicEvidenceObject
          .getAttributeValue("caseParticipantRoleID");
      ruleObject.getAttributeValue("caseParticipantRoleID").specifyValue(
        caseParticipantRoleID);

      final Date startDate =
        (Date) dynamicEvidenceObject.getAttributeValue("startDate");
      ruleObject.getAttributeValue("startDate").specifyValue(startDate);

      // set additional fields on rule object as appropriate

      ruleObject.getAttributeValue("successionID").specifyValue(
        activeEvidenceDtls.successionID);

    }
```

When creating rule objects, you need to specify the value for all rule object attributes that have <specified/> derivation.

In the previous example, the HCRBenefit rule class has a number of attributes that have the 'specified' derivation. Each of these attribute values is set using the external evidence data. The following snippet of the HCRBenefit rule class shows the 'specified' derivation type for the 'startDate' rule attribute displayed in bold.

```
<Attribute name="startDate">
      <Annotations>
        <Label
          label-id="startDate"
          name="The date on which the unearned income commenced.&#x0D;"
        />
      </Annotations>
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>
```

When implementing a custom evidence handler you need to search the associated rule class for all attributes with the specified derivation and specify the attribute value on the rule object using data that is obtained from the external evidence record. For the startDate attribute, the following code reads the external evidence attribute value and specifies the rule object attribute value with this date.

```
final Date startDate =
        (Date) dynamicEvidenceObject.getAttributeValue("startDate");
    ruleObject.getAttributeValue("startDate").specifyValue(startDate);
```

Cloning and modifying active evidence rule objects

In some scenarios, business logic determines that an active evidence record is actually equivalent or partially equivalent to an external evidence record. Perhaps, by means of a partial field match. In this situation, business logic is needed to define an algorithm to create rule objects with attribute values that are specified either from the active evidence field values, or from external evidence field values.

About this task

For example, business logic might define an algorithm where the external evidence value is used to populate the rule attribute value unless there is no equivalent field on the external evidence. In that case, the active evidence value is used to populate the rule attribute value.

You might encounter this scenario if not all of the rule attribute fields that need to be specified map to fields on the external evidence type. In this situation, data from the active evidence needs to be used to supplement the rule object being created.

Procedure

1. Load active evidence as rule objects.
2. Loop the rule object list, for each existing rule object.
   a) Create a cloned rule object.
   b) Loop each attribute on the existing rule object, depending on the attribute name. You can use one of the following options:

   1. Specify a cloned rule object attribute value using the existing rule object value.
   2. Specify a cloned rule object attribute value using the external evidence.

   The decision per attribute name requires business logic to determine the correct source to be used when specifying the rule object attribute value.

Customizing an external evidence handler
Complete the following steps to customize or replace a default projected eligibility evidence handler with a custom implementation.

Before you begin
Create a new Projected Eligibility Evidence Handler implementation that replaces the default handler. You can implement an entirely new handler or use the default evidence handler as the starting point for your customization.

About this task

Assuming that you have successfully created a customized handler implementation, you must modify the Guice bindings so that the new evidence handler is used by projected eligibility rather than the default evidence handler.

Procedure

1. Modify the new custom evidence handler to extend the default evidence handler that you are replacing.
   For example, change

```
public class SampleIncomeProjectedEligibilityEvidenceHandlerImpl implements
  ProjectedEligibilityEvidenceHandler {
```

to

```
 public class SampleIncomeProjectedEligibilityEvidenceHandlerImpl extends
    IncomeProjectedEligibilityEvidenceHandlerImpl {
```

2. Create a module class, which creates a Guice link binding replacing the default evidence handler with the customized evidence handler or add the new link binding to an existing module class.
   For example, see this custom module, which contains a link binding to replace the default income evidence handler with a binding to a custom income handler.

```
/**
 * Contains modified projected eligibility evidence handler Guice bindings.
 */
@AccessLevel(AccessLevelType.EXTERNAL)
public class SampleProjectedEligibilityModule extends AbstractModule {

  /**
   * {@inheritDoc}
   */
  @Override
  protected void configure() {

    // Link binding replacing the default income projected evidence
    // handler with a custom income projected eligibility handler
    bind(IncomeProjectedEligibilityEvidenceHandlerImpl.class)
        .to(SampleIncomeProjectedEligibilityEvidenceHandlerImpl.class);
  }
```

3. If you created a new module class, update the *ModuleClassname.dmx* file to reference this new module.

Disabling an evidence handler

Complete the following steps to disable a projected eligibility evidence handler by replacing the default evidence handler with a custom evidence handler that has an empty implementation.

Procedure

1. Create a custom disabled evidence handler with no implementation that extends the default evidence handler that you are disabling.

For example, see this custom Income Projected Eligibility Evidence Handler that contains no implementation.

```
/**
 * {@inheritDoc}
 */
public final class SampleDisabledIncomeProjectedEligibilityEvidenceHandlerImpl extends
  IncomeProjectedEligibilityEvidenceHandlerImpl {

  @Override
  public Set<String> defineInMemoryRuleClasses(final CaseKey caseKey,
    final Session session,
    final EvidenceDescriptorDtlsList evidenceDescriptorDtlsList,
    final PROJECTEDELIGIBILITYTYPEEntry projectedEligibilityType) {

    // no rule objects will be created in memory by this handler
    return new HashSet<String>();

  }

  @Override
  public void createRuleObjects(final CaseKey caseKey, final Session session,
    final EvidenceDescriptorDtlsList evidenceDescriptorDtlsList,
    final PROJECTEDELIGIBILITYTYPEEntry projectedEligibilityType)
    throws AppException, InformationalException {

    // create no rule objects
  }

}
```

2. Create a module class, which creates a Guice link binding replacing the default evidence handler with the customized evidence handler or add the new link binding to an existing module class.
For example, see this custom module, which contains a link binding to replace the default income evidence handler with a custom disabled income handler.

```
/**
 * Contains disabled projected eligibility evidence handler Guice bindings.
 */
@AccessLevel(AccessLevelType.EXTERNAL)
public class SampleProjectedEligibilityModule extends AbstractModule {

  /**
   * {@inheritDoc}
   */
  @Override
  protected void configure() {

    // Link binding replacing the default income projected evidence handler
    // with a custom income projected eligibility handler that does nothing
    bind(IncomeProjectedEligibilityEvidenceHandlerImpl.class)
      .to(SampleDisabledIncomeProjectedEligibilityEvidenceHandlerImpl.class);
  }
```

3. If you created a new module class, update the *ModuleClassname.dmx* file to reference this new module.

Enabling projected eligibility logging

The default external evidence handlers can output log/trace information while being executed. This log output can be used to aid in the investigation of unexpected projected eligibility results in conjunction with other tools like SessionDoc. This log output is disabled by default.

About this task

Add Logging to custom external evidence handlers to aid investigation of projected eligibility issues.

Procedure

1. Log in to the Cúram application as a user with system administrator permissions.

2. Modify the value of the 'Projected Eligibility Message Logging Level' property to be 'trace_on'.

3. Publish changes.

Customizing the citizen account with new evidence types

After you add new external evidence types, you might want to update the citizen account to reflect any new citizen choices. You might also want to change the evidence types that you want customers to be able to contest.

12.14 Customizing the completion process for annual renewals

To customize or replace the default annual renewal completion implementation with a custom implementation, create a module class and update the *ModuleClassName.dmx* file to reference the new module.

Before you begin

Create an implementation of `CompleteAnnualRenewal` that extends `AbstractCompleteAnnualRenewal`. Then, modify the Guice bindings to use the custom implementation rather than the default implementation.

Procedure

1. Create a module class that creates a Guice link binding that replaces the default implementation with the customized implementation or add the new link binding to an existing module class. For example, the following custom module contains a link binding to replace the default binding with a custom implementation:

```
/**
 * Contains custom annual renewal completion Guice bindings.
 */
@AccessLevel(AccessLevelType.EXTERNAL)
public class SampleModule extends AbstractModule {

  /**
   * {@inheritDoc}
   */
  @Override
  protected void configure() {

    // Link binding replacing the default implementation with a custom
  implementation.
    binder().bind(AbstractCompleteAnnualRenewal.class).
       .to(CustomAbstractCompleteAnnualRenewal.class);

  }
```

2. If you created a new module class, update the *ModuleClassName.dmx* file to reference the new module.

12.15 Customizing the citizen account for periodic data matching and annual renewals

You can modify the citizen account to change the evidence that citizens can contest, and you can customize the periodic data matching and annual renewals messages that are displayed to the citizen.

Configuring contestable evidence types

By default, the Death Status and MEC evidence types are contestable for periodic data matching, and the income evidence type is contestable for annual renewals. A **Contest** button and a cluster containing the contestable evidence types are displayed in the citizen account, for each contestable evidence type. Complete the following steps to configure the contestable evidence types.

About this task

Contestable evidence types are specified in the curam.healthcare.pdm.contestable.evidences property in the "HEALTHCARE" section.

Procedure

1. Log in to the Cúram Administration application as a user with administrator permissions
2. Modify the curam.healthcare.pdm.contestable.evidences property to add or remove EvidenceTypes codes.

Adding contestable evidence types to the citizen account

After you add a contestable evidence type for periodic data matching and annual renewals, you must complete the following steps to enable citizens to contest the evidence from the citizen account.

About this task

By default, the Death Status and MEC evidence types are contestable for periodic data matching, and the income evidence type is contestable for annual renewals.

Procedure

1. Modify the webclient\components\HCROnline\CitizenAccount\lifeevents \CitizenAccount_arpdm.vim file to include clusters for the new evidence types.
2. Override the EJBServer\components\HCROnline\source\curam\healthcare\pdm\sl\impl \HealthCarePDMProcessingImpl.java Service Layer implementation to incorporate the changes. This service class implements the HealthCarePDMProcessing interface.
3. Modify the ContestableEvidenceDetailsList struct to capture the data for the new evidence.

Modifying periodic data matching home page messages

Complete the following steps to modify the default set of messages that can be displayed on the periodic data matching home page.

About this task

Procedure

1. To modify the messages in the DMX files:

    a) Modify the messages in the following DMX files:

        -   CITIZENHOMEMENUITEM
        -   LOCALIZABLETEXT
        -   TEXTTRANSLATION

    b) Override the HealthCarePDMProcessingImpl Java class or create and associate a new Java class.

2. To create messages in the DMX files:

    a) Create an entry in the CITIZENHOMEMENUITEM table.

    b) Create a Java class containing the implementation of the new business logic to determine on what condition the menu item message is displayed.

    c) Associate the new Java class file name to respective entry in the CITIZENHOMEMENUITEM table for the column "CLASSNAME".

3. To modify the messages in the *AnnualRenewal.properties* file:

    a) Modify the messages in the *data\initial\blob\prop \CitizenMessagesForPDM.properties* file.

    b) If required, override the PDMMessagesEventListener class.

    c) The messages are inserted into the PARTICIPANTMESSAGECONFIG table. Use the Administration Application to modify the messages.

Modifying periodic data matching My Updates page messages

Complete the following steps to modify the default set of messages that can be displayed on the default periodic data matching cluster on the **My Updates** page messages.

About this task

The messages that can be displayed on **My Updates** page are read from the *EJBServer \components\HCROnline\message\PDMUpdates.xml*. These messages are populated at run time in the Service layer implementation.

Procedure

1. Modify the messages in the *PDMUpdates.xml* file.

2. HealthCarePDMProcessingImpl is provided with a default implementation. You can create your own implementation by providing a new implementation for the HealthCarePDMProcessing interface.

Modifying annual renewals home page messages

Complete the following steps to modify the default set of messages that can be displayed on the annual renewals home page.

Procedure

1. To modify messages in the DMX files:
   a) Modify the messages in the following DMX files:
      - CITIZENHOMEMENUITEM
      - LOCALIZABLETEXT
      - TEXTTRANSLATION
   b) Override the AnnualRenewalMenuItemProducer Java class or create and associate a new Java class.

2. To create messages in the DMX files:
   a) Create an entry in the CITIZENHOMEMENUITEM table.
   b) Create a Java class containing the implementation of the new business logic to determine on what condition the menu item message is displayed.
   c) Associate the new Java class file name to respective entry in the CITIZENHOMEMENUITEM table for the column "CLASSNAME".

3. To modify the messages in the *AnnualRenewal.properties* file:
   a) Create a file to extend the AnnualRenewalMessageHelper interface. Extend the AnnualRenewalMessageHelperImpl and override the required APIs.

      The messages are inserted into the PARTICIPANTMESSAGECONFIG table.
   b) Modify the *CURAM_DIR\EJBServer\components\HCROnline\data \initial\blob\prop\AnnualRenewal.properties* file to add new messages or to modify the existing messages.

Modifying the annual renewals My Updates page

By default, the annual renewals implementation includes the CASEEVIDENCE.EXTERNALINCOMEDETAILS, CASEEVIDENCE.ANNUALTAXRETURN evidence types. Complete the following steps to modify the default annual renewal **My Updates** page to add new evidence types or to add new columns on the page.

Procedure

1. Implement the AnnualRenewalHelper interface. You can provide a new implementation or extend the AnnualRenewalHelperImpl. Inject the new implementation class.

2. Model the existing structs to accommodate the new evidence changes:
   a) Add a Boolean attribute in EvidenceTypeDetails struct to display a new cluster for the evidence type and ensure that the cluster is visible only when it contains records.
   b) Add a new attribute to hold the details of new evidence type. The data type for this list should be of type curam.citizenaccount.annualrenewal.facade.struct.EvidenceDetails.

c) Modify the EvidenceDetails struct to add the new attribute for display on the UI.

3. Update the *HCROnline\CitizenAccount\lifeevents* *\CitizenAccount_arpdm.vim* file to add the new columns or a new cluster.

12.16 Customizing evidence converters

Complete the following tasks to customize the default evidence mappings by modifying the evidence converters.

External evidence converters

External evidence converters create or modify the existing evidence on a case according to the external evidence on the case. Each evidence converter must implement the ExternalEvidenceConverter interface.

Registered external evidence converters are used by the automatic-completion batch processes to create evidence from polled external evidence. External evidence converters are responsible for converting one specific type of polled evidence into evidence on the case. An evidence converter cannot convert more than one polled evidence type. Depending on business requirements, an evidence converter either creates new evidence on a case, modifies existing evidence on the case or does both. A converter usually only creates or modifies evidence of one specific type. However, this is not a requirement. It is possible that a piece of polled evidence can map to several different evidence records on a case. The evidence converter converts one evidence record per invocation, so in situations where there are several different items of the same external polled evidence, the converter must be invoked for each polled evidence record.

Implementing a new external evidence converter

Complete the following steps to implement a new evidence converter by using an abstract helper class. The creation of custom external evidence converters is simplified by the inclusion of the abstract helper class, CommonExternalEvidenceConverter.

About this task

Several default converters are provided to convert polled evidences of types 'Death Status', 'MinimumEssentialCoverage', 'Annual Tax Return' and 'Income Details'. The ExternalEvidenceConverterManager invokes all registered converters that match the evidence types that were polled. You can implement and register additional external evidence converters for other types of polled evidence. Custom evidence converters can be created for the default polled evidence types or a new custom dynamic evidence type. All evidence that is created by the default evidence converters has an evidence change reason specified as 'Reported by External Party'.

While evidence converters usually create new evidence on a case, there are instances where they also need to modify existing evidence on the case. In some situations, the evidence that is created by the converter needs to replace or end date the evidence on the case. To achieve this an evidence converter needs to modify the existing evidence record before adding the new evidence.

In situations where there are multiple evidence records to be modified, business logic might need to be incorporated into the evidence converter to identify the correct evidence records to modify. In some situations, a converter might only need to modify existing evidence and not insert any new evidence records.

Procedure

1. Create a class that extends CommonExternalEvidenceConverter, specifying the source evidence type and the target evidence type.

```
public class SampleEvidenceTypeExternalEvidenceConverterImpl extends
CommonExternalEvidenceConverter<SOURCE_EVIDENCE_TYPE, TARGET_EVIDENCE_TYPE> {
```

In the example after this procedure, the source evidence type is a Dynamic evidence static evidence type AnnualTaxReturnDtls while the target is dynamic evidence.

2. The new class must implement all abstract methods that are defined in the CommonExternalEvidenceConverter class:

```
protected abstract CASEEVIDENCEEntry getSourceEvidenceType();
```

```
protected abstract CASEEVIDENCEEntry getTargetEvidenceType();
```

```
protected abstract TT getCreateEvidenceDetails(final ST sourceEvidenceObject,
final EvidenceDescriptorDtls descriptor);
```

```
protected abstract void mapModifyEvidenceDetails(
```

3. The new class must also override the following methods, which are defined in the CommonExternalEvidenceConverter class:

```
protected boolean shouldModifyEvidence(final ST sourceEvidenceObject,
final TT targetEvidenceObject)
```

```
protected boolean shouldCreateEvidence(final ST sourceEvidenceObject)
```

Example

The following example code snippet shows a sample evidence converter which converts polled Foreign Residency (static) evidence into Demographics (dynamic) evidence on the case.

> **Note:** This example is not a valid source/target evidence conversion mapping.

```java
/**
 * External Foreign Residency evidence converter into Demographics evidence.
 */
public class SampleForeignResidencyExternalEvidenceConverterImpl
  extends
  CommonExternalEvidenceConverter<ForeignResidencyDtls, DynamicEvidenceObject> {

  @Override
  protected CASEEVIDENCEEntry getTargetEvidenceType() {

    return CASEEVIDENCEEntry.FOREIGNRESIDENCY;
  }

  @Override
  protected CASEEVIDENCEEntry getSourceEvidenceType() {

    return CASEEVIDENCEEntry.DEMOGRAPHICS;
  }

  @Override
  protected DynamicEvidenceObject getCreateEvidenceDetails(
    final ForeignResidencyDtls sourceEvidenceObject,
    final EvidenceDescriptorDtls descriptor) throws AppException,
    InformationalException {

    final DynamicEvidenceObject targetEvidenceObject =
      new DynamicEvidenceObject(descriptor.caseID, descriptor.receivedDate,
        getTargetEvidenceType().getCode());

    targetEvidenceObject.setAttributeValue("concernRoleID",
      sourceEvidenceObject.concernRoleID);

    targetEvidenceObject.setAttributeValue("comments",
      sourceEvidenceObject.comments);

    // set remaining attributes on target evidence record

    return targetEvidenceObject;
  }

  @Override
  protected void mapModifyEvidenceDetails(
    final ForeignResidencyDtls sourceEvidenceObject,
    final DynamicEvidenceObject targetEvidenceObject) throws AppException,
    InformationalException {

    // End date target evidence
    targetEvidenceObject.setAttributeValue("endDate",
      sourceEvidenceObject.startDate.addDays(-1));
    targetEvidenceObject.setAttributeValue("source",
      HCINCOMESOURCE.EXTERNALSYSTEM);
  }

  @Override
  protected boolean shouldModifyEvidence(
    final ForeignResidencyDtls sourceEvidenceObject,
    final DynamicEvidenceObject targetEvidenceObject) throws AppException,
    InformationalException {

    // Add business logic to determine if existing evidence records on case
    // need to be modified.

    // Simple implementation, don't modify existing records
    return false;
  }

  @Override
  protected boolean shouldCreateEvidence(
    final ForeignResidencyDtls sourceEvidenceObject) throws AppException,
    InformationalException {

    // Add business logic to determine if new evidence record is required
    // on case.

    // Simple implementation, always add new record representing polled source
    // evidence.
    return true;
  }
```

Customizing an external evidence converter

Complete the following steps to customize or replace a default external evidence converter with a custom implementation.

Before you begin

Create a new External Evidence Converter implementation that replaces the default converter. You can implement an entirely new converter or extend the default evidence converter overriding specific methods as required in your customization.

About this task

Assuming that you have successfully created a customized converter implementation, you must modify the Guice bindings so that the new evidence converter is used by the external evidence converter manager rather than the default evidence converter.

Procedure

1. Modify the new custom evidence converter to extend the default evidence converter that you are replacing. For example, change:

```
public class SampleAnnualTaxReturnExternalEvidenceConverterImpl extends
        CommonExternalEvidenceConverter<AnnualTaxReturnDtls, DynamicEvidenceObject> {
```

to

```
public class SampleAnnualTaxReturnExternalEvidenceConverterImpl extends
        AnnualTaxReturnExternalEvidenceConverterImpl {
```

> **Note:** If you are replacing specific methods of an existing evidence converter, then your custom implementation already extends this default implementation.

2. Create a module class, which creates a Guice link binding replacing the default evidence converter with the customized evidence converter, or add the new link binding to an existing module class. For example, see this custom module, which contains a link binding to replace the default Annual Tax Return external evidence converter with a binding to a custom Annual Tax Return evidence converter.

```
/**
 * Contains modified external evidence converter Guice bindings.
 */
@AccessLevel(AccessLevelType.EXTERNAL)
public class SampleExternalEvidenceConverterModule extends AbstractModule {

    /**
     * {@inheritDoc}
     */
    @Override
    protected void configure() {

      // Link binding replacing the default Annual Tax Return external
      // external converter with a custom Annual Tax Return evidence converter.
      bind(AnnualTaxReturnExternalEvidenceConverterImpl.class)
        .to(SampleAnnualTaxReturnExternalEvidenceConverterImpl.class);
    }
```

3. If you created a new module class, update the *ModuleClassname.dmx* file to reference this new module.

Disabling an external evidence converter

Complete the following steps to disable an external evidence converter by replacing the default evidence converter with a custom evidence converter that performs no evidence conversion.

Procedure

1. Create a custom disabled evidence converter which extends the default evidence converter that you are disabling. The custom converter needs to override the convert() method so that no evidence is converted.

   For example, see this custom Annual Tax Return external evidence converter that contains an implementation which converts no evidence.

   ```
   /**
    * {@inheritDoc}
    */
   public class SampleDisabledAnnualTaxReturnExternalEvidenceConverterImpl extends
     AnnualTaxReturnExternalEvidenceConverterImpl {

     @Override
     public Set<EvidenceKey> convert(final CaseKey caseKey,
         final EvidenceDescriptorDtls evidenceDescriptorDtls) throws AppException,
         InformationalException {

         // return empty set indicating no evidence was converted by this evidence
   converter
         final Set<EvidenceKey> evidenceSet = new HashSet<EvidenceKey>();

         return evidenceSet;
     }
   }
   ```

2. Create a module class, which creates a Guice link binding that replaces the default evidence converter with the customized evidence converter, or add the new link binding to an existing module class. For example, see this custom module, which contains a link binding to replace the default Annual Tax Return external evidence converter with a custom disabled Annual Tax Return external evidence converter.

   ```
   /**
    * Contains disabled external evidence converter Guice bindings.
    */
   @AccessLevel(AccessLevelType.EXTERNAL)
   public class SampleDisabledExternalEvidenceConverterModule extends AbstractModule {

     /**
      * {@inheritDoc}
      */
     @Override
     protected void configure() {

       // Link binding replacing the default Annual Tax Return external evidence
   // converter with a custom disabled Annual Tax Return evidence converter.
       bind(AnnualTaxReturnExternalEvidenceConverterImpl.class)
         .to(SampleDisabledAnnualTaxReturnExternalEvidenceConverterImpl.class);

     }
   ```

3. If you created a new module class, update the *ModuleClassname.dmx* file to reference this new module.

13 Customizing inconsistency period processing

Inconsistency period processing allows a caseworker to give a client a reasonable opportunity period to provide outstanding verifications for evidence that requires verification. Cases can proceed during that period as if outstanding verifications were provided. The default inconsistency period processing infrastructure consists of a batch process, a workflow, and the inconsistency period processing APIs. You can create a custom event handler to customize the default inconsistency period processing.

Related tasks

Customizing change of circumstances on page 77
To customize change of circumstances for your environment, you must be familiar with the default implementation. Use this information to understand the process flow, and to identify the steps that you must complete to customize your system.

Customizing periodic data matching and annual renewals on page 129
Use the following information to help you to customize your periodic data matching or annual renewals implementation to your requirements.

13.1 Creating a custom event handler for inconsistency period processing

By default in HCR, an inconsistency period is created only once for the lifetime of a case. This behavior is coded in the curam.hcrcase.sl.event.impl.MilestoneCreationEventHandler. You can create a custom event handler to modify this default behavior.

About this task

By default, the curam.hcrcase.sl.event.impl.MilestoneCreationEventHandler.eventRaised(Event) event handler is used, where the Event object contains caseID as primary event data and relatedID as secondary event data. The handler reads a list of milestone configuration details with the curam.core.sl.entity.intf.MilestoneLink.searchMilestoneConfigDetailsByCreationEventAndConfigID(Creationl API. The list is then iterated and MilestoneDelivery is created when caseID and milestone configurationID have no associated milestones deliveries and when milestone is in "INPROGRESS" or "NONSTARTED" state.

By default, Program Group Manager raises an inconsistency period event when a case contains evidence with outstanding verifications. The event is raised by curam.healthcare.sl.impl.ProgramGroupManager.manageProgramGroup(CaseKey). This event is registered through the /EJBServer/components/HCR/events/handler_config.xml file with event class identifier of INCONSISTENCYPERIOD. You can find the default event handler at curam.hcrcase.sl.event.impl.MilestoneCreationEventHandler. MilestoneCreationEventHandler results in the creation of the inconsistency period milestone if no milestones were previously created for the case.

For more information, see "Merging Event Files" in the *Server Developer's Guide*.

Procedure

1. Create a *handler_config.xml* file at */EJBServer/components/
   %custom_component_name%/events/*.

2. Disable the existing event handler. You can disable it with the following event-registration.
   It is important to provide a removed="true" attribute and point to the correct existing event
   handler.

```
  <event-registration
 handler="curam.hcrcase.sl.event.impl.MilestoneCreationEventHandler" removed="true">
    <event-classes>
      <event-class identifier="INCONSISTENCYPERIOD"/>
    </event-classes>
  </event-registration>
```

3. Create a custom event handler.
   For example, curam.custom.event.impl.CustomMilestoneCreationCustomerEventHandler.

4. Register a custom event handler as shown:

```
  <event-registration
 handler="curam.hcrcase.sl.event.impl.MilestoneCreationCustomerEventHandler">
    <event-classes>
      <event-class identifier="INCONSISTENCYPERIOD"/>
    </event-classes>
  </event-registration>
```

5. Ensure that the new custom component takes precedence in the component order. You must do
   a clean server build when you modify component order.

Related concepts

Related reference

13.2 InconsistencyPeriod workflow

The InconsistencyPeriod workflow processes cases after the inconsistency period finishes.

This workflow takes MilestoneDelivery ID as an input. The workflow first checks whether any
outstanding verifications or issues are pending against the Insurance Affordability integrated case
of the milestone delivery ID. If no verifications exist, the workflow ends and the milestone is set
to complete. If verifications are present, the workflow checks whether the outstanding issues and
verifications have associated evidence that has been retrieved from an external system such as
the federal hub. This external evidence is used to verify the information to which the client has
attested.

If all the external evidence is present, then the workflow changes the waiver date to 10 days after
the current date. After the workflow moves the waiver date, the workflow copies the external
evidence to the client-attested evidence, updates the ClntAttestModifiedEvidence entity, and
sends the potential eligibility notification to the primary client. After the notification is sent,
the workflow completes the milestone. If the external evidence is not present against active
outstanding issues or verifications, then the workflow suspends the product delivery cases and
completes the milestone.

13.3 Inconsistency period workflow APIs

By default, the following APIs are called by the inconsistency period workflow.

```
curam.healthcare.sl.intf.HCREvidenceIssueVerifications.checkForOutstandingVerificationsAndIssues
```

Checks for all the outstanding verifications and issues against a given milestone delivery ID. Returns Boolean value indicating if there are pending outstanding verifications or issues.

```
curam.healthcare.sl.intf.HCREvidenceIssueVerifications.checkExtSystemDataForAvailability
```

Checks if all the active outstanding issues and verifications against given milestone ID have the respective external system evidence. The external system evidences checked against are INCARCERATION, ESI, MEC and EXTERNALINCOMEDETAILS.

```
curam.healthcare.sl.intf.HCREvidenceIssueVerifications.pushWaiversDateForInconsistencyPeriod
```

Pushes the waivers date by 10 days for the insurance affordability integrated case. The number of days is configurable and is configured with the help of ENV_INCONSISTENCY_PERIOD_DUE_EXTENSION_DAYS variable in Environment.xml.

```
curam.healthcare.sl.intf.HCREvidenceIssueVerifications.overrideClientAttestedEvidence
```

Copies the external evidences to client attested evidence. This method ends the existing client attested evidence with current date and creates a new evidence with the available external system evidence. ClntAttestModifiedEvidence entity is used to store the case and evidence details.

```
curam.healthcare.sl.intf.HCREvidenceIssueVerifications.sendPotentialEligiblityNotification
```

Sends potential eligibility notification to the primary client.

```
curam.healthcare.sl.intf.HCREvidenceIssueVerifications.suspendPDCasesByMileStoneDelivery
```

Suspends all the product delivery cases with the given milestone delivery ID.

```
curam.healthcare.sl.intf.HCREvidenceIssueVerifications.endMilestoneForInconsistencyPeriod
```

Completes the milestone for a given milestone ID related to the inconsistency period.

13.4 Inconsistency Period Evidence Activation batch process

This batch process activates the evidence that is created or modified by the InconsistencyPeriod workflow.

Inconsistency Period Evidence Activation initiates the following processing steps:

1. Reads all the cases from ClntAttestModifiedEvidence, taking the date as input.
2. Activates the created or modified evidence for the case and updates the record status on ClntAttestModifiedEvidence.

Parameters

Parameter	Description	Default value
processingDate	Current date	Not applicable

13.5 Inconsistency Period Evidence Activation Stream batch process

This batch process supports streaming for the inconsistency period Insurance Affordability cases.

Parameters

Parameter	Description	Default value
processingDate	Current date	Not applicable

14 Configuring Account Transfer with the Federally Facilitated Exchange

You can configure how Account Transfer applications are processed and sent and received between Cúram and the Federally Facilitated Exchange.

> **Note:** The source code for the Account Transfer process is delivered with Cúram. You can use the source code to build any customizations that are required for the Account Transfer process. The location of the Account Transfer source code is `EJBServer/components/FederalExchange/sample/src.zip`. For more information about the source code for the Account Transfer process and the release of the sample implementation, see the *CMS Account Transfer Business Services Description (BSD)* related link.

Account Transfer uses the data store and the Persistence Infrastructure. Account transfer functionality is focused on accurately capturing applicant details, their circumstances, and data relevant to the correct eligibility determination.

The Intake datastore is used to enter account information into the Cúram application. The Intake datastore is populated by mapping the contents of the Inbound Account Transfer payload onto the datastore. Only pre-existing datastore evidence entities are supported. Using only pre-existing datastore evidence entities ensures that cases and evidence that are created from account transfers and the citizen portal are identical.

The Income Support for Medical Assistance (Health Care Reform) is flexible. You can implement your own custom data mappings to support your particular requirements.

Related information

[CMS Account Transfer Business Services Description (BSD)](#)

14.1 The FederalExchange component

The FederalExchange component helps state agencies to process Account Transfer applications that originate from the Federally Facilitated Exchange (FFE). In addition, the FederalExchange component sends applications that originate from the state agency to the FFE for applicants that are not eligible for Medicaid or CHIP.

14.2 Configuring Federal Exchange

Configure Account Transfer to and from the federal exchange by modifying the appropriate properties.

About this task

For information about configuring properties, see "Configuring Application Properties" in the *Cúram System Configuration Guide*.

Activating Account Transfer

Account Transfer is switched off by default. When you activate Account Transfer, eligibility determinations are processed and prepared to be sent to the FFE by the FederalExchange component.

Procedure

To activate Account Transfer, set the `curam.healthcare.account.transfer.activate.outbound.mapping` property to true.

Enabling batch processing of account transfer applications

If you want to process Account Transfer applications by batch processing, you can modify a property to stop the account transfer application data from being sent to Cúram for inbound applications and to the FFE for outbound applications.

About this task

Enabling batch processing prevents the mapping of the data and the processing of that mapped data by Cúram or the FFE. The *FederalExchangeApplication* entity stores the jobs that are pending for each batch process.

Procedure

To process Account Transfer applications by batch processing, change the value of the `curam.healthcare.account.transfer.processing.mode` property from the default value of `online` to `batch`.

Configuring the sending of Account Transfers to Cúram

You might want to complete the mappings in real time and to stop the processing just before the Account Transfer is sent to Cúram for case processing.

About this task

If you stop processing applications, the applications remain in a PENDING state on the *FederalExchangeApplication* entity.

For other possible states for entries on this entity, see the HCRFedExchangeAppStatus code table.

Procedure

To ensure that no Account Transfer applications are sent to Cúram, update the `curam.healthcare.account.transfer.auto.submit` property from true to false.

Selecting the source data set for outbound mapping

Outbound mapping can be completed from two different sets of source data, intake and case processing. Different data store schemas are used to store the data in each case.

About this task

The following sets of source data are available:

- **Intake**
  The data that is stored in the data store as a result of a case worker completing the internal case worker intake application for Health Care Reform.
- **Case processing**
  The data that is stored in the data store as a result of running an implementation of HCRDatastoreBuilder to convert case and person evidence for a Health Care Reform application to data store data.

Procedure

1. Set the `curam.healthcare.account.transfer.outbound.mapping.source` property to intakefor the data store data from the intake application for Health Care Reform, or caseprocessingfor the data that is derived from case and person evidence.
2. Set the `curam.healthcare.account.transfer.internal.datastore.schema` property to the correct schema name depending on the source of the data.

Setting the identity of the sender US state

Ensure that the sender is identified correctly by setting the correct US state. The codes that are used to denote the sender state are stored as properties.

About this task

Procedure

Update the following properties for the US state for which HCR is implemented:

```
curam.healthcare.account.transfer.sender.state.code
curam.healthcare.account.transfer.sender.county
curam.healthcare.account.transfer.sender.category.code
```

Setting the Account Transfer agency type

You can configure the Account Transfer Agency type to be Medicaid, CHIP, or both.

Procedure

Update the `curam.healthcare.account.transfer.agency.type` property with the Account Transfer agency type. The agency type can be M (Medicaid), C CHIP, or B (Both). The default value is B.

Setting the federal exchange code

You can set the code or name for the federal exchange that is included with inbound and outbound Account Transfer requests.

Procedure

Update the `curam.healthcare.account.transfer.federal.exchange.code` property with the code or name of the federal exchange that is included in with inbound and outbound account transfer requests.

Linking the Datastore schema name to the Account Transfer person reference

You can configure the Datastore schema name that links the Account Transfer external person reference to the Cúram person reference. The Datastore schema name is also used for the mapped reference of the person after data mapping has been performed.

Procedure

Update the `curam.healthcare.account.transfer.person.link.schema` property with the Datastore schema name.

Setting the data store schema name for the FFE schema

You can set the schema name for the data store schema representation of the FFE schema.

Procedure

Set the schema name for the data store schema representation of the FFE schema in the `curam.healthcare.fedexchange.version.schema` property.

Configuring Account Transfer date and time formats

You can configure the date and time formats for inbound and outbound Account Transfers to ensure consistency between inbound and outbound payloads and their destinations.

Procedure

1. Set the date/time formats for inbound Account Transfers by configuring the following application properties.

Application property	Description
`curam.healthcare.account.transfer.mapping.inbound.date.format`	Date format that is used to create the date in the Datastore based on the FFM date for inbound Account Transfers.
`curam.healthcare.account.transfer.mapping.inbound.datetime.format`	Date time format that is used to create the Cúram date time that is based on the FFM date time for inbound Account Transfers.

2. Set the date/time formats for outbound Account Transfers by configuring the following application properties.

Application property	Description
`curam.healthcare.account.transfer.mapping.format.date`	Date format that is used to create the FFM date that is based on the Cúram date for outbound Account Transfers.
`curam.healthcare.account.transfer.mapping.format.datetime`	Date time format that is used to create the FFM date time that is based on the Cúram date time for outbound Account Transfers.

14.3 Extending Federally Facilitated Exchange data mappings

You can modify the default Federally Facilitated Exchange data mappings to add attributes or entities to the data that is sent or received.

Federally Facilitated Exchange (FFE) mappings are called when data is received from the FFE (inbound) or sent to the FFE (outbound).

When you receive data from the FFE, you must map data from the FFE data schema to the Cúram data store schema so that Cúram can process that data.

When you send data to the FFE, you must map data from the Cúram data store schema to the FFE data schema so that the FFE can process that data.

Adding or updating the attributes for a data store entity

Modify the properties of the appropriate Persistence Infrastructure event to add or update an attribute.

About this task

After an entity is mapped by the FederalExchange component, a Persistence Infrastructure event is sent to allow custom listeners of the event to add or update the attributes on the entity.

As with all data-store processing, the attributes that are added to an entity must conform to the data store schema that is configured for that instance of data store data. For information about the event signature and more information on usage, see the Javadoc.

Procedure

To add or update an attribute, configure the appropriate event for the custom listeners:

• Inbound

```
curam.hcr.fedexchange.mapper.impl.EntityMapper.MapEvent.customInboundMap
```

• Outbound

```
curam.hcr.fedexchange.mapper.impl.EntityMapper.MapEvent.customOutboundMap
```

Example listener:

```
/**
 * Raised when in-bound mapping (from FFM to Curam) has been completed
 * on a given
 * data store entity.
 *
 * @param mappedEntity
 *          The data store entity that contains mapped data.
 *          Note that if the entity is a child it will not have
 *          been added to its parent at this point and will therefore
 *          not have a unique identifier.  The result of this is that
 *          no children can be added to this entity during the
 *            processing
 *          of this method.
 *  @param originalElement
 *          The original data that can act as the source for the
 *          mapped data.
 *
 * @throws AppException
 *            Generic Exception Signature.
 * @throws InformationalException
 *            Generic Exception Signature.
 */
public void customInboundMap(Entity mappedEntity,
    Element originalElement)
                        throws AppException, InformationalException{}
```

Adding an entity as a child of a mapped data store entity

When an entity is mapped and you add child entities to the entity, an event is sent that allows custom listeners to add extra child entities to the entity.

About this task

Any child entity types that are added must exist in the data store schema for the data store data that is being processed.

Procedure

To add a child entity, configure the appropriate event for the custom listeners:

- Inbound

```
curam.hcr.fedexchange.mapper.impl.EntityMapper.MapEvent.
```

- Outbound

```
curam.hcr.fedexchange.mapper.impl.EntityMapper.MapEvent.
customOutboundMapChildren
```

Example listener:

```
/**
* Raised when in-bound mapping (from FFM to Curam) has been completed * on a given
  * data store entity and when children can be added to that entity.
  * Custom processing should check for an existing child before
  * creating one.
  *
  * @param mappedEntity
  *          The data store parent entity that contains mapped data.
  *          This entity can be used to create valid child entities
  *          underneath.
  * @param originalElement
  *          The original data that can act as the source for the
  *          mapped child data.  It is possible to traverse up or down
  *          the DOM tree using the originalElement as the starting
  *          point
  *
  * @throws AppException
  *           Generic Exception Signature.
  * @throws InformationalException
  *           Generic Exception Signature.
  */
 public void customInboundMapChildren(Entity mappedEntity,
  Element originalElement)
                       throws AppException, InformationalException{}
```

Adding or replacing a top-level data store entity

An element mapping provider can map implementations for a target entity type. You can use this element mapping provider to add custom processing to create an entity that is at a top level.

About this task

Typically, a child of the Application root entity or another entity type that can have only a single instance.

Procedure

Use the appropriate events to add a custom mapping implementation for the element mapping provider:

- Inbound

```
curam.hcr.fedexchange.mapper.impl.ElementMapperEvent.
addElementMapperEvent(Map elementMappers)
```

The elementMappers contains a map of entity types and mapping implementations to which you can append extra entity types and custom mapping implementations that are called as part of the element mapping provider processing.

- Outbound

```
curam.hcr.fedexchange.mapper.ffe.impl.FFEElementMapperEvent.
addFFEElementMapperEvent(Map elementMappers)
```

A custom listener to this event is implemented in much the same way as for the listener for the inbound provider event. The target entity type that is being added to the map is an entity type in the Federal Exchange data store schema. The mapping implementation maps data from the data store schema to the Federal Exchange data store schema.

Example listener:

```
/**
 * Raised when {@linkplain ElementMapperProvider} is initialized to
 * allow
 * additional EntityMapper to be included.
 *
 * @param elementMappers
 *            The map containing a string that represents the
 *            element being mapped from and the mapping implementation
 *            that creates and maps to the corresponding element on the
 *            target schema.
 *
 * @throws AppException
 *            Generic Exception Signature.
 * @throws InformationalException
 *            Generic Exception Signature.
 */
public void addElementMapperEvent(Map<String,
                        Provider<? extends EntityMapper>> elementMappers)
        throws AppException, InformationalException;
```

Adding or updating entities for an outbound response to the FFE

When the response to be sent to the FFE is built by the FederalExchange component, an event is sent with all of the response data. Listeners to this event can then update the response data before they send it to the FFE.

Procedure

To add or update outbound entities, configure the following event:

```
curam.hcr.fedexchange.mapper.impl.EntityMapper.
MapEvent.customOutboundMapResponse
```

Example listener:

```
/**
  * Raised when out-bound mapping (from Curam to FFM) has been completed on the
  * response being
 * sent to the FFM.
 * Custom processing should check for an existing response entity before
 * creating one.
 *
 * @param mappedEntity
 *            The data store parent entity that contains mapped data.
 *            This entity for an applicant can be used to create valid response
 *             entities.
 * @param originalElement
 *            The original data that can act as the source for the
 *            mapped response data.
 *
 * @throws AppException
 *            Generic Exception Signature.
 * @throws InformationalException
 *            Generic Exception Signature.
 */
public void customOutboundMapResponse(Entity mappedEntity,
                                  Element originalElement)
                        throws AppException, InformationalException{}
```

14.4 The Web Service Java API

You can use the Federal Exchange component Java API for data that is received from or sent to the Account Transfer web service, or to send data to the FFE.

Inbound Account Transfer payload processing

You can use the Java API as entry and exit points for data that is sent or received from the Account Transfer web service.

Use the
`curam.hcr.fedexchange.ws.impl.AccountTransferWS.initiateAccountTransfer`
method to send an account transfer to Cúram from the FFE or to send an account transfer response to Cúram from the FFE.

The inbound webservice passes the payload document to the
`curam.hcr.fedexchange.ws.impl.AccountTransferWSImpl.initiateAccountTransfer(Documen`
API. The data is then persisted to the federalexchangemessagelog database table. Further processing varies depending on your local configuration. An application case of type 'Account Transfer COC Application Case' is created when a payload is identified as an inbound COC payload. A program of type 'Account Transfer COC Program' is associated with this case type. Evidence types that are associated with the Insurance Affordability application case are also associated with the Account Transfer COC application case. The Account Transfer COC application case is otherwise the same as the Insurance Affordability application case.

Verifications are not configured for the default Account Transfer COC application case. The evidence broker is configured to not automatically accept or activate evidence from the Account Transfer COC application case to the integrated case. Evidence that is not modified is filtered out and the caseworker is shown only evidence that is modified. The caseworker can manually accept or reject the evidence on the integrated case.

If evidence brokering from the Account Transfer COC application case to the integrated case fails, then the Account Transfer COC application case remains open. If an Account Transfer COC application case is open and another COC payload is subsequently processed for any of the case members from the original case, then a task is raised by the system informing the caseworker of the presence of one or more active COC applications in progress. If multiple integrated cases are found for the Account Transfer COC application case, then a task is raised by the system informing the caseworker of the presence of multiple integrated cases.

Inbound payload identification and routing
Use the global application ID of the payload to identify and track inbound COC payloads. Depending on the signature date of the payload and whether it is a duplicate payload, different processing will be enacted.

If the global application ID of the payload is the same as one of the payloads which has been processed and the original signature date is different from the processed payload; then the payload is identified as a COC payload and will be processed as an Account Transfer COC application.

If multiple payloads with the same application ID and different original signature dates are received, these payloads are treated as multiple change of circumstances reported by the applicant. Each payload requires separate processing.

If multiple payloads with the same application ID and same original signature date are received, then the first received payload is processed as the Account Transfer COC. The other payloads are ignored.

If the global application ID and transfer ID of an incoming payload is found to be same as that of a payloads that has already been processed, then the payload is not processed further. This represents a duplicate payload received by the account transfer system.

Inbound payload mapping configuration

Use the mapping configuration XML files for the Account Transfer COC application case to customize the mapping for inbound payloads.

Account transfer data mappings take into account differences in data types between CMS and Cúram schemas and map them accordingly. FFM data is stored in suitable IEG data types. There are no validations upon storage, except for the documentation definition mandated validations. Validation mismatches are caught during the mapping process. For example, the number of babies due in a pregnancy is considered optional in an inbound data element mapping by the FFM. It is mandatory in Cúram. The value is defaulted to 1 by the data mapper. If the inbound payload or a consecutive inbound change of circumstance contains the correct number of babies due, the number is updated in the FFM.

A similar flexible approach is taken where the FFM can provide date information as Date or DateTime data types. Mappers are flexible enough to work with either of these inputs. When one data element depends on another, the data mapper considers both elements to make the correct mapping. Account Transfer infrastructure allows mappers to traverse the inbound payload in any order. This means that any number of comparisons and dependencies can be considered by the mapper logic. This mapping flexibility is also possible for outbound account transfers.

Occasionally, the FFM provides a number of string or numeric values that are converted to a single boolean data type in Cúram. This is the most suitable approach for the existing evidence infrastructure.

The Federal Exchange component relies on enumerators to map FFM data to compatible Cúram code table value and vice versa. You must provide custom enumerators where custom code-tables are used instead of default code-tables. Enumerator naming conventions follow standard Cúram naming conventions. For example, **`<Area of mapping Reference>`**`FieldMap.class`.

In the mapping configuration XML, the initial application date is mapped to the evidence start date.

You can configure the Account Transfer COC mappings using the following files:

- *`EJBServer\components\FederalExchange\data\initial\clob`*
  `\ATCOCEvidenceMappingConfiguration_1_6.xml`
- *`..\clob\ATCOCEvidenceMappingConfiguration_1_7.xml`*
- *`..\clob\ATCOCEvidenceMappingConfiguration_1_8.xml`*

Process inbound Account Transfer COC payloads in batch

You can use the Java API to create a batch process that takes Change of Circumstance Account Transfers in a 'Pending' state and passes them for subsequent processing.

Use the `curam.hcr.fedexchange.ws.impl.AccountTransferWS.initiateAccountTransfer` method to implement a batch process for Account Transfer COC payloads. Use the Account Transfer payload as the argument for the batch process. All inbound payloads on the FederalExchangeApplication entity with a status of IBD_UPDATE_PENDING should be processed in the batch.

For more information about batch processes, see the *Batch Processes Developer Guide*.

Related tasks

[Enabling batch processing of account transfer applications on page 158](#)
If you want to process Account Transfer applications by batch processing, you can modify a property to stop the account transfer application data from being sent to Cúram for inbound applications and to the FFE for outbound applications.

Outbound processing

To send data to the FFE, the API includes events that provide the data to be sent.

```
curam.hcr.fedexchange.ws.impl.AccountTransferWS.
OutBoundDataEvent.sendOutBoundTransferDataEvent
```

This property sends an account transfer application from Cúram. The listener that receives this event can alter the data to meet specific custom needs (if not already catered for by the custom mapping processing) and then send that data to the FFE by a web service. Any mapping updates that must be made by custom processing must use the events that are sent during data mapping.

```
curam.hcr.fedexchange.ws.impl.AccountTransferResponseWS.
OutBoundResponseEvent.sendOutBoundResponseEvent
```

This property sends a response of an account transfer application from Cúram. The listener that receives this event can alter the data to meet their specific needs and then send the data to the FFE through a web service. Any mapping updates that must be made by custom processing must use the events that are sent during outbound response processing.

HCRFedExchangeAppStatus code table descriptions

A list of the possible status states of the FederalExchangeApplication that uses the HCRFedExchangeAppStatus code table. For clarity, the status states are divided by the direction of the request. The status states are listed in the same sequence in which the transitions happen.

Table 37: Account Transfer from FFM to State Medicaid Agency

Code	Java Identifier	Full Description
HCRIFEIP	IBD_IN_PROGRESS	The initial status on creation of a Federal Exchange Application. On creation, the record contains the root data store entity ID of the external data store that is used to store the Account Transfer payload from the Federally-Facilitated Marketplace (FFM).
HCRIFEUD	IBD_UPDATE_PENDING	This state is the other initial state that is possible for an Account Transfer received by the State. The Federal Exchange Application is created in this state if there is an existing Account Transfer with the same Global Application ID. In other words, this transfer is considered to be a change of circumstance.
HCRIRSAK	IBD_ACKNOWLEDGED	Set after an inbound request is stored and successfully acknowledged.
HCRIFEER	IBD_ERROR	Set when any issues are encountered during the mapping of the FFM payload to the internal data store, or when the FFM payload is stored in the external data store.
HCRORSIP	OBD_RESPONSE_IN_PROGRESS	Set when the processing for sending a response to the FFM is initiated.
HCRORSAK	OBD_RESPONSE_ACKNOWLEDGED	Set after the response sent by the State Medicaid Agency is acknowledged successfully by the Federally-Facilitated Exchange (FFE).
HCRIFEPD	IBD_PENDING	If transfers are configured to happen in batch mode. The Account Transfer payload is stored in the external data store but no further processing happens as part of the online processing.

Table 38: Account Transfer from State Medicaid agency to FFM

Code	Java Identifier	Full Description
HCROFEIP	OBD_IN_PROGRESS	Set on a new instance of FederalExchangeApplication that is created for a transfer from the State to the FFM.
HCROFEPD	OBD_PENDING	Set if transfers are configured to happen in batch mode. No further processing is done for this transfer as part of online processing.
HCROFEAK	OBD_ACKNOWLEDGED	Set after a transfer from the State is acknowledged successfully by the FFE.
HCROFEER	OBD_ERROR	Set if an acknowledgement to an outbound transfer was not received or is not successful. Also set if there are any issues during mapping from the HCR data store to the FFM data store. If there were errors during mapping, Federal Exchange Applications are not transferred.
HCRIRSIP	IBD_RESPONSE_IN_PROGRESS	Set on receiving the response from the FFM for an Account Transfer from the State.
HCRIFEAK	IBD_RESPONSE_ACKNOWLEDGED	Set when the response from the FFM for an Account Transfer from the State is successfully acknowledged
HCRIRAER	IBD_RESPONSE_ACKNOWLEDGED_ERROR	Set if any issues were encountered when the response to an outbound account transfer is stored, or if there were issues with the generation of an acknowledgement.

14.5 Adding a new entity

You can add a new entity to replace an existing entity or to create an entity that is not mapped and created by default. For each new entity, write an entity mapper and add the new entity to the Federal Exchange data store schema.

Writing an EntityMapper

You must write an EntityMapper for each new entity. An EntityMapper must implement the `curam.hcr.fedexchange.mapper.impl.EntityMapper` interface.

About this task

An implementation of `curam.hcr.fedexchange.mapper.impl.ElementMapperUtil` is provided in the map method to facilitate searching for required elements and attributes in the source XML to be used to populate the entity or entities that are being created.

Procedure

1. Using the provided example, implement an EntityMapper.
2. After you implement the EntityMapper, register it by using the ElementMapperEvent inbound or outbound event as appropriate. This depends whether the Mapper implementation is being called for inbound or outbound processing.

Example

This example outlines how IncomeItem entities might be mapped from the FederalExchange external system into Cúram and added to the data store.

```java
/**
 * Sample entity mapping implementation that creates a new
 * data store entity and appends it to a parent entity.
 */
public class SampleEntityMapperImpl implements EntityMapper {

  /** The source element to map from, the source elements of interest can be
   * searched for by using this element **/
  private Element source;
  /** The FederalExchangeApplication persistence infrastructure implementation
   *  for the FederalExchangeApplication entity **/
  private FederalExchangeApplication federalExchangeApplication;

  @Override
  public void setSource(Element source) {
    this.source = source;
  }

  @Override
  public void map(Entity parent, ElementMapperUtil elementMapperUtil) {
    Datastore ds = parent.getDatastore();
    //get the element from the source i.e. the element from the FederalExchange
    //XML
    List<Element> incomeItems =
      elementMapperUtil.getElements(FFEEntityType.PERSONINCOME.entityType(),
        source);

    for(Element incomeItemSource : incomeItems){
      //create the new entity in the target data store
      Entity incomeItem = ds.newEntity(EntityType.INCOMEITEM.entityType());
      //set the attributes on the new target entity
      incomeItem.setTypedAttribute(IncomeItemFieldMap.STARTDATE.hcrField(),
        FieldMapperUtil.formatDate(
          elementMapperUtil.getAttribute(incomeItemSource,
           elementMapperUtil.createFindAttributeQuery(
             IncomeItemFieldMap.STARTDATE.ffeField())))));
      incomeItem.setTypedAttribute(IncomeItemFieldMap.ENDDATE.hcrField(),
        FieldMapperUtil.formatDate(
          elementMapperUtil.getAttribute(incomeItemSource,
              elementMapperUtil.createFindAttributeQuery(
                IncomeItemFieldMap.ENDDATE.ffeField())))));

      incomeItem.setTypedAttribute(IncomeItemFieldMap.INCOMEAMOUNT.hcrField(),
        elementMapperUtil.getAttribute(incomeItemSource,
          elementMapperUtil.createFindAttributeQuery(
            IncomeItemFieldMap.INCOMEAMOUNT.ffeField())));
      //add the new entity as a child of the parent entity
      parent.addChildEntity(incomeItem);
    }
  }

  @Override
  public void postMap(Entity rootEntity, Entity personEntity) {
    //no post map processing required
  }

  @Override
  public void setFederalExchangeApplication(
      FederalExchangeApplication federalExchangeApplication) {
    this.federalExchangeApplication = federalExchangeApplication;
  }
}
```

Updating the Federal Exchange data store schema

If a new entity is being added by custom processing, then you must update the data store schema that is used to store the entity for inbound and outbound mapping.

Before you begin

It is important to note the following when you update the Federal Exchange data store schema for Account Transfer.

- If element text exists in the payload from the Federal Exchange, then this text is converted into an attribute. This allows the text to be stored in the data store. For example:

```
<IncomeAmount>1200</IncomeAmount>
```

You define that Federal Exchange payload XML in the data store schema as follows:

```
<xsd:element name="IncomeAmount">
      <xsd:complexType>
          <xsd:attribute name="value" type="d:SVR_STRING"/>
       </xsd:complexType>
</xsd:element>
```

Note the use of the attribute value to store the element text.

- If the element in the Federal Exchange payload contains a name space prefix, then the data store schema must contain an attribute that defines the name space prefix value as the default value. For example:

```
<hix-core:IncomeAmount>1200</hix-core:IncomeAmount>
```

You define that Federal Exchange payload XML in the data store schema as follows:

```
<xsd:element name="IncomeAmount">
      <xsd:complexType>
          <xsd:attribute name="value" type="d:SVR_STRING"/>
          <xsd:attribute name="nameSpacePrefix" type="d:SVR_STRING"
   default="hix-core:"/>
      </xsd:complexType>
</xsd:element>
```

Procedure

1. Identify the relevant data store schema. The name of the data store schema name that stores the Federal Exchange data for Account Transfer is denoted by the *curam.healthcare.fedexchange.version.schema* property.
2. Update the data store schema with the new entity.

14.6 Account transfer workflows

View the default Account transfer workflows in the **Workflow** section of the Administration Workspace.

The default Account Transfer workflows can be viewed in the Process Definition Tool. In the Administration Workspace, navigate to the workflow from the left navigation menu: **Workflow**

> **Released Processes** > **Account Transfer Inbound COC** or **Account Transfer Straight Through Authorise**.

The **Account Transfer Inbound COC** workflow raises tasks for the related integrated case if there is evidence that requires attention in the inbound change of circumstance. The **Account Transfer Inbound COC** workflow can be customized to automate the evidence transfer process.

The **Account Transfer Straight Through Authorise** workflow is enacted in the following scenarios:

- When no evidence on the inbound change of circumstance requires attention
- When there are only person participants on the change of circumstance application and no open duplicate account transfer change of circumstance applications exist for those members

15 Monitoring Health Care Reform

The extensive and fine-grained customization options that are available in Cúram enable you to fully reflect the current Federal and State legislation. However, this fine-grained customization means that you must monitor your installation for the arrival of the unusual or non-routine applications that can arise when legislation is tested in practice. Use the following monitoring options in Cúram to identify incoming exceptions to your custom implementation that might require action on your part.

15.1 Monitoring HCR applications

Use the following Cúram Income Support for Medical Assistance views to monitor the progress of applications though the system, and to help you to troubleshoot issues.

HCR application intake process overview

Use this information to understand how HCR applications are processed, from the submission of an application to the creation of product delivery cases.

For documentation purposes, the process diagram is split into two parts.
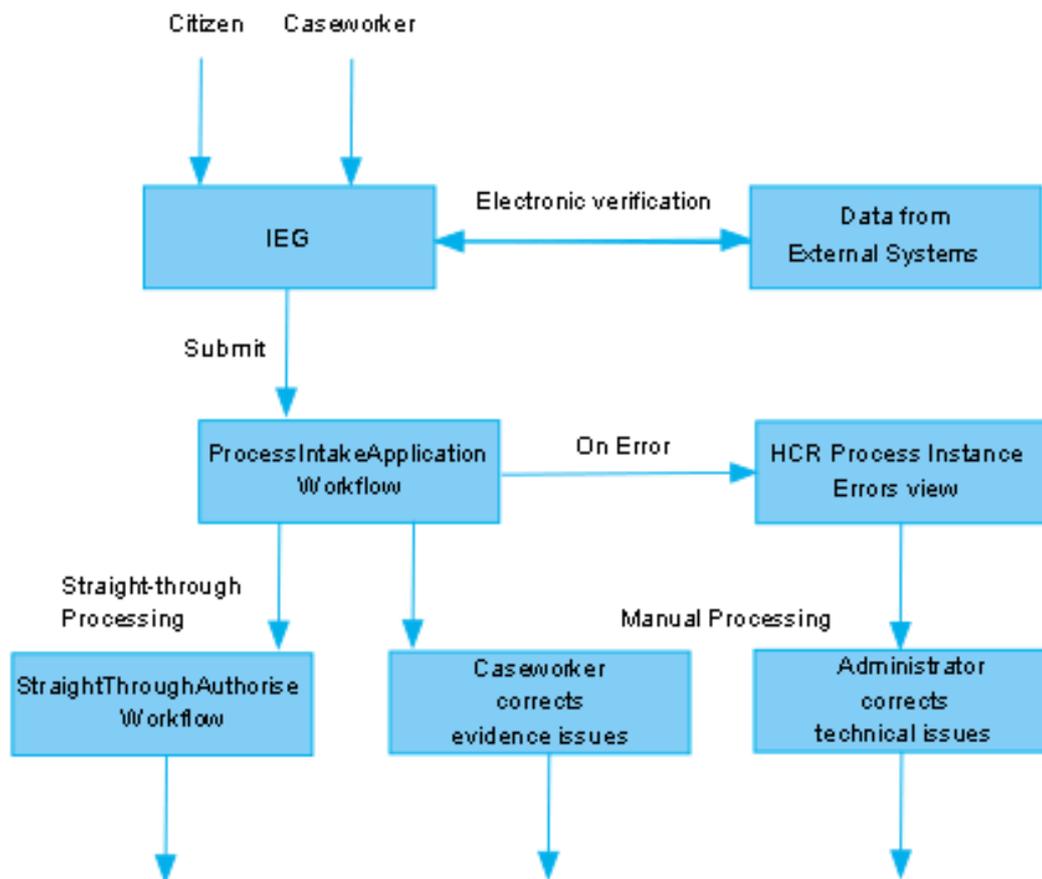
Figure 7: Application intake process diagram: Part 1

- **A citizen or caseworker submits an application**
  A citizen or caseworker completes the dynamic application questionnaire and submits the application.
- **Qualified citizen data is verified with external systems**
  Data that is configured to be verified with external systems is compared with the specified external data sources and the data store is updated where required.

- **The citizen submits the application and the intake process starts**
  When the citizen clicks submit, the ProcessIntakeApplication workflow starts. An application case is created and data from the data store is applied to the application case as evidence. The ProcessIntakeApplication workflow has a resilient mode of operation to gracefully handle certain types of error. You can enable this resilient application handling by setting the curam.intake.use.resilience property to true. When resilient mode is enabled, ProcessIntakeApplication handles invalid evidence by creating as much evidence as possible and then assigning the application case to a caseworker for manual processing.

  - **The straight-through processing workflow**
    If none of the data on the case requires manual processing, the case is routed to the StraightThroughProcessing workflow. Straight-through applications are authorized automatically. On authorization, the deferred transaction EVIDENCE_SHARE_BULK is started and evidence is shared to the integrated case.
  - **Manual processing**
    If any of the data on the case requires manual processing, the case is routed to the deferred transaction process APPLICATIONAUTHORIZATION for manual processing. A caseworker can correct the evidence issues and authorize the case. On authorization, the deferred transaction EVIDENCE_SHARE_BULK is started and evidence is shared to the integrated case.
  - **HCR Process Instance Errors view**
    Some errors can occur because of technical issues in source code or rules. These errors cannot be corrected by a caseworker and are sent to the Process Instance Error Queue. You can see these errors in the HCR Intake Process Errors view.
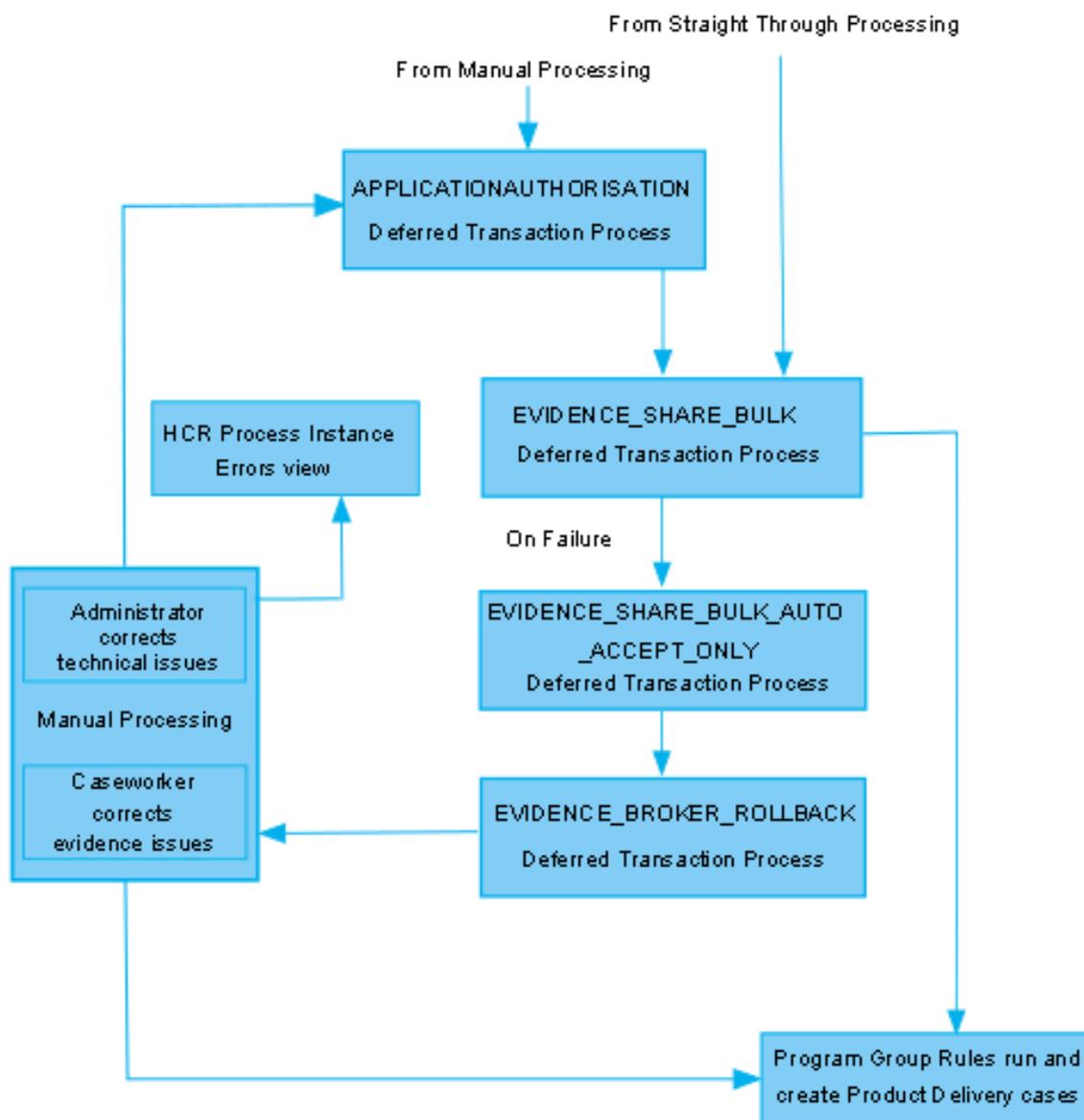
Figure 8: Application intake process diagram: Part 2

- **Where possible, evidence from the application case is brokered directly to the integrated case**
  On authorization, the EVIDENCE_SHARE_BULK deferred transaction process is started and evidence is shared to the integrated case. Program group logic processing is triggered automatically.
- **If brokering fails partially, then evidence from the application case is added as in-edit or incoming evidence to the integrated case, manual intervention is then needed**
  The evidence is directed to the EVIDENCE_SHARE_BULK_AUTO_ACCEPT_ONLY deferred transaction process. A caseworker accepts the in-edit or incoming evidence, corrects

any issues, and applies the changes. Program group logic processing is triggered when the changes are applied.

- **If brokering fails fully, manual intervention is then needed**

  If adding the evidence as 'Incoming' evidence fails, then the EVIDENCE_BROKER_ROLLBACK deferred transaction process initiates a number of fallback actions:

  - The application case is set to Authorization Failed status to indicate that authorization process failed.
  - Any new integrated cases or product delivery cases that were created are closed.
  - When the cause of the authorization failure is addressed, the caseworker submits the application case for authorization.

- **The program group logic runs on the integrated case and creates the required product delivery cases**

  Program group logic is triggered when evidence changes are applied to the integrated case and creates the required product delivery cases.

  Program group logic is triggered automatically by the EVIDENCE_SHARE_BULK deferred transaction process, and each time a caseworker applies evidence changes on the integrated case.

> **Note:** The Health Care Reform program group logic to determine which potential multiple product delivery cases are to be created depends on predefined rule sets and therefore bypasses the Common Intake product delivery creation process. This logic does not configure the product delivery type for the program and therefore does not use the productDeliveryCaseID field on the programauthorisationdata table. For the Common Intake process, if a product delivery type is configured against a program, this product delivery type is created as part of a successful program authorization and recorded in the ProgramAuthorisationData entity.

Related tasks

[2.2 Configuring the resilient option for the process intake application workflow on page 44](#)
After you install version 6.0.5.5 or later, ensure that you set the resilient option for the process intake application workflow, which enables a more granular workflow with better error handling.

(deprecated) Monitoring HCR intake reports

Use (deprecated) HCR Intake Reports to monitor all HCR applications that are being processed by the system, from initiation through to an eligibility decision. The reports provide a count of applications in each state and highlights applications that require intervention to progress.

Before you begin

> **Warning:** Refreshing these reports generates significant load on the system and can prevent intake applications from being processed. Before you refresh the reports, ensure that no ongoing intake applications or other system activities are affected.

About this task

If technical issues require investigation, a message is displayed under **Intervention Required**. To investigate these issues, open the HCR Intake Process Errors view to get more detailed information.

Procedure

1. Log in to the Cúram Administration application as a user with administrator permissions.

2. Select **Administration Workspace** > **Process Monitoring** > **HCR Intake Reports**.

   A confirmation page opens.

3. Confirm that you want to run the reports by clicking **Yes**, or close the tab to exit without running the reports.

🖼 *(deprecated) HCR Intake Reports*

HCR Intake Reports provide a business view of all HCR applications that are being processed by the system, from initiation through to an eligibility decision. The reports provide a count of applications in each state and highlights applications that require intervention to progress.

> **Warning:** Refreshing these reports generates significant load on the system and can prevent intake applications from being processed. Before you refresh the reports, ensure that no ongoing intake applications or other system activities are affected.

You can choose to view the current status of applications that have been submitted since a specified date. By default, this date is set to one week in the past from the present date.

HCR Intake Reports display the status of applications and programs as follows:

- **Incoming Applications**
  The total number of applications that are started is shown, divided into the number of applications that are started and pending submission, and the number of applications that were submitted. The applications are categorized by the source of the application:

  - **Account Transfer**
    Applications that are submitted from external systems through account transfer.
  - **Caseworker**
    Applications that are submitted from the Cúram application by a caseworker.
  - **Online**
    Applications that are submitted from an online citizen account by a citizen.

- **Applications Received**
  The total number of applications that are received by the agency is shown and this number generally matches the total number of applications that are submitted as represented in **Incoming Applications**.

> **Note:** Occasionally, these two numbers might not match due to the short time interval between the submission and receipt of an application.

The applications are categorized by their status:

- **In Progress**

  Applications that are currently in progress.

- **Closed**

  Applications that progressed to the completion of the intake process and for which an eligibility decision is available.

- **Withdrawn**

  Applications that a citizen withdrew after they submitted them.

- **Intervention Required**

  The number of in-progress applications that could not be processed automatically and that require intervention to progress them further. The applications are categorized by the type of intervention that is needed.

  - **Outstanding Registrations**

    Applications where person registration is incomplete, that is, which are associated with one or more prospect persons. All applications with outstanding registrations are assigned to this category, irrespective of any other interventions that they might require.

  - **Outstanding Verifications**

    Applications that have evidence that requires verification, excluding applications with outstanding registrations.

  - **Failed Validations**

    Applications that are in the Awaiting Resolution state, due to invalid evidence.

  - **Failed to Determine Eligibility**

    Integrated cases where an eligibility decision could not be made, typically because of incoming or in-edit evidence.

  - **Failed to Broker Evidence**

    Application cases in the Authorization Failed state, where evidence could not be brokered from the application case onto other cases.

  If technical errors require investigation, a message is displayed. To investigate technical errors, open the **HCR Intake Process Errors** view.

- **Eligible Programs**

  The number of currently active HCR programs that were created since the specified date, categorized by type. The programs can be generated as a result of the HCR Intake process, as well as other application processes, such as Change of Circumstances.

Table 39: Default Programs and abbreviations

Abbreviation	Program
CHIP	Children's Health Insurance Program
EMA	Emergency Medicaid
ESI	Employer Sponsored Insurance
Exem	Exemption
IA	Insurance Assistance
MA	Streamlined Medicaid
SBHP	State Basic Health Plan

Abbreviation	Program
UQHP	Unassisted Qualified Health Plan

> **Note:**
>
> Reports depend on the curam.intake.use.resilience application property that was introduced in Cúram 6.0.5.5. Reports are not intended for use with applications that pre-date this release.
>
> The application process is considered complete once an eligibility decision is made on the application. The report tracks applications up to that point. It does not track subsequent inconsistency period processing that may arise for some applications.
>
> Customization of these reports is not recommended. These reports rely on underlying functions that have the potential to change in future releases.

Monitoring HCR intake process instance errors

Use the HCR Intake Process Errors view to monitor technical problems that occur in the intake process. This view is an intake-oriented summary view of the process instance error queue that identifies the background workflow and deferred processes that are used in intake, and quantifies any instances of those processes that require technical intervention.

About this task

> **Important:** While these processes are the only ones used in Intake, some processes identified on the dashboard are shared processes. That is, they might also be used by other business processes outside Intake.

The workflow chart tracks all outstanding errors, so an error is presented here if it occurred in the selected time period and is not yet resolved. In contrast, deferred processing has no distinction between resolved and outstanding errors, so the deferred processing chart presents all errors that occurred over the selected time period.

From each process instance error count that is displayed on the dashboard, you can link to a list of process instance errors for that particular process. You can then do a root cause analysis and take remedial action.

Procedure

1. Log in to the Cúram Administration application as a user with administrator permissions.
2. Select **Administration Workspace** > **Process Monitoring** > **HCR Process Instance Errors** to open the view.

15.2 Monitoring Cúram processes

Use the following Cúram views to monitor and troubleshoot problems with process instances and to see process instance errors.

About this task

Use these views to see workflow processes and see specific errors in workflow and deferred processes. Plan to monitor the information in the following locations regularly for potential errors or exceptions. You can troubleshoot problems by steps such as suspending process instances or overriding event waits, or by retrying or aborting failed workflow process instances.

Monitoring workflow process instances

Use the **Process Instances** view to see the status of each workflow process instance. By searching and filtering, you can see the current process instances and their status. Generally, the complete or in-progress processes are of most interest.

About this task

For troubleshooting, you have the following options:

- You can suspend a process instance that is in progress. You must resume the process instance before any further activities can run.
- You can stop a process instance that is in progress. Once aborted, a process instance cannot be resumed.
- All activities that wait for events to be raised have a failure mode where the event they are waiting on is raised before the activity runs. To progress such process instances, you can override the event wait.

Procedure

1. Log in to the Cúram Administration application as a user with administrator permissions.
2. Select **Administration Workspace** > **Process Monitoring** > **Process Instances**
3. Use the search and filtering options to see the current workflow processes on the system.

Monitoring Process Instance Errors

Use the **Process Instance Errors** view to find workflow process or deferred process errors.

About this task

Plan to monitor the **Process Instance Errors** view regularly for potential operational errors or exceptions. You can abort or retry failed workflow process instances.

Procedure

1. Log in to the Cúram Administration application as a user with administrator permissions.
2. Select **Administration Workspace** > **Process Monitoring** > **Process Instance Errors**
3. Use the search and filtering options to find process instance errors.

4. Click the error details for more information.

Process Instance Errors

The Workflow Engine records information about errors that occur during the lifetime of a workflow process instance. You can use this information for troubleshooting problems with the process instance.

This troubleshooting includes retrying or aborting failed workflow process instances.

Retrying a failed process instance instructs the Workflow Engine to re-enact the workflow process instance from where it failed.

Aborting stops the process instance and its activities and closes any tasks that are associated with manual activities in the process instance. Depending on where the process was aborted, some manual steps might be required before the process is fully stopped.

16 Running a bulk reassessment of all open integrated cases

If you change the HCR system so that it affects numerous integrated cases, you can use the BulkICReassessment batch process to identify and process a full reassessment on all open integrated cases for HCR. For cases where the determination changes as a result of this reassessment, the new determination is stored and the old one superseded. A report is generated when the batch process completes.

Before you begin

> **Important:** By default, this batch process runs on all open integrated cases. Where appropriate, use a custom case selection strategy to limit the number of cases that are reassessed.

The BulkICReassessmentStream batch process supports batch streaming.

About this task

The following are some of the changes that can affect numerous integrated cases:

- Publishing CER rule set changes.
- Publishing CER product configuration changes.
- Publishing CER data configuration changes.
- Applying rate table changes.

Table 40: Default batch process report

Field	Description
Number of integrated cases selected	The number of integrated cases that were selected.
Number of integrated cases processed	The number of integrated cases that were successfully processed.
Number of integrated cases skipped	The number of integrated cases that were not successfully processed. That is, an error occurred that prevented reassessment.
*Number of products reassessed	The number of products that were reassessed.
*Number of products with decision changed	The number of products where the decision changed as a result of the reassessment.
*Number of products created	The number of products that were created as a result the reassessment.
*Number of products skipped	The number of products that were skipped during integrated case reassessment. Products are skipped if the certification period ends in a previous year.

* Product level counts are reported separately per product type. For example, the number of Streamlined Medicaid products that were reassessed.

Procedure

1. Choose a custom case selection strategy to limit the cases that are being reassessed, if appropriate for your environment.

   To customize integrated case selection, use the standard Guice dependency injection mechanism to implement `curam.healthcare.sl.impl.BulkICReassessmentCaseSelector`. Custom implementations extend `curam.healthcare.sl.impl.AbstractBulkICReassessmentCaseSelector` rather than directly implement the interface.

2. Optional: Turn notifications on or off for each integrated case that fails to reassess due to an error. Set the `curam.workflow.gendetermineeligibityfailureticket` environment property to `YES` to generate a notification for each skipped integrated case.

3. Optional: Customize the batch process reports as follows:

   a) You can customize the report messages text by updating the `/EJBServer/ components/HCR/messsage/BPOBatchBulkICReassessment.xml` message file. For more information about customizing

   b) Configure the batch report to count cases at a product level.

   Product level counts are recorded in the default implementation of `curam.healthcare.sl.impl.ProgramGroupManager`.

   Custom implementations can provide the product level counts by making the following calls at appropriate points during the reassessment. The `productType` is the code table string for the product. The `productID` is the ID of the product and is used here to prevent duplicate recording.

   - • Use `curam.healthcare.sl.impl.ICReassessmentCounters.get().incrementReassessedCo productID)` to increment the product reassessment count by 1.
   - • Use `curam.healthcare.sl.impl.ICReassessmentCounters.get().incrementCreatedCount productID)` to increment the product created count by 1.
   - • Use `curam.healthcare.sl.impl.ICReassessmentCounters.get().incrementDecisionChan product Id)` to increment the product decision changed count by 1.
   - • Use `curam.healthcare.sl.impl.ICReassessmentCounters.get().incrementSkippedCount productID)` to increment the product skipped count by 1.
   - •Use the `curam.healthcare.sl.impl.ICReassessmentCounters.get().clearCounters()` call before the reassessment of the integrated case.

4. Configure the batch process by setting the following environment variables to the appropriate values for the environment in which the batch process is running.

Property	Description	Default value
curam.batch.bulkicreassessment...	The number of integrated cases in each chunk.	5
curam.batch.bulkicreassessment...	In case the batch process sleeps while it is waiting for processing to complete.	No
curam.batch.bulkicreassessment...	The interval in milliseconds that the batch process waits before it retries to read the chunk key table.	1000
curam.batch.bulkicreassessment...	The interval in milliseconds that the batch process waits before it retries to read the chunk table for unprocessed chunks.	1000
curam.batch.bulkicreassessment...	In case the batch process attempts to process any unprocessed chunks that are found after all streams are completed.	No

5. Run the BulkICReassessment batch process.

   To ensure that the database is used to its full capacity, you can use the BulkICReassessmentStream batch process, which supports running in multiple streams to allow for concurrent execution on one or more computers.

6. Review the report.

16.1 BulkICReassessment

This batch process identifies and does a full reassessment on numerous open Income Support for Medical Assistance (Health Care Reform) integrated cases. You can run this process when changes affect numerous integrated cases, and you want to reassess the cases. For cases where the determination changes as a result of this reassessment, the new determination is stored and the old determination superseded.

> **Important:** By default, this batch process runs on all open Income Support for Medical Assistance (Health Care Reform) integrated cases. Where appropriate, use a custom case selection strategy to limit the number of cases that are reassessed.

Class and method

curam.healthcare.sl.impl.BulkICReassessment.process

Parameters

Parameter	Description	Default value
Batch Process Instance ID	A unique identifier that allows multiple instances of the same batch process to run at the same time effectively. When no instance ID is specified, only one instance of the batch process can run.	N/A

16.2 BulkICReassessmentStream

This batch process supports batch streaming for bulk integrated case reassessment in HCR and can run only with the BulkICReassessment batch process.

To start a stream for an instance of the Bulk IC Reassessment process, link the Bulk IC Reassessment batch process (or multiple stream batch processes) to the particular batch process instance by using the Batch Process Instance ID parameter. For example, where the Batch Process Instance ID is batch_reassessment_1 for an instance of the Bulk IC Reassessment batch process, you must also set the Batch Process Instance ID parameter for the Bulk IC Reassessment Stream batch process (or multiple stream batch processes) to batch_reassessment_1. Any number of Bulk IC Reassessment Stream batch processes can be linked to the same instance of the Bulk IC Reassessment batch process.

Class and Method

curam.healthcare.sl.intf.BulkICReassessmentStream.process

Parameters

Parameter	Description	Default value
Batch Process Instance ID	A unique identifier that allows multiple instances of the same batch process to run at the same time effectively. When no instance ID is specified, only one instance of the batch process can run.	N/A

17 Customizing the intake process

You can customize the default Health Care Reform intake process. For example, you can customize address mapping.

17.1 Address mapping

Use a custom implementation that extends the `curam.healthcare.intake.impl.AddressDataIntakeApplicationListener` class.

During the intake process, when the HCR IEG script is submitted from either the portal or the caseworker application address data is mapped from the data store to case evidence.

During the pre-mapping phase, the `curam.healthcare.intake.impl.AddressDataIntakeApplicationListener` is used. To facilitate the datastore-to-evidence mapping process, the class creates and updates address data in the data store. For more information, see the Javadoc of the class.

To customize the behavior of the class, use a custom implementation that extends the class. An example of a custom implementation that extends the class is shown.

```
public class CustomAddressDataIntakeApplicationListener
   extends curam.healthcare.intake.impl.AddressDataIntakeApplicationListener {

  @Override
  public void preMapDataToCuram(final IntakeApplication intakeApplication)
    throws AppException, InformationalException {

    super.preMapDataToCuram(intakeApplication);

    //
    // Add custom logic...
    //
  }
}
```

Then, bind the class in a Guice module. An example of binding the class in a Guice module is shown.

```
bind(AddressDataIntakeApplicationListener.class).to(CustomAddressDataIntakeApplicationListener.class)
```

Notices

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the Merative website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of Merative

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of Merative.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

Merative reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by Merative, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

MERATIVE MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Merative or its licensors may have patents or pending patent applications covering subject matter described in this document. The furnishing of this documentation does not grant you any license to these patents.

Information concerning non-Merative products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Merative has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Merative products. Questions on the capabilities of non-Merative products should be addressed to the suppliers of those products.

Any references in this information to non-Merative 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 Merative product and use of those websites is at your own risk.

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.

The licensed program described in this document and all licensed material available for it are provided by Merative under terms of the Merative Client Agreement.

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 Merative, 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. Merative, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. Merative shall not be liable for any damages arising out of your use of the sample programs.

Privacy policy

The Merative privacy policy is available at https://www.merative.com/privacy.

Trademarks

Merative ™ and the Merative ™ logo are trademarks of Merative US L.P. in the United States and other countries.

IBM®, the IBM® logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

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.

Oracle and Java are 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™, 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 company, product, and service names may be trademarks or service marks of others.