Merative Social Program Management 8.1

Cúram Person and Prospect Person Evidence Developers Guide

Note

Before using this information and the product it supports, read the information in Notices on page 45

Edition

This edition applies to Merative™ Social Program Management 8.0.0, 8.0.1, 8.0.2, 8.0.3, and 8.1.

Contents

Note iii

Edition v

1 Developing with Person and Prospect Person Evidence 9

1.1 Overview 9
Pre-requisites 9
Sections in this Guide 9
1.2 Person/Prospect Person Evidence Overview 10
Person and Prospect Person Data as Evidence 10
How Person or Prospect Person Evidence is Managed 10
1.3 Designing Person/Prospect Person Evidence Solutions 12
Data: Dynamic Evidence Types 12
Flow: Evidence Broker 13
Cúram Express Rules: Case Eligibility/Entitlement Calculations 14
1.4 Dynamic Evidence Type Data Mappings 14
Address 15
Bank Account 15
Birth and Death 15
Contact Preferences 16
Email Address 16
Gender 16
Identification 16
Name 17
Phone Number 17
Relationship 17
Snapshot Tables 18
1.5 Customizing Person/Prospect Person Evidence 18
Replicators 18
Converters 25
Evidence Sharing Automation 30
Selection of Primary Information 35
Reciprocal Evidence 37
Participant Data Case Owner 41

Notices 45

Privacy policy 46
Trademarks 46

1 Developing with Person and Prospect Person Evidence

Use this information to design a person/prospect person evidence solution. This work involves a consideration of the data, its structure, evidence constraints, and the flow of the data around the system. Some of the information that is stored about persons and prospect persons is held as evidence, which can be shared between cases on the system.

1.1 Overview

The purpose of this guide is to provide a high level technical understanding of person/prospect person evidence and its components. This guide also outlines the available customization options and extension points and provides instructions on how to implement these customizations. This guide is intended for developers and architects intending to implement a person/prospect person evidence solution.

Important: This guide is only applicable to those readers that are using the participant application with person and prospect person dynamic evidence.

Pre-requisites

The guide assumes that the reader is familiar with the following.

- *Cúram Evidence Guide*
- *Cúram Participant Guide*
- *Cúram Dynamic Evidence Configuration Guide*
- *Google Guice*

Sections in this Guide

The following list describes the sections within this guide:

- **Person/Prospect Person Evidence Overview**
  This section provides a high level overview of the key technical aspects of person/prospect person evidence.
- **Designing Person/Prospect Person Evidence Solutions**
  This section outlines some design considerations that should be taken into account when designing a person/prospect person evidence solution.
- **Dynamic Evidence Type Data Mappings**
  This section describes the mapping of data from the dynamic evidence types supplied with the application to legacy database tables.
- **Customizing Person/Prospect Person Evidence**
  This section describes the customization options and extension points available for person/prospect person evidence.

1.2 Person/Prospect Person Evidence Overview

Person and Prospect Person Data as Evidence

Some of the information stored about persons and prospect persons is held as evidence, which can be shared between cases on the system.

A number of Social Program Management components and technologies come together to enable the storing of person and prospect person evidence and the flow of this evidence through the system:

- Dynamic Evidence is used to store the captured person/prospect person data and perform basic validations.
- Cúram Express Rules (CER) are used to execute complex validations against the captured data.
- The Evidence Broker can optionally be used to broker the data between cases.
- Verifications can optionally be used to apply verifications to the captured data when it is brokered between cases.

Depending on the business requirements, some level of configuration, customization or both may be required. The section describes at a high level how the system manages person and prospect person data and identifies the points at which configuration and/or customization might be required.

Note: Careful consideration should be given to the required business behavior of the system during the design phase of a Social Program Management implementation. To develop an understanding of how to configure a system to support the business requirements, as a starting point, see the Cúram Participant Guide and Evidence.

How Person or Prospect Person Evidence is Managed

The management of person and prospect person data as evidence is underpinned by the following key foundations:

- Each person and prospect person has an associated person or prospect person case (Participant Data Case) created 'under the hood' following registration.
- Person and prospect person data is stored as evidence on dynamic evidence tables and is described by the dynamic evidence types that are associated with the person, a prospect person, or both.
- The data that is recorded as evidence is replicated back to the existing database tables; the existing database tables need to be in sync with the dynamic evidence.
- When a person or prospect person record is edited, the data is retrieved from, and written to, the dynamic evidence tables (and again, replicated back to the existing database tables).
- In some cases, application screens and processing continue to read from the existing database tables.
- The person and prospect person case types can be configured to have participant data brokered to and from other cases, by using the Evidence Broker.

Person/Prospect Person Evidence Types

A number of dynamic evidence types are provided and are associated with the person and prospect person participants. These evidence types and their attributes must not be removed or disassociated from the person and prospect person.

Where there is a requirement to manage additional data as person and prospect person evidence, new evidence types can be created as described in the Cúram Dynamic Evidence Configuration Guide and associated with the person and prospect person in the administration component. Equally, new attributes can be added to the existing dynamic evidence types. Where the data being added to new or existing evidence types is already present on an existing legacy database table, additional customization work must be performed:

- The code that replicates the data from the dynamic evidence tables to the legacy database tables (the 'replicator') must be extended to replicate the additional data being stored as evidence.
- Where data already exists on the legacy database tables, this data must be copied to the equivalent dynamic evidence table(s). The code that performs that operation (the 'converter') must be extended to convert the additional data into evidence.

More detail and example implementations for both of these are provided in the chapter 'Customizing Person/Prospect Person Evidence'.

Evidence Validations

Person and prospect person evidence is validated when created and edited.

These validations are implemented in one of two ways:

- Using the Dynamic Evidence Editor validations functionality. For example, mandatory field validations.
- Using Cúram Express Rule Sets. For example, cross-evidence validations.

Where new dynamic evidence types or attributes are added, customers should use one of these mechanisms to add any validations required. This is described in more detail in the Cúram Dynamic Evidence Configuration Guide.

When evidence is brokered to the person and prospect person, these validations are not checked. Brokered evidence is always accepted to prevent it being lost, as there is no concept of 'incoming evidence' for a person and prospect person. When person/prospect person evidence is entered, it is validated immediately. However, evidence brokered in from another case is automatically accepted and activated, even if the validation checks fail. For other case types, when person/prospect person evidence is brokered onto the case, a validation failure prevents the evidence from being activated.

Evidence Sharing

The evidence framework can share evidence between a person/prospect person, application cases and ongoing cases. The Evidence Broker enables and mediates this sharing of evidence. Evidence sharing is uni-directional and per evidence type. This means that different targets can receive and share an evidence type in different ways. If required, one case type might be able to receive shared evidence, but might not be able to share its own evidence.

There are three main functions which triggers the evidence broker to broadcast evidence:

- When a new person is added to a target case. For example, where person/prospect person evidence such as identification evidence is configured for sharing to an integrated case and a

person is added to an integrated case, the evidence broker first checks to see if that person has any person/prospect person evidence. If evidence is found, the evidence broker then checks for active identification evidence and shares it to the integrated case.
- When evidence changes are made to a source case. For example, when changes are made to a person's identification evidence, the evidence broker shares those changes to the integrated case.
- When a new target case is created. For example, any time a new integrated case is created, the evidence broker searches for person/prospect person identification evidence to be shared. If this evidence is found, the evidence broker shares the identification evidence to the integrated case.

For more information about the evidence broker, see .

1.3 Designing Person/Prospect Person Evidence Solutions

When designing a person/prospect person evidence solution the designer should consider the data, its structure, constraints and the flow of that data around the system.

Data: Dynamic Evidence Types

Structure

Person/Prospect Person evidence is primarily stored as dynamic evidence and the data structures that represent it are dynamic evidence types. Dynamic evidence types define the data, its type and behavior such as volatility, calculated attributes and so on. Once new dynamic evidence types are defined they must be activated and associated with the relevant case types, persons and prospect persons.

For more information about how to define dynamic evidence type, see Configuring dynamic evidence.

Things to consider:

- Does the evidence vary over time?
- Is the evidence type reciprocal? If so, the evidence type should have participant and related participant attributes.
- What case types should the evidence be available on?
- Consider making the evidence type 'Preferred', if it is to be commonly used. It allows case workers to quickly create evidence for frequently recorded evidence types.

Constraints

Validations

A number of standard validations, frequently used in evidence processing, are provided in the Dynamic Evidence Editor. More complex validations, such as cross-evidence validations, can be included using Cúram Express Rules.

More information on validations can be found in the Cúram Dynamic Evidence Configuration Guide.

Things to consider:

- What validations are required to ensure integrity of data?
- When person/prospect person evidence is entered, it is validated immediately; however, evidence brokered in from another case is automatically accepted and activated, even if the validation checks fail. For other case types, when person/prospect evidence is brokered onto the case, a validation failure prevents the evidence from being activated.
- Include any validations required to enforce succession constraints.
- Try to use standard validation patterns where possible. Validation rule sets should only be developed if they cannot be implemented using standard validations.
- If developing validation rule sets for more complex validations, be mindful of how data retrieval is performed. If performed incorrectly, this can have a significant impact on performance.

Important: System processes rely on the validations shipped with the application and it is not compliant to remove or alter these validations.

Verifications

This section is only applicable for those readers licensed to use the verifications component. Verification is the process of checking the accuracy of evidence.

The verification of evidence can take a number of forms; it can be provided by documents, for example, birth certificates or bank statements, or by verbal means, for example, telephone calls. When evidence is captured, the verification engine is invoked in order to determine if the evidence requires verification.

Note: With the exception of evidence brokered to a person/prospect person record, evidence cannot be activated until all mandatory verification requirements are met.

For more information on verifications and their configuration, see the Verification Guide.

Things to consider:

- Does the evidence require verification?
- What are the rules around verification?
- What information needs to be provided by the client?

Flow: Evidence Broker

The Evidence Broker is the mechanism that allows evidence to be shared throughout the system. When the Evidence Broker broadcasts evidence to a person/prospect person record, the evidence is automatically accepted and activated on the person/prospect person record, so the user does not have to manually accept and activate evidence.

For more information about the Evidence Broker and configuration options, see the Sharing evidence by using the evidence broker.

Things to consider:

- Is the same evidence type used on more than one case type? If so, should changes to this evidence be communicated to other cases?
- Should the target case be set up to automatically accept changes or should the case worker be forced to intervene to decide on whether to accept this incoming evidence?

- In order for system processing to function correctly, it is essential that person/prospect person data recorded outside of the participant manager be shared back to the participant manager.

Cúram Express Rules: Case Eligibility/Entitlement Calculations

Areas where Cúram Express Rules are used to read participant data from legacy database tables for the purposes of case eligibility and entitlement calculations, should be analyzed to decide where this data should be sourced from.

There are three options, each of which have their own benefits and limitations:

- Read participant data from the dynamic evidence stored by the participant manager
- Read participant data, which has been brokered onto cases
- Continue to read from the legacy tables

Read participant data from the dynamic evidence stored by the participant manager

Things to consider:

- Working off primary data source
- Changes in evidence causes immediate recalculations
- No opportunity for case worker to review

Read participant data which has been brokered onto cases

Things to consider:

- This is the recommended option for any new development
- Changes only take place when evidence is activated
- Evidence type has to be configured to be brokered onto the case

Continue to read from the legacy tables

Things to consider:

- This option should be considered carefully and is only recommended for upgrading customers.

1.4 Dynamic Evidence Type Data Mappings

The tables here show the data mappings from the dynamic evidence types to the legacy database tables.

Note: Replicators perform this mapping and converters perform the reverse mapping.

Address

Table 1: Address Mapping

Dynamic Evidence Attribute	Database Column
participant	ConcernRoleAddress.concernRoleID (calculated using caseParticipantRoleID)
address	Address.addressData
fromDate	ConcernRoleAddress.startDate
toDate	ConcernRoleAddress.endDate
addressType	ConcernRoleAddress.typeCode
comments	ConcernRoleAddress.comments

Bank Account

Table 2: Bank Account Mapping

Dynamic Evidence Attribute	Database Column
participant	ConcernRoleBankAccount.concernRoleID (calculated using caseParticipantRoleID)
accountName	BankAccount.name
accountNumber	BankAccount.accountNumber
iban	BankAccount.iban
accountType	BankAccount.typeCode
sortCode	BankAccount.bankSortCode
bic	BankAccount.bic
fromDate	BankAccount.startDate
toDate	BankAccount.endDate
accountStatus	BankAccount.bankAccountStatus
jointAccountInd	BankAccount.jointAccountInd
comments	BankAccount.comments

Birth and Death

Table 3: Birth and Death Mapping

Dynamic Evidence Attribute	Database Column
person	Person/ProspectPerson.concernRoleID (calculated using caseParticipantRoleID)
birthLastName	Person/ProspectPerson.personBirthName
mothersBirthLastName	Person/ProspectPerson.motherBirthSurname
dateOfBirth	Person/ProspectPerson.dateOfBirth
dateOfDeath	Person/ProspectPerson.dateOfDeath
comments	N/A

Contact Preferences

Table 4: Contact Preferences Mapping

Dynamic Evidence Attribute	Database Column
participant	ConcernRole.concernRoleID (calculated using caseParticipantRoleID)
preferredLanguage	ConcernRole.preferredLanguage
preferredCommunication	ConcernRole.prefCommMethod
comments	N/A

Email Address

Table 5: Email Address Mapping

Dynamic Evidence Attribute	Database Column
participant	ConcernRoleEmailAddress.concernRoleID (calculated using caseParticipantRoleID)
emailAddress	EmailAddress.emailAddress
fromDate	ConcernRoleEmailAddress.startDate
toDate	ConcernRoleEmailAddress.endDate
emailAddressType	ConcernRoleEmailAddress.typeCode
comments	EmailAddress.comments

Gender

Table 6: Gender Mapping

Dynamic Evidence Attribute	Database Column
person	Person/ProspectPerson.concernRoleID (calculated using caseParticipantRoleID)
gender	Person/ProspectPerson.gender
comments	N/A

Identification

Table 7: Identification Mapping

Dynamic Evidence Attribute	Database Column
participant	ConcernRoleAlternateID.concernRoleID (calculated using caseParticipantRoleID)
alternateID	ConcernRoleAlternateID.alternateID
altIDType	ConcernRoleAlternateID.typeCode
fromDate	ConcernRoleAlternateID.startDate
toDate	ConcernRoleAlternateID.endDate

Dynamic Evidence Attribute	Database Column
comments	ConcernRoleAlternateID.comments

Name

Table 8: Name Mapping

Dynamic Evidence Attribute	Database Column
participant	AlternateName.concernRoleID (calculated using caseParticipantRoleID)
title	AlternateName.title
firstName	AlternateName.firstForename
middleName	AlternateName.otherForename
lastName	AlternateName.surname
suffix	AlternateName.nameSuffix
initials	AlternateName.initials
nameType	AlternateName.nameType
comments	AlternateName.comments

Phone Number

Table 9: Phone Number Mapping

Dynamic Evidence Attribute	Database Column
participant	ConcernRolePhoneNumber.concernRoleID (calculated using caseParticipantRoleID)
phoneCountryCode	PhoneNumber.phoneCountryCode
phoneAreaCode	PhoneNumber.phoneAreaCode
phoneNumber	PhoneNumber.phoneNumber
phoneExtension	PhoneNumber.phoneExtension
fromDate	ConcernRolePhoneNumber.startDate
toDate	ConcernRolePhoneNumber.endDate
phoneType	ConcernRolePhoneNumber.typeCode
comments	PhoneNumber.comments

Relationship

Table 10: Relationship Mapping

Dynamic Evidence Attribute	Database Column
participant	ConcernRoleRelationship.concernRoleID (calculated using caseParticipantRoleID)
relatedParticipant	ConcernRoleRelationship.relConcernRoleID

Dynamic Evidence Attribute	Database Column
fromDate	ConcernRoleRelationship.startDate
toDate	ConcernRoleRelationship.endDate
relationshipType	ConcernRoleRelationship.relationshipType
endReason	ConcernRoleRelationship.relEndReasonCode
comments	ConcernRoleRelationship.comments

Snapshot Tables

When person/prospect person data is registered or maintained, this data is **not** replicated to the following snapshot tables.

- AlternateNameSnapshot
- ConcernRoleAddressSnapshot
- ConcernRoleAlternateIDSnapshot
- ConcernRoleBankAccountSnapshot
- ConcernRoleRelSnapshot
- ConcernRoleSnapshot
- PersonSnapshot
- ProspectPersonSnapshot

1.5 Customizing Person/Prospect Person Evidence

This section describes the customization options and extension points available for person/prospect person evidence. Some or all of these may be applicable to you depending on your existing customizations and configurations.

There are five main areas to consider, listed below:

- Replicators
- Converters
- Selection of Primary Information
- Reciprocal Evidence
- Participant Data Case Owner

Each of these areas are described in detail and examples are also provided. **Please note, these are samples only.**

Replicators

What is a Replicator?

A replicator reflects changes in evidence onto the relevant legacy tables for the purposes of backward compatibility. The replicator takes the dynamic evidence details and converts them to a struct containing the details to be stored on the legacy tables. These details are then written to the relevant database tables, thus ensuring that the information on the legacy tables is in sync with the primary data source, the dynamic evidence. Default replicator implementations are provided for each of the person/prospect person evidence types.

These default implementations contain extension points to allow replication to custom fields, which is covered in the following section.

Note: Only the last version in a succession set is used to replicate data to the legacy tables.

Replicator Extension

Why Extend a Replicator? - In cases where legacy database tables have been extended, it may be necessary to extend a replicator.

It is possible to extend the replicators supplied with the application to allow replication of person/prospect person evidence to custom database columns. Interfaces are available for each supplied evidence type and can be found in the package curam.pdc.impl, listed here.

Custom implementations can be written to use these interfaces, depending on the evidence type.

Replicator Extender Interfaces:

- PDCAddressReplicatorExtender
- PDCAlternateIDReplicatorExtender
- PDCAlternateNameReplicatorExtender
- PDCBankAccountReplicatorExtender
- PDCBirthAndDeathReplicatorExtender
- PDCContactPreferencesReplicatorExtender
- PDCEmailAddressReplicatorExtender
- PDCGenderReplicatorExtender
- PDCPhoneNumberReplicatorExtender
- PDCRelationshipsReplicatorExtender

The majority of the interfaces have one method `assignDynamicEvidenceToExtendedDetails`. It accepts two parameters:

- dynamicEvidenceDataDetails - the dynamic evidence details
- details - the struct containing the extended details for the legacy table

`PDCBirthAndDeathReplicatorExtender` and `PDCGenderReplicatorExtender` have two methods, `assignDynamicEvidenceToExtendedPersonDetails` and `assignDynamicEvidenceToExtendedProspectPersonDetails`. `assignDynamicEvidenceToExtendedPersonDetails` accepts two parameters:

- dynamicEvidenceDataDetails - the dynamic evidence details
- details - the struct containing the extended person details for the legacy table

`assignDynamicEvidenceToExtendedProspectPersonDetails` also accepts two parameters:

- dynamicEvidenceDataDetails - the dynamic evidence details
- details - the struct containing the extended prospect person details for the legacy table

Example: Implementing a Person/Prospect Person Evidence Replicator Extender

The following example outlines how to extend a replicator to map person/prospect person evidence to custom fields. This example provides a very basic implementation of an extension to the `PDCPhoneNumberReplicatorExtender`. In this scenario, the PhoneNumber table is

extended and contains a custom field 'phoneProvider'. The dynamic evidence configuration for Phone Number also contains an attribute 'phoneProvider'. This example assumes that the struct `ParticipantPhoneDetails` is already extended to include the custom field. For more information on dynamic evidence configuration, see the Cúram Dynamic Evidence Configuration Guide. The responsibility of the custom replicator extension implementation is to map the dynamic evidence data to the struct attribute that represents the 'phoneProvider' attribute on the extended PhoneNumber table.

Note: A mapping of data is all that is necessary; the default implementation performs the actual replication of data.

The steps involved in extending a replicator are:

- Provide a replicator extension implementation that maps the custom data back to the legacy table
- Add a binding to the new replicator extension implementation

Step 1: Provide a Replicator Extension Implementation

The first step is to provide a new implementation that implements the relevant replication extension interface for the evidence type and maps the custom data back to the legacy table. The code snippet here demonstrates a custom implementation for `PDCPhoneNumberReplicatorExtender`. It simply assigns the value of the dynamic evidence attribute to the phoneProvider struct attribute. This information is then inserted along with the other dynamic evidence attributes through the default implementation for `PDCPhoneNumberReplicatorExtender`.

```
public class SampleReplicatorExtenderImpl
  implements PDCPhoneNumberReplicatorExtender {

  public void assignDynamicEvidenceToExtendedDetails(
    DynamicEvidenceDataDetails dynamicEvidenceDataDetails,
    ParticipantPhoneDetails details)
    throws AppException, InformationalException {

    details.phoneProvider =
    dynamicEvidenceDataDetails.getAttribute("phoneProvider").getValue();
  }
}
```

Step 2: Add a Binding to the New Replicator Extension Implementation

Guice bindings are used to register the implementation.

```
public class SampleModule extends AbstractModule {

public void configure() {

  // Register the replicator extension implementation
  Multibinder<PDCPhoneNumberReplicatorExtender>
sampleReplicatorExtender =
  Multibinder.newSetBinder(binder(),
PDCPhoneNumberReplicatorExtender.class);

  sampleReplicatorExtender.addBinding().
to(SampleReplicatorExtenderImpl.class);
 }
}
```

Note: New Guice modules must be registered by adding a row to the ModuleClassName database table. See the Persistence Cookbook for more information.

Why Implement a Replicator?

In cases where new dynamic evidence types are introduced, it may be necessary to implement a new replicator.

Implementing a Replicator

Replicators can be easily developed to cater for scenarios such as new dynamic evidence types. A detailed example is provided in the next section and outlines the steps and artifacts necessary to get a new replicator up and running. Replicator implementations are invoked through an event based mechanism.

When dynamic evidence is activated after an insert, modify or remove, an event is thrown. For new evidence types an event listener needs to be developed to listen for this event and invoke the replication process, this is discussed in more detail later in this section. The next section demonstrates how to implement a replicator.

Example: Implementing a Person/Prospect Person Evidence Replicator

The following example outlines how to implement a replicator. In this scenario, Sample Foreign Residency is configured as a new dynamic evidence type. For more information on how to configure a new evidence type, see the Cúram Dynamic Evidence Configuration Guide. The new Sample Foreign Residency evidence type has the following attributes,

- participant - the case participant role id of the person/prospect person that the evidence is being entered for
- country - the country of residency
- fromDate - the date the residency started
- toDate - the date the residency ended
- reason - the reason for residency in this country

It is assumed that this dynamic evidence type is activated and is configured for use with person/prospect person. Until now Sample Foreign Residency information is stored as static evidence on the SampleForeignResidency database table. It is now necessary to store this information as dynamic evidence. A new replicator may be required to replicate evidence changes to the legacy database table so that this table is in sync with the dynamic evidence.

The steps involved in implementing a replicator are:

- Provide a replicator interface for the dynamic evidence type
- Provide a replicator implementation that replicates dynamic evidence to the legacy database table
- Implement an event listener that triggers the replication
- Add a binding to the new event listener implementation

Step 1: Provide a Replicator Interface

The new replicator interface should contain three methods -

`replicateInsertEvidence` which replicates activated inserted Sample Foreign Residency evidence to the Sample Foreign Residency legacy database table. It accepts one parameter:

- evidenceDescriptorDtls the activated evidence descriptor details

`replicateModifyEvidence` which replicates activated modified Sample Foreign Residency evidence to the Sample Foreign Residency legacy database table. It accepts two parameters:

- evidenceDescriptorDtls the activated evidence descriptor details
- previousActiveEvidDescriptorDtls the evidence descriptor details for the evidence that was active before the modify

`replicateRemoveEvidence` which replicates activated removed Sample Foreign Residency evidence to the Sample Foreign Residency legacy database table. It accepts one parameter:

- evidenceDescriptorDtls the activated evidence descriptor details

```
@ImplementedBy(SampleForeignResidencyReplicatorImpl.class)
public interface SampleForeignResidencyReplicator {

  public void replicateInsertEvidence(
    final EvidenceDescriptorDtls evidenceDescriptorDtls)
    throws AppException, InformationalException;

  public void replicateModifyEvidence(
    final EvidenceDescriptorDtls evidenceDescriptorDtls,
    final EvidenceDescriptorDtls previousActiveEvidDescriptorDtls)
    throws AppException, InformationalException;

  public void replicateRemoveEvidence(
    final EvidenceDescriptorDtls evidenceDescriptorDtls)
    throws AppException, InformationalException;
}
```

Step 2: Provide a Replicator Implementation

The replicator implementation should provide implementations for the three methods described in the previous section. These methods should convert the dynamic evidence data to data suitable to be written to the legacy database tables and update the legacy tables for this evidence type.

```
public class SampleForeignResidencyReplicatorImpl
  implements SampleForeignResidencyReplicator {

  protected SampleForeignResidencyReplicatorImpl() {
  }

  public void replicateInsertEvidence(
    final EvidenceDescriptorDtls evidenceDescriptorDtls)
    throws AppException, InformationalException {

    SampleForeignResidency sampleForeignResidencyObj =
   SampleForeignResidencyFactory.newInstance();
    SampleForeignResidencyDtls sampleForeignResidencyDtls =
   new SampleForeignResidencyDtls();
    UniqueID uniqueIDObj = UniqueIDFactory.newInstance();

    EvidenceControllerInterface evidenceControllerObj =
      (EvidenceControllerInterface) EvidenceControllerFactory
.newInstance();

    EIEvidenceKey eiEvidenceKey = new EIEvidenceKey();
    eiEvidenceKey.evidenceID = evidenceDescriptorDtls.relatedID;
    eiEvidenceKey.evidenceType = evidenceDescriptorDtls.
evidenceType;

    EIEvidenceReadDtls eiEvidenceReadDtls =
      evidenceControllerObj.readEvidence(eiEvidenceKey);

    DynamicEvidenceDataDetails dynamicEvidenceDataDetails =
      (DynamicEvidenceDataDetails) eiEvidenceReadDtls.
evidenceObject;

    sampleForeignResidencyDtls.countryCode =
      dynamicEvidenceDataDetails.getAttribute("country").getValue();

    sampleForeignResidencyDtls.startDate =
   (Date) DynamicEvidenceTypeConverter.convert(
      dynamicEvidenceDataDetails.getAttribute("fromDate"));

    sampleForeignResidencyDtls.endDate =
   (Date) DynamicEvidenceTypeConverter.convert(
      dynamicEvidenceDataDetails.getAttribute("toDate"));

    sampleForeignResidencyDtls.reasonCode =
      dynamicEvidenceDataDetails.getAttribute("reason")
.getValue();

    sampleForeignResidencyDtls.concernRoleID =
evidenceDescriptorDtls.participantID;
    sampleForeignResidencyDtls.foreignResidencyID =
uniqueIDObj.getNextID();
    sampleForeignResidencyDtls.statusCode =
RECORDSTATUS.NORMAL;

    sampleForeignResidencyObj.insert(sampleForeignResidencyDtls);
  }

  public void replicateModifyEvidence(
    final EvidenceDescriptorDtls evidenceDescriptorDtls,
    final EvidenceDescriptorDtls
previousActiveEvidDescriptorDtls)
    throws AppException, InformationalException {

    List<SampleForeignResidencyKey> sampleForeignResidencyKeyList =
      new ArrayList<SampleForeignResidencyKey>();

    SampleForeignResidencyDtls sampleForeignResidencyDtls =
   new SampleForeignResidencyDtls();

    EvidenceControllerInterface evidenceControllerObj =
      (EvidenceControllerInterface)
EvidenceControllerFactory.newInstance();

    EIEvidenceKey eiEvidenceKey = new EIEvidenceKey();
    eiEvidenceKey.evidenceID =
previousActiveEvidDescriptorDtls.relatedID;
    eiEvidenceKey.evidenceType =
```

Step 3: Implement an Event Listener

A new event listener needs to be implemented to listen for events raised of type Sample Foreign Residency that occur as a result of evidence activation. This listener should implement the interface curam.pdc.impl.PDCEvents and provide implementations for the three methods. This is where the replication process can be kicked off and any other custom processing that may need to happen.

```
public class SampleForeignResidencyEventsListener
  implements PDCEvents {

  @Inject
  private SampleForeignResidencyReplicator
sampleForeignResidencyReplicator;

  public void insertedEvidenceActivated(
    EvidenceDescriptorDtls evidenceDescriptorDtls)
    throws AppException, InformationalException {

    if (evidenceDescriptorDtls.evidenceType.equals
("SAMPLEFOREIGNRESIDENCY")) {
      sampleForeignResidencyReplicator.replicateInsertEvidence
(evidenceDescriptorDtls);
    }
  }

  public void modifiedEvidenceActivated(
    EvidenceDescriptorDtls evidenceDescriptorDtls,
    EvidenceDescriptorDtls previousActiveEvidDescriptorDtls)
    throws AppException, InformationalException {

    if (evidenceDescriptorDtls.evidenceType.equals
("SAMPLEFOREIGNRESIDENCY")) {
      sampleForeignResidencyReplicator.replicateModifyEvidence
(evidenceDescriptorDtls,
      previousActiveEvidDescriptorDtls);
    }
  }

  public void removedEvidenceActivated(
    EvidenceDescriptorDtls evidenceDescriptorDtls)
    throws AppException, InformationalException {

 if (evidenceDescriptorDtls.evidenceType.equals
("SAMPLEFOREIGNRESIDENCY")) {
      sampleForeignResidencyReplicator.replicateRemoveEvidence
(evidenceDescriptorDtls);
    }
  }
}
```

Step 4: Add a Binding to the New Event Listener Implementation

Guice bindings are used to register the implementation.

```
public class SampleModule extends AbstractModule {

  public void configure() {

    // Register the event listener
    Multibinder<PDCEvents> sampleEventListeners =
      Multibinder.newSetBinder(binder(), PDCEvents.class);

    sampleEventListeners.addBinding().to(
      SampleForeignResidencyEventsListener.class);
  }
}
```

Note: New Guice modules must be registered by adding a row to the ModuleClassName database table. See the Persistence Cookbook for more information.

Converters

What is a Converter? - A converter is a mechanism for converting legacy person/prospect person data to dynamic evidence. When legacy database tables are populated external to the application, by using tools such as the Cúram Data Manager(DMX files), converters can be used to convert this data to dynamic evidence.

Default converter implementations are provided for each of the person/prospect person evidence types. These default implementations contain extension points to allow conversion of custom fields, which is covered in the following section.

Why Extend a Converter?

In cases where legacy database tables are extended, it may be necessary to extend a converter. Converters are generally only used in a development environment or for upgrade tooling and should not be used as part of everyday processing.

Converter Extension

The converters provided with the application can be extended to allow conversion of custom database columns to person/prospect person dynamic evidence. Interfaces are available for each evidence type and can be found in the package curam.pdc.impl, these are listed below. Custom implementations can be written that make use of these interfaces, depending on the evidence type.

Converter Extension (Populator) Interfaces:

- PDCAddressEvidencePopulator
- PDCAlternateIDEvidencePopulator
- PDCAlternateNameEvidencePopulator
- PDCBankAccountEvidencePopulator
- PDCBirthAndDeathEvidencePopulator
- PDCContactPreferencesEvidencePopulator
- PDCEmailAddressEvidencePopulator
- PDCGenderEvidencePopulator
- PDCPhoneNumberEvidencePopulator
- PDCRelationshipsEvidencePopulator

The majority of the interfaces have one method `populate`. It accepts varying parameters depending on the evidence types.

`PDCBirthAndDeathEvidencePopulator` and `PDCGenderEvidencePopulator`, interfaces have two methods, `populatePerson` and `populateProspectPerson`.

`populatePerson` accepts four parameters:

- concernRoleKey - unique identifier for the concern role that this evidence is relating to
- caseIDKey - the unique identifier of the Participant Data Case
- personDtls - the struct containing the extended person details from the legacy table
- dynamicEvidenceDataDetails - the dynamic evidence details

`populateProspectPerson` also accepts four parameters:

- concernRoleKey - unique identifier for the concern role that this evidence is relating to
- caseIDKey - the unique identifier of the Participant Data Case
- prospectPersonDtls - the struct containing the extended prospect person details from the legacy table
- dynamicEvidenceDataDetails - the dynamic evidence details

Example: Implementing a Person/Prospect Person Evidence Populator

The following example outlines how to extend a converter to map custom database columns to person/prospect person evidence. This example provides a very basic implementation of an extension to `PDCPhoneNumberEvidencePopulator`. In this scenario, the PhoneNumber table is extended and contains a custom column 'phoneProvider'. The dynamic evidence configuration for Phone Number also contains an attribute 'phoneProvider'. The responsibility of the custom populator implementation is to convert the struct attribute that represents the 'phoneProvider' attribute on the extended PhoneNumber table to dynamic evidence data. For more information on dynamic evidence configuration, see the Cúram Dynamic Evidence Configuration Guide.

Note: The conversion of data is all that is necessary, the default converters will look after the actual storage of the dynamic evidence.

The steps involved in extending a converter are:

- Provide a populator implementation that converts the custom field from the legacy table to dynamic evidence data
- Add a binding to the new populator implementation

Step 1: Provide a Populator Implementation

The first step is to provide a new implementation that implements the relevant populator interface for the evidence type and converts the custom field from the legacy table to dynamic evidence. The code snippet below demonstrates the custom implementation for the `PDCPhoneNumberEvidencePopulator`, it converts the phoneProvider struct attribute to the dynamic evidence equivalent attribute. This dynamic evidence is then stored along with the other dynamic evidence attributes through the default converter implementation.

```
public class SamplePopulatorImpl
  implements
PDCPhoneNumberEvidencePopulator {

  public void populate(
    ConcernRoleKey concernRoleKey,
CaseIDKey caseIDKey,
    ConcernRolePhoneNumberDtls
concernRolePhoneNumberDtls,
    PhoneNumberDtls phoneNumberDtls,
    DynamicEvidenceDataDetails dynamicEvidenceDataDetails)
    throws AppException, InformationalException {

    DynamicEvidenceDataAttributeDetails phoneProvider =
      dynamicEvidenceDataDetails.getAttribute("phoneProvider");

    DynamicEvidenceTypeConverter.setAttribute(phoneProvider,
      phoneNumberDtls.phoneProvider);
  }
}
```

Add a Binding to the New Populator Implementation

Guice bindings are used to register the implementation.

```
public class SampleModule extends AbstractModule {

public void configure() {

  // Register the populator implementation
  Multibinder<PDCPhoneNumberEvidencePopulator> samplePopulator =
  Multibinder.newSetBinder(binder(), PDCPhoneNumberEvidencePopulator.class);

  samplePopulator.addBinding().to(SamplePopulatorImpl.class);
  }
}
```

Note: New Guice modules must be registered by adding a row to the ModuleClassName database table. See the Persistence Cookbook for more information.

Why Implement a Converter?

In cases where new dynamic evidence types are introduced, it may be necessary to implement a new converter. Converters are generally only used in a development environment or for upgrade tooling and should not be used as part of everyday processing.

Implementing a Converter

Converter implementations can be developed using the `PDCConverter` interface. The `PDCConverter` interface can be found in the curam.pdc.impl package. This interface has one method `storeEvidence`.

It accepts two parameters:

- concernRoleKey - the unique identifier of the concern role
- caseIDKey - the unique identifier of the Participant Data Case.

The next section demonstrates how to implement a converter.

Example: Implementing a Person/Prospect Person Evidence Converter

The following example outlines how to implement a converter. In this scenario, Sample Foreign Residency is configured as a new dynamic evidence type. For more information on how to configure a new evidence type, see the Cúram Dynamic Evidence Configuration Guide. The new Sample Foreign Residency evidence type has the following attributes:

- participant - the case participant role id of the person/prospect person that the evidence is being entered for
- country - the country of residency
- fromDate - the date the residency started
- toDate - the date the residency ended
- reason - the reason for residency in this country

It is assumed that this dynamic evidence type is activated and is configured for use with person/prospect person. Sample Foreign Residency information was previously stored as static evidence on the SampleForeignResidency database table. It is now necessary to store this information as dynamic evidence. A new converter is required which takes this information from the legacy table and converts and stores it as dynamic evidence.

The steps involved in implementing a converter are:

- Provide a converter implementation that converts the legacy data to dynamic evidence
- Add a binding to the new converter implementation

Step 1: Provide a Converter Implementation

The code snippet demonstrates the implementation for the `PDCConverter`. It retrieves all Sample Foreign Residency information for a person/prospect person from the legacy

SampleForeignResidency table, converts this information to a dynamic evidence data structure, and stores the resulting information.

```
public class SampleForeignResidencyConverterImpl
  implements PDCConverter {

  @Inject
  private EvidenceTypeDefDAO etDefDAO;

  @Inject
  private EvidenceTypeVersionDefDAO etVerDefDAO;

  public void storeEvidence(ConcernRoleKey concernRoleKey, CaseIDKey caseIDKey)
    throws AppException, InformationalException {

    PDCCaseIDCaseParticipantRoleID pdcCaseIDCaseParticipantRoleID =
      new PDCCaseIDCaseParticipantRoleID();

    ParticipantDataCase participantDataCaseObj =
ParticipantDataCaseFactory.newInstance();
    pdcCaseIDCaseParticipantRoleID.caseID =
      participantDataCaseObj.getParticipantDataCase(concernRoleKey).caseID;

    CaseIDTypeCodeKey caseIDTypeCodeKey = new CaseIDTypeCodeKey();
    caseIDTypeCodeKey.caseID = pdcCaseIDCaseParticipantRoleID.caseID;
    caseIDTypeCodeKey.typeCode = CASEPARTICIPANTROLETYPE.PRIMARY;

    pdcCaseIDCaseParticipantRoleID.caseParticipantRoleID =
      CaseParticipantRoleFactory.newInstance().readByCaseIDAndTypeCode
(caseIDTypeCodeKey).dtls.caseParticipantRoleID;

    SampleForeignResidency sampleForeignResidencyObj =
 SampleForeignResidencyFactory.newInstance();

    SampleForeignResidencyReadMultiKey sampleForeignResidencyReadMultiKey =
      new SampleForeignResidencyReadMultiKey();
    sampleForeignResidencyReadMultiKey.concernRoleID =
 concernRoleKey.concernRoleID;

    SampleForeignResidencyReadMultiDtlsList sampleForeignResidencyList =
      sampleForeignResidencyObj.
searchByConcernRole(sampleForeignResidencyReadMultiKey);

   for (SampleForeignResidencyReadMultiDtls
sampleForeignResidencyReadMultiDtls :
sampleForeignResidencyList.dtls) {

      final EvidenceTypeKey eType = new EvidenceTypeKey();
      eType.evidenceType = "SampleForeignResidency";

      EvidenceTypeDef evidenceType =
        etDefDAO.readActiveEvidenceTypeDefByTypeCode(eType.evidenceType);

      EvidenceTypeVersionDef evTypeVersion =
        etVerDefDAO.getActiveEvidenceTypeVersionAtDate(evidenceType,
 Date.getCurrentDate());

      DynamicEvidenceDataDetails dynamicEvidenceDataDetails =
        DynamicEvidenceDataDetailsFactory.newInstance(evTypeVersion);

      DynamicEvidenceDataAttributeDetails participant =
        dynamicEvidenceDataDetails.getAttribute("participant");

      DynamicEvidenceTypeConverter.setAttribute(participant,
        pdcCaseIDCaseParticipantRoleID.caseParticipantRoleID);

      DynamicEvidenceDataAttributeDetails country =
        dynamicEvidenceDataDetails.getAttribute("country");

      DynamicEvidenceTypeConverter.setAttribute(country,
        sampleForeignResidencyReadMultiDtls.countryCode);

      DynamicEvidenceDataAttributeDetails fromDate =
        dynamicEvidenceDataDetails.getAttribute("fromDate");

      DynamicEvidenceTypeConverter.setAttribute(fromDate,
sampleForeignResidencyReadMultiDtls.startDate);

      DynamicEvidenceDataAttributeDetails endDate =
        dynamicEvidenceDataDetails.getAttribute("toDate");

      DynamicEvidenceTypeConverter.setAttribute(endDate,
sampleForeignResidencyReadMultiDtls.endDate);
```

Step 2: Add a Binding to the New Converter Implementation

Guice bindings are used to register the implementation.

```
public class SampleModule extends AbstractModule {

  public void configure() {

  // Register the converter implementation
  Multibinder<PDCConverter> sampleForeignResidencyConverter =
    Multibinder.newSetBinder(binder(), PDCConverter.class);

  sampleForeignResidencyConverter.addBinding().
to(SampleForeignResidencyConverterImpl.class);
  }
}
```

Note: New Guice modules must be registered by adding a row to the ModuleClassName database table. See the Persistence Cookbook for more information.

Evidence Sharing Automation

What is Evidence Sharing Automation?

Evidence sharing automation is the ability of an evidence record to be shared between cases and be automatically activated, without being blocked by validations or requiring manual intervention by a system user.

The result of automated sharing is one of....

1. The shared evidence is ignored because the same information is already recorded on the target case.
2. The evidence is deemed new and therefore a new evidence record is created on the target case.
3. An existing evidence record on the target case is identified as a match and is updated to reflect the new information received in the shared evidence.

This automation reflects the process that a caseworker would go through in considering new evidence.

Why use Evidence Sharing Automation?

In normal circumstances when evidence is configured to be shared between cases the sharing can be configured to auto-activate. This means that the evidence is activated on the target case if possible. If it is not possible, for example a validation rule blocks activation, then the evidence is moved to an in-edit state allowing the case worker to manually process the evidence.

On a Person case there is no 'in-edit state' for evidence, instead evidence is created in an active state. To enable the automatic activation on sharing of evidence the validations that would normally block activation must be overcome. This is achieved by implementing an automation strategy for the evidence type.

For more on the automation of evidence sharing for the Person case, refer to the Cúram Participant Guide.

Implementing an Automation Strategy

The objective of the automation strategy is to decide on 1 of 3 outcomes for the shared evidence.

1. The shared evidence is ignored because the same information is already recorded on the target case.
2. The evidence is deemed new and a new evidence record is created on the target case.
3. An existing evidence record on the target case is identified as a match and is updated to reflect the new information received in the shared evidence.

To identify which option to take for an evidence type, the sharing process calls on the evidence-type-specific strategy to make the correct decision.

The PDCEvidence interface is provided as the contract through which an evidence type defines its automation strategy. Each evidence type that requires an automation strategy, that is, any evidence type that is configured for use on the Person case, implements this interface describing how that evidence type is handled when shared to or from the Person case.

Before implementing an automation strategy for an evidence type you must first decide on what the strategy is. This is typically the task of a business analyst who decides what rules should be applied to decide whether the shared evidence is ignored, inserted, or results in an existing evidence record being modified. After the strategy has been defined it can be implemented by the PDCEvidence interface.

Refer to the Javadoc of the curam.pdc.impl.PDCEvidence interface for further details on how to use the interface to define the automation strategy for the evidence type.

Note: When evidence that is configured on a Person case is shared, the business validations that are normally executed to prevent conflicting data from being captured are switched off. The automation strategy that you employ here is the replacement for those validations. If no custom strategy is provided for an evidence type the default implementations of the strategy is employed. The default strategy may be inappropriate for the evidence type, and this can lead to conflicting evidence, such as duplicate records being added to the case.

Note: The sharing strategy is also employed for logically equivalent sharing, that is, where an evidence record of one type is shared to an evidence of a different type. The automation strategy must cater for this type of sharing if it is expected that this sharing configuration exists. For example, when comparing fields between records of different types you may need to differentiate between an attribute that doesn't exist on one of the evidence types versus an attribute that does not match. In this scenario your business rules may choose to treat the source attribute with no corresponding target attribute as matching even though a comparison cannot be made.

For more information about defining logically equivalent evidence, see the related link.

Related information

Configuring an Automation Strategy

Follow these steps to configure a new automation strategy.

1. Retrieve the identifier for the evidence type that the automation strategy is applied. This can be done using the below SQL

```
SELECT code FROM CodeTableItem WHERE TableName='EvidenceType' and
 Description='<Evidence Name>'
```

   For example, SELECT code FROM CodeTableItem WHERE TableName='EvidenceType' and Description='Gender Details' returns "PDC0000262".

2. Bind automation strategy to the evidence type code by adding to an existing or creating a new Guice Module

```
public class EvidenceSharingAutomationModule  extends AbstractModule {

  @Override
   public void configure() {
     final MapBinder< String, PDCEvidence> pdcEvidenceMapBinder =
      MapBinder.newMapBinder(binder(),  String.class, PDCEvidence.class);

     String genderDetailsCode =  "PDC0000262";
     // bind evidence type to evidence automation strategy

 pdcEvidenceMapBinder.addBinding(genderDetailsCode).to(PDCGenderEvidenceImpl.class);
   }
}
```

   To override an existing implementation use a linked binding to bind the original implementation to the custom implementation.

```
@Override
  public void configure() {

    bind(PDCGenderEvidenceImpl.class).to(MyCustomGenderEvidenceImpl.class);

  }
```

Example Automation Strategy

The following example is taken from the Gender Evidence strategy that is applied to the product evidence type Gender Details.

```
package curam.pdc.impl;


 import java.util.Set;

 import com.google.inject.Singleton;

 import curam.codetable.EVIDENCENATURE;
 import curam.codetable.impl.EVIDENCENATUREEntry;
 import curam.core.sl.infrastructure.impl.EIEvidenceReadDtls;
 import curam.dynamicevidence.impl.DynamicEvidenceDataAttributeDetails;
 import curam.dynamicevidence.impl.DynamicEvidenceDataDetails;
 import curam.evidence.impl.EvidenceAttributeDataDetails;
 import curam.util.exception.AppException;
 import curam.util.exception.InformationalException;
 import curam.util.type.Date;


/**
 * Gender evidence sharing automation strategy
 */
@Singleton
class PDCGenderEvidenceImpl  extends AbstractPDCEvidenceImpl {

   // ___________________________________________________________________________
  /**
   * Defines  if this evidence is multi-timeline per type. Gender evidence does not
 support having more
   * than one gender at the same time, so returns  false.
   */
  @Override
   public boolean isMultipleTimeLinePerType(DynamicEvidenceDataDetails
 sharedEvidenceDetails)  throws AppException,
      InformationalException {
     return false;
   }

  /**
   * Compare the shared evidence to evidence on the target  case for an exact match.
   *
   * Returns  true if there is an exact match. If there is an exact match the
   * evidence will not be shared.
   *
   */
  @Override
   public boolean matchAllEvidenceDetails( final DynamicEvidenceDataDetails
 sharedEvidenceDetails,  final Date sharedEffectiveDate,
   final DynamicEvidenceDataDetails originalEvidenceDetails,  final Date
 originalEffectiveDate)
   throws AppException, InformationalException {

    // Default implementation will compare all fields to determine exact match.
 // No need to implement this  method unless you want to change the definition of an
 exact match.
 // This implementation is just here for  illustrative purposes.
 return super.matchAllEvidenceDetails(sharedEvidenceDetails, sharedEffectiveDate,
 originalEvidenceDetails, originalEffectiveDate);
   }

   // ___________________________________________________________________________
  /**
   * Compare the shared evidence to evidence on the target  case for a 'partial' match.
   *
   * Returns  true if there is a partial match. If there is a partial match, the
 evidence
   * sharing process will  continue, but may fail to share the evidence based on other
 conditions.
   */
  @Override
   public boolean matchEvidenceDetails(
     final DynamicEvidenceDataDetails sharedEvidenceDetails,  final Date
 sharedEffectiveDate,
     final DynamicEvidenceDataDetails originalEvidenceDetails,  final Date
 originalEffectiveDate)
     throws AppException, InformationalException {

    EvidenceAttributeDataDetails sourceEvidenceAttributeDetails =  new
 EvidenceAttributeDataDetails(
       sharedEvidenceDetails, EVIDENCENATUREEntry.get(EVIDENCENATURE.DYNAMIC));
     EvidenceAttributeDataDetails targetEvidenceAttributeDetails =  new
 EvidenceAttributeDataDetails(
      originalEvidenceDetails, EVIDENCENATUREEntry.get(EVIDENCENATURE.DYNAMIC));
```

Selection of Primary Information

Legacy participant manager entities have the notion of primary indicators, where users are able to specify which bank account/phone number and so forth represents the primary data when the evidence is created. This is not the case with dynamic evidence types. The user does not specify the primary record; instead there is an algorithm in the background that calculates which should be the primary record.

These algorithms are based on a defined business strategy and can be modified, details of which are outlined in the following section.

Why Change the Selection of Primary Information?

As the identification of the primary record is not user-driven, it can be necessary to modify this selection process, if the default business strategy is not preferable.

Changing the Selection of Primary Information

The strategies that determine which data should be selected as primary information can be modified by using the default primary handler interfaces.

An interface is defined for each dynamic evidence type supplied that has a primary identifier on its legacy table, found in the curam.pdc.impl package and are listed here.

Primary handler implementations are started with an event based mechanism. When dynamic evidence is activated after an insert, modify or remove operation, an event is thrown. For new evidence types an event listener needs to be developed to listen for this event and start the appropriate algorithm that determines the primary data, this is discussed in more detail later in this section. The next section demonstrates how to implement a primary handler.

Primary Handler Interfaces:

- PDCPrimaryAddressHandler
- PDCPrimaryAlternateIDHandler
- PDCPrimaryAlternateNameHandler
- PDCPrimaryBankAccountHandler
- PDCPrimaryEmailAddressHandler
- PDCPrimaryPhoneNumberHandler

Changing the Selection of Primary Information Example

This example outlines how to implement a primary handler. In this scenario, the defined business strategy for selecting a primary phone number is to select the phone number with the latest start date.

The steps involved in implementing a primary handler are:

- Provide a primary handler implementation that identifies the primary record
- Add a binding to the new primary handler implementation

Step 1: Provide a Primary Handler Implementation

The first step is to provide a new implementation that implements the relevant primary handler interface for the evidence type and identifies the primary record. The code snippet demonstrates

an implementation for `PDCPrimaryPhoneNumberHandler`, it takes the phone number with the latest start date and sets it as the primary record.

```
public class SamplePrimaryPhoneNumberHandlerImpl
  implements PDCPrimaryPhoneNumberHandler {

  protected SamplePrimaryPhoneNumberHandlerImpl() {
  }

  public void setPrimaryPhoneNumber
(EvidenceDescriptorDtls evidenceDescriptorDtls)
    throws AppException, InformationalException {

    ConcernRoleKey concernRoleKey = new ConcernRoleKey();
    concernRoleKey.concernRoleID =
evidenceDescriptorDtls.participantID;

    ConcernRolePhoneNumberDtlsList concernRolePhoneNumberDtlsList =
      ConcernRolePhoneNumberFactory.newInstance().
searchByConcernRole(concernRoleKey);

    ConcernRole concernRoleObj = ConcernRoleFactory.newInstance();
    ConcernRoleDtls concernRoleDtls = concernRoleObj.read(concernRoleKey);
    Date currentPrimaryPhoneNumberStartDate = Date.kZeroDate;

    List<SampleSortedPhoneNumber> list =
new ArrayList<SampleSortedPhoneNumber>();

    for (ConcernRolePhoneNumberDtls
concernRolePhoneNumberDtls:concernRolePhoneNumberDtlsList.dtls) {

      PhoneNumberKey phoneNumberKey = new PhoneNumberKey();
      phoneNumberKey.phoneNumberID = concernRolePhoneNumberDtls.phoneNumberID;

      if (concernRolePhoneNumberDtls.phoneNumberID ==
concernRoleDtls.primaryPhoneNumberID) {
        currentPrimaryPhoneNumberStartDate = concernRolePhoneNumberDtls.startDate;
      }

      SampleSortedPhoneNumber sampleSortedPhoneNumber =
        new SampleSortedPhoneNumber(concernRolePhoneNumberDtls);
      list.add(sampleSortedPhoneNumber);
    }

    Collections.sort(list);

    SampleSortedPhoneNumber newPrimaryPhoneNumber = list.get(0);

    if (newPrimaryPhoneNumber.getStartDate().
after(currentPrimaryPhoneNumberStartDate)) {
      concernRoleDtls.primaryPhoneNumberID =
newPrimaryPhoneNumber.getPhoneNumberID();
      concernRoleObj.pdcModify(concernRoleKey, concernRoleDtls);
    }
}
  class SampleSortedPhoneNumber implements
Comparable<SampleSortedPhoneNumber> {
    private long phoneNumberID;
    private Date startDate;

    SampleSortedPhoneNumber(ConcernRolePhoneNumberDtls dtls) {
      this.phoneNumberID = dtls.phoneNumberID;
      this.startDate = dtls.startDate;
    }

    public long getPhoneNumberID() {
      return phoneNumberID;
    }

    public Date getStartDate() {
      return startDate;
    }

    public int compareTo(SampleSortedPhoneNumber o) {
      return o.getStartDate().compareTo(this.getStartDate());
    }
  }
}
```

Step 2: Add a Binding to the New Primary Handler Implementation

Guice bindings are used to register the implementation.

```
public class SampleModule extends AbstractModule {

  public void configure() {

    // Register the primary handler implementation
    bind(PDCPrimaryPhoneNumberHandler.class).to(
      SamplePrimaryPhoneNumberHandlerImpl.class);
  }
}
```

Note: New Guice modules must be registered by adding a row to the ModuleClassName database table. See the Persistence Cookbook for more information.

Reciprocal Evidence

What is Reciprocal Evidence? - Reciprocal evidence is a type of evidence which consists of two pieces of evidence that must be processed together. The relationship dynamic evidence type is an example of reciprocal evidence.

When Person A is recorded as a spouse of Person B, the corresponding relationship evidence, Person B is a spouse of Person A is recorded. When evidence is inserted, modified or removed for Person A, the system inserts, modifies or removes the corresponding relationship evidence for Person B.

Why Provide a Reciprocal Evidence Implementation?

If you develop a new reciprocal dynamic evidence type, then you must also provide an implementation of the `ReciprocalEvidenceConversion` interface.

Reciprocal Evidence Implementations

When evidence is inserted, modified or removed a hook point is invoked, that by default triggers the reciprocal evidence handler functionality. This new evidence hook point is called the `GlobalEvidenceHook` and can be found in the curam.core.sl.infrastructure.impl package. The `GlobalEvidenceHook` Interface allows custom processing to occur after evidence operations complete.

GlobalEvidenceHook Interface:

The `GlobalEvidenceHook` interface contains the following methods:

`postInsertEvidence` is invoked after evidence is inserted and accepts two parameters:

- caseKey - the identifier of the case that the evidence belongs to
- evKey - the identifier and type of the evidence

`postModifyEvidence` is invoked after evidence is modified and accepts two parameters:

- caseKey - the identifier of the case that the evidence belongs to
- evKey - the identifier and type of the evidence

`postRemoveEvidence` is invoked after evidence is removed and accepts two parameters:

- caseKey - the identifier of the case that the evidence belongs to

- evKey - the identifier and type of the evidence

`postDiscardPendingUpdate` is invoked after a pending update of evidence is discarded and accepts two parameters:

- caseKey - the identifier of the case that the evidence belongs to
- evKey - the identifier and type of the evidence

`postDiscardPendingRemove` is invoked after a pending remove of evidence is discarded and accepts two parameters:

- caseKey - the identifier of the case that the evidence belongs to
- evKey - the identifier and type of the evidence

Reciprocal Evidence Handler:

The default implementation for the `GlobalEvidenceHook` invokes the reciprocal evidence handler functionality. The reciprocal evidence handler is responsible for all common reciprocal evidence processing. It locates reciprocal evidence and if found performs the same changes on it that were performed on the original evidence. If the reciprocal evidence is not found, and the original evidence was inserted, then it inserts the corresponding reciprocal evidence. As the reciprocal evidence handler is core to the reciprocal evidence processing it cannot be customized directly, but can be customized by way of the `GlobalEvidenceHook`, if necessary.

Reciprocal Evidence Conversion Interface:

The `ReciprocalEvidenceConversion` interface is responsible for reciprocal and original evidence comparison, participant retrieval and for creating new and modified reciprocal evidence from original evidence. To make custom evidence reciprocal, a `ReciprocalEvidenceConversion` interface implementation must be provided. While the handler is not aware of the internal evidence structure, the conversion interface implementation is, as a result this is where the main customization point lies. The `ReciprocalEvidenceConversion` interface can be found in the curam.core.sl.infrastructure.impl package and contains the following methods:

- Object getReciprocal(final Object original, final long targetCaseID) - Creates reciprocal evidence details from the original evidence details
- Object getUpdatedReciprocal(final Object original, final Object unmodifiedReciprocal) - Creates modified reciprocal evidence details from the original evidence details and from un-modified reciprocal evidence details
- long getPrimaryParticipant(final Object originalEvidence) - Retrieves the primary participant (concern role ID) from the original evidence. Note that the primary participant on the original evidence is the related participant on the reciprocal evidence
- long getRelatedParticipant(final Object originalEvidence) - Retrieves related participant (concern role ID) from the original evidence. Note that the related participant on the original evidence is the primary participant on the reciprocal evidence
- boolean matchEvidenceDetails(final Object evidenceDetails1, final Object evidenceDetails2) - Checks evidence details for a match. Implementation of this method determines if two evidence details passed in are considered as a match.

The following section demonstrates how to implement reciprocal evidence.

Reciprocal Evidence Implementation Example

The following example demonstrates a reciprocal evidence implementation. In this scenario, Working Relationship is configured as a new dynamic evidence type.

For more information on how to configure a new evidence type, see Configuring dynamic evidence. The new Working Relationship evidence type is identified as reciprocal and has the following attributes,

- participant - the case participant role id of the person/prospect person that the evidence is being entered for
- relatedParticipant - the case participant role id of the related person/prospect person
- workingRelationship - the working relationship between the two participants

It is assumed that this dynamic evidence type is activated and is configured for use with person/prospect person. In order for this reciprocal evidence to be handled correctly by the infrastructure, an implementation of the `ReciprocalEvidenceConversion` interface must be provided.

The steps that are involved:

- Provide a reciprocal evidence conversion implementation
- Add a binding to the new reciprocal evidence conversion implementation

Step 1: Provide a Reciprocal Evidence Conversion Implementation

```
public class SampleWorkingRelationshipReciprocalConversion
  implements ReciprocalEvidenceConversion {

  @Inject
  private EvidenceTypeDefDAO etDefDAO;

  @Inject
  private EvidenceTypeVersionDefDAO etVerDefDAO;

  public SampleWorkingRelationshipReciprocalConversion() {

  }

  public Object getReciprocal(
    final Object original, final long targetCaseID)
    throws AppException, InformationalException {

    DynamicEvidenceDataDetails originalDetails =
 (DynamicEvidenceDataDetails) original;

    String workingRelationshipOriginal =
      originalDetails.getAttribute
("workingRelationship").getValue();

    String workingRelationshipRec = "";

    if (workingRelationshipOriginal.equals("ISMANAGEROF")) {
      workingRelationshipRec = "ISMANAGEDBY";
    }

    EvidenceTypeKey evdType = new EvidenceTypeKey();
    evdType.evidenceType = "WORKINGRELATIONSHIP";

    EvidenceTypeDef evdTypeDef =
      etDefDAO.readActiveEvidenceTypeDefByTypeCode
(evdType.evidenceType);

    EvidenceTypeVersionDef evTypeVersion =
      etVerDefDAO.getActiveEvidenceTypeVersionAtDate
(evdTypeDef, Date.getCurrentDate());

    DynamicEvidenceDataDetails reciprocalDetails =
      DynamicEvidenceDataDetailsFactory.newInstance
(evTypeVersion);

    DynamicEvidenceDataAttributeDetails workingRelationshipAttr =
      reciprocalDetails.getAttribute("workingRelationship");

    DynamicEvidenceTypeConverter.setAttribute
(workingRelationshipAttr, workingRelationshipRec);

    DynamicEvidenceDataAttributeDetails participantAttr =
      reciprocalDetails.getAttribute("participant");

    DynamicEvidenceTypeConverter.setAttribute(participantAttr,
        originalDetails.getAttribute
("relatedParticipant").getValue());

    DynamicEvidenceDataAttributeDetails relatedParticipantAttr =
      reciprocalDetails.getAttribute("relatedParticipant");

    DynamicEvidenceTypeConverter.setAttribute
(relatedParticipantAttr,
        originalDetails.getAttribute("participant").getValue());

    return reciprocalDetails;
  }

  public Object getUpdatedReciprocal(
    final Object original, final Object unmodifiedReciprocal)
    throws AppException, InformationalException {

    DynamicEvidenceDataDetails originalDetails =
      (DynamicEvidenceDataDetails) original;
    DynamicEvidenceDataDetails reciprocalDetails =
      (DynamicEvidenceDataDetails) unmodifiedReciprocal;

    long caseParticipantRoleIDRec = Long.parseLong(
      reciprocalDetails.getAttribute("participant").getValue());
    long relCaseParticipantRoleIDRec = Long.parseLong(
      reciprocalDetails.getAttribute
("relatedParticipant").getValue());
    String workingRelationshipRec =
```

Step 2: Add a Binding to the New Reciprocal Evidence Conversion Implementation

Guice bindings are used to register the implementation.

```
public class SampleModule extends AbstractModule {

  public void configure() {

    MapBinder<CASEEVIDENCEEntry,
ReciprocalEvidenceConversion> recEvidenceConversionMapBinder =
      MapBinder.newMapBinder(binder(),
CASEEVIDENCEEntry.class, ReciprocalEvidenceConversion.class);

    reciprocalEvidenceConversionMapBinder.addBinding(
   CASEEVIDENCEEntry.get("WORKINGRELATIONSHIP")).to(
      SampleWorkingRelationshipReciprocalConversion.class);
  }
}
```

Note: New Guice modules must be registered by adding a row to the ModuleClassName database table. See the Persistence Cookbook for more information.

Reciprocal Evidence Limitations

The reciprocal evidence handling infrastructure has the following limitations:

- The evidence must be temporal evidence. It can be static, dynamic, or generated evidence.
- The evidence must have a participant and related participant, alternatively, the `ReciprocalEvidenceConversion` implementation code must be able to determine the participant and related participant by using the evidence details.
- When reciprocal evidence and its related original evidence are both on the same case then their changes must be always applied together, otherwise original and reciprocal evidence data are out of sync.
- Reciprocal evidence can be processed automatically only if both related participants are registered as MEMBER or PRIMARY participants on the same case, or evidence is recorded as person/prospect person evidence.
- The reciprocal handler is supported for evidence types that are modeled to allow corrections only.

Participant Data Case Owner

Why Change the Participant Data Case Owner?

It may be necessary to change the participant data case owner if tighter control is required around the ownership of participants.

Changing the Participant Data Case Owner

When a person/prospect person is registered on the system, a case is created in the background to help manage this data, this is also known as a 'Participant Data Case'.

By default, this case has a case owner, the logged in user. It is possible to change this to a different case owner by way of the `PDCCaseOwnerAssignmentStrategy` Interface. The `PDCCaseOwnerAssignmentStrategy` Interface can be found in the curam.pdc.impl package and has one method `createOwner`. It accepts two parameters:

- key - the identifier of the Participant Data Case

- ownerDtls - the details of the Participant Data Case owner

Changing the Participant Data Case Owner Example

The example here outlines how to change the Participant Data Case Owner, in this scenario the owner is set to the system user.

The steps that are involved:

- Provide a case owner assignment strategy implementation that sets the case owner
- Add a binding to the case owner assignment strategy implementation

Step 1: Provide a Case Owner Assignment Strategy Implementation

The code snippet demonstrates a sample implementation for `PDCCaseOwnerAssignmentStrategy`, it sets the owner to be the system user.

```
@Singleton
public class SampleCaseOwnerAssignmentStrategyImpl
  implements PDCCaseOwnerAssignmentStrategy {

  public void createOwner(CaseHeaderKey key,
OrgObjectLinkDtls ownerDtls)
    throws AppException, InformationalException {

    ownerDtls.orgObjectType = ORGOBJECTTYPE.USER;
    ownerDtls.userName = UserAccessFactory.newInstance().
getSystemUserDetails().userName;

    OrgObjectLinkFactory.newInstance().insert(ownerDtls);

    OrgObjectLinkKey orgObjectLinkKey = new OrgObjectLinkKey();
    orgObjectLinkKey.orgObjectLinkID = ownerDtls.orgObjectLinkID;

    CaseUserRoleDtls caseUserRoleDtls =
 new CaseUserRoleDtls();
    caseUserRoleDtls.caseID = key.caseID;
    caseUserRoleDtls.orgObjectLinkID =
 orgObjectLinkKey.orgObjectLinkID;
    caseUserRoleDtls.typeCode = CASEUSERROLETYPE.OWNER;
    caseUserRoleDtls.recordStatus = RECORDSTATUS.NORMAL;

    curam.core.sl.entity.fact.CaseUserRoleFactory.newInstance()
.insert(caseUserRoleDtls);

    CaseHeader caseHeaderObj = CaseHeaderFactory.newInstance();
    CaseHeaderDtls caseHeaderDtls = caseHeaderObj.read(key);
    caseHeaderDtls.ownerOrgObjectLinkID =
 orgObjectLinkKey.orgObjectLinkID;
    caseHeaderObj.modify(key, caseHeaderDtls);
  }
}
```

Step 2: Add a Binding to the Case Owner Assignment Strategy Implementation

Guice bindings are used to register the implementation.

```
public class SampleModule extends AbstractModule {

  public void configure() {

    // Register the implementation
    bind(PDCCaseOwnerAssignmentStrategy.class)
   .to(SampleCaseOwnerAssignmentStrategyImpl.class);
  }
}
```

Note: New Guice modules must be registered by adding a row to the ModuleClassName database table. See the Persistence Cookbook for more information.

Setting the Case Owner for Participant Data Cases Created by System Processes

Use the values in the application properties to set the case owner where a system process creates the Participant Data Case.

In some situations, a system process rather than a specific user creates the Participant Data Case. A deferred process or batch is an example of where a system process creates the Participant Data Case. In situations where a system process creates the Participant Data Case, set the case owner by using the values that are specified in the application properties `curam.case.owner.system.user.type` and `curam.case.owner.system.user.reference`.

The following table specifies how to set the case owner by using the values that are specified in the application properties.

Table 11: Setting the case owner by using the values that are specified in the application properties.

Application property	Step to perform
`curam.case.owner.system.user.type`	Specify the user type by using a value from the code table `OrgObjectType`. User, organization unit, position, and work queue are examples of user type.
`curam.case.owner.system.user.reference`	Specify the unique reference of the user. A username and a position ID are examples of the unique reference of the user.

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.