IBM Cúram Social Program Management
Version 7.0.2

Cúram Evidence Broker Developers Guide

IBM

Note

Before using this information and the product it supports, read the information in

Contents

List of Figures

List of Tables

Chapter 1. Developing with Evidence Broker

Use this information to learn about the Evidence Broker and how it is implemented. The Evidence Broker enables flexible sharing of evidence between programs and systems to ensure that the most up-to-date evidence changes are available. Before evidence sharing can occur, it is necessary to define the evidence types available for sharing and to configure how this sharing occurs.

Introduction

Purpose

The purpose of this guide is to provide a high level understanding of Cúram Evidence Broker and its components. This guide also describes how custom evidence can be shared using the Cúram Evidence Broker.

Audience

This guide is for architects and developers responsible for implementing evidence sharing.

Prerequisites

The reader should be familiar with the business requirements for evidence sharing and how the Cúram Evidence Broker works. For a high-level overview, see the `Cúram Evidence Broker Guide`.

Chapters in this Guide

The following list describes the chapters within this guide:

Evidence Broker Architecture
This chapter provides a high level overview of the key technical aspects of the Cúram Evidence Broker.

Implementing the Evidence Comparison Interface
This chapter outlines the steps for implementing the Evidence2Compare interface.

Implementing Transfer Evidence
This chapter discusses the implementation of the transferEvidence evidence interface operation and why it is required.

Using the Broadcast Evidence Hook
This chapter looks at the Broadcast Evidence hook which allows customers route the evidence broadcast through their custom processing.

Implementing Evidence Sharing Strategy Interface
This chapter provides high level instructions on how to implement the EvidenceSharingStrategy interface.

Evidence Broker Web Service
This chapter provides a high level overview of the evidence broker web service that facilitates evidence sharing with remote systems.

Evidence Broker Architecture

Introduction

This chapter describes the architecture of the Cúram Evidence Broker.

Architecture

Evidence Broker enables flexible sharing of evidence between programs and systems to ensure that the most up to date evidence changes are available to the configured programs and systems improving the speed and accuracy at which changes are propagated. The system and programs that share the evidence are called source and the ones that receive these updates are called target. Source and target could refer to the same system if the system supports multiple programs.

Following are the key components of Evidence Broker:

Change Notification Interface
This evidence broker web services interface is used to share and accept evidence changes from and to remote systems. An API version of this interface exists for more performant sharing if the source and target are the same system.

Evidence Broker Sharing Configuration
Evidence broker sharing configuration allows systems to configure systems, programs, and evidence types as the source and target for evidence sharing.

Evidence Broker Sharing Strategy
A case type specific sharing strategy to allow organizations additional flexibility in providing consent to sharing of evidence changes. A default sharing strategy exists that uses evidence broker configuration data to determine the sharing targets.

Evidence Broker Broadcast Hook
An evidence type specific hook that is invoked while applying evidence changes to the target case.

Evidence Sharing Steps

Following are the high levels steps in the evidence sharing process.

Detect Change
The evidence sharing process starts after a change of evidence is detected, typically when the user approves evidence, in the source system.

Apply Sharing Strategy
A case type specific sharing strategy is invoked to determine targets configured for sharing and to notify the targets of the change. The default sharing strategy uses evidence broker sharing configuration to determine targets and to notify them of the change. The sharing strategy is executed in a deferred process to avoid any performance impact on the evidence approval process.

Notify Change
As part of the sharing strategy, the change is notified to the target system via a web service call or an API depending on if the sharing is done across systems or not.

If sharing is via web services, the sources system creates an XML document containing the change details. In case the target system is the same as the source system and API call to Evidence Broker with the identifier of the changed source evidence is required for change notification.

Process Change Notification
After receiving the change notification the target system validates the details by applying the appropriate schema or loading the evidence details from the database. Evidence broker then checks the configuration to see if it is allowed to accept the specific evidence type changes from the target system.

If an appropriate configuration exists, Evidence Broker determines the cases in that might be impacted from the change and invokes the evidence type specific hook (Broadcast Evidence hook) to process the evidence application on the target case. Evidence type specific transferEvidence operation on EvidenceInterface is called to map various keys from the source to the target case.

Synchronize Change
A task is sent to the caseworker if an action is required in applying the shared evidence. The caseworker then using the synchronization screens applies the changes to the target case. The synchronization screens use the Evidence Compare interface to return the data source and target evidence data in a format that is conducive to presenting it on the screen in a user friendly manner.

Evidence Broker Sharing Strategy

Each time evidence is shared from the source case Evidence Broker invokes the case type specific Evidence Sharing Strategy. This allows, at a case type level, flexibility in deciding how the sharing should take place. A default sharing strategy that is appropriate for most situations is included out of the box. Custom implementation of the sharing strategy can institute a consent model that uses customer specific logic to determine if a particular evidence can be shared and also decide on specific targets that it can be shared with. The transport mechanism of changes from source to the target can also be modified using a custom strategy.

To facilitate easy creation of new strategies a helper class ProcessEvidenceHelper has been provided. This class contains reusable code and provides various helper functions thought to be useful in creating a new strategy.

Evidence Compare Interface

When a user selects the Compare link on the Synchronization screen of the Evidence Broker, the Evidence2Compare interface identifies all the pieces of evidence for comparison and returns the evidence comparison data in a format that can be understood by the evidence comparison screen. The Evidence Broker API determines which evidence records need to be returned for comparison. It has been enhanced to transform the data returned from the Evidence2Compare interface into xml format to be understood by the evidence comparison screens.

Evidence generated by the Cúram Evidence Generator will implement the Evidence2Compare interface. Customers not using the Cúram Evidence Generator need to ensure their custom evidence, which is being shared, implements the interface. They also need to provide the necessary handcrafted functionality in its implementation which builds up the comparison data to be transformed into xml by the Evidence API.

Transfer Evidence

The transferEvidence operation, which is one of the functions on the Evidence Interface, handles the foreign keys on a custom entity when evidence is broadcast from one case to another. For example, if a custom entity has one or more case participant role fields, code needs to exist in this function to manage the foreign keys. This is so these fields on the new record on the target case do not point at case participants on the source case. For evidence generated by the Cúram Evidence Generator, the code for managing the foreign keys will be automatically generated.

Note: It should be noted that the transferEvidence interface operation was originally added for the transfer evidence functionality. The code, whether it be generated or handcrafted, should cater for both the transferring and broadcasting of evidence. The transferring of evidence can take place without the Evidence Broker being installed.

Broadcast Evidence Hook

The Broadcast Evidence hook allows customers provide an alternative mechanism for broadcasting evidence. Any time the evidence broker is triggered to look for incoming evidence available for sharing, this hook will be called before the evidence is broadcast to the target case. Customers can use this hook to call processing that is usually invoked when evidence is added to a case. Customers may want to invoke this same processing when evidence is shared on cases. For example, a workflow may be invoked as part of an evidence insert, either pre or post, which initiates other events.

The Cúram Evidence Generator automatically inserts in the create evidence service layer functions a pre and post step for calling custom processing before and after the evidence is created. These steps apply to general evidence creation and shared evidence creation. When the Cúram Evidence Generator is not used, customers can still implement a hook by handcrafting pre and post steps in their own create evidence business processes. Customers should update their existing create evidence processes to distinguish between evidence which is being shared and evidence which is being inserted.

Integration with Evidence Generator

The Cúram Evidence Broker has been integrated with the Cúram Evidence Generator to streamline the implementation of evidence sharing. When evidence is generated, it can be shared without any custom code having to be written aside from listing the classes of evidence types in the Evidence2Compare Registrar. The Cúram Evidence Generator automatically implements the Evidence2Compare interface on the generated service layer. It provides implementations for the Evidence2Compare interface for every generated evidence type. The evidence generator also provides implementations of the transferEvidence operation, where required, on the entity layer. This generation saves a considerable amount of development time.

From the perspective of the Evidence API, integration with the evidence generator automatically makes the create evidence business process 'evidence sharing' aware. The generated service layer create functions can recognize the difference between inserted evidence and evidence broadcast from a source case. As described in the previous section, this allows customers to use the same pre and post steps of the insert evidence function for the broadcast evidence.

Evidence Broker Configuration

Evidence Broker configuration can be set up manually by an administrator. This is done by enabling sharing when assigning evidence types to cases and by setting up the source and target evidence types (as described in the `Cúram Evidence Broker Guide`).

Implementing Evidence Compare Interface

Introduction

The purpose of this chapter is to provide instructions on how to implement the Evidence2Compare interface. Most of the instructions in this chapter relate to evidence sharing that is not generated. When evidence is generated, the only step required is adding evidence type / class pairings to the Evidence2Compare registrar.

Identify Evidence Types Available for Sharing

Before evidence sharing can occur, it is necessary to define the evidence types available for sharing and to configure how this sharing will occur. More specifically this includes defining the source and target evidence types and cases (see the `Cúram Evidence Broker Guide` for more information).

Determine Classes which Implement Evidence Compare Interface

The evidence generator implements the Evidence2Compare interface at the service layer. Some custom evidence may not have a service layer, in which case it is possible to implement the interface at the facade or entity layer. To implement at the facade layer, it is necessary to have a separate facade for each evidence type since a single implementation cannot cater for multiple evidence types. There are no limitations for entities as the entity to evidence type relationship is one-to-one.

Here is a sample declaration of the Sample Sporting Activity Evidence2Compare implementation which lives on the SampleMaintainSportingActivity facade layer:

```
/**
    * Facade methods for the Sample Sporting Grant Activity
    * product.
    */
   public class SampleMaintainSportingActivity
     extends curam.sample.facade.base.SampleMaintainSportingActivity
     implements Evidence2Compare {
     .
     .
     //_____
     /**
```

```
        * Return details that will comprise the XML blob used to
        * populate the evidence comparison screen inside the
        * Evidence Broker.
        *
        * @param key Identifies an evidence entity
        * @return Evidence entity details
        */
      public EvidenceComparisonDtls getComparisonData(
        EvidenceCaseKey key)
        throws AppException, InformationalException {
    }
  }
```

Map Evidence Compare Classes to Evidence Type

After deciding the classes which implement the Evidence2Compare interface, it is necessary to add these classes to the Evidence2Compare map that provides a look up for the implementing class using the evidence type. This can be done in two ways: using Google Guice module or by a registrar. Though both approaches achieve the same goal, the Google Guice route is preferred over the registrar route. Typically the registrar route should be used when overriding the OOTB implementation. Both approaches are outlined below using the Sample Sporting Activity evidence type referred to in the previous section:

Using Google Guice

This can be done by creating a Guice module class and adding a corresponding entry in the MODULE table. A Guice module class is created by deriving a class from com.google.guice.AbstractModule and overriding the configure method to add the following statement:

```
MapBinder<String, Method> evidence2CompareMapBinder =
    MapBinder.newMapBinder(binder(), String.class, Method.class,
        new RegistrarImpl(RegistrarType.EVIDENCE_TO_COMPARE));

    evidence2CompareMapBinder.addBinding(CASEEVIDENCE.SAMPLEADDRESS)
        .toInstance(SampleAddressFactory.class.getMethod(
            ReflectionConst.kNewInstance, new Class[0]));
```

Where 'RegistrarType.EVIDENCE_TO_COMPARE' is an annotation which is used to differentiate between various registrar maps.

It is not necessary to create a new module for each of such hooks you have to bind. Single module class per component will work well.

Using Registrar

Define the following method in a class:

```
public void registerEvidence2Compare() throws
    AppException, InformationalException {
      Evidence2CompareMap map =  EvidenceController.
      getEvidence2CompareMap();
      map.putEvidenceType(CASEEVIDENCE.SAMPLEADDRESS,
      SampleAddressFactory.class);
    }
```

The class which implements the registrar must be added to the ENV_EVIDENCE2COMPARE_REGISTRARS_LIST environment variable. Out-of-the-box, for example the facade class curam.sample.sl.fact.SampleSportingGrantEvidenceRegistrarFactory, is added to the ENV_EVIDENCE2COMPARE_REGISTRARS_LIST variable. Further additions should be added in a comma delimited fashion, with no space left between the comma and the next addition to the list.

Provide Implementation for the Evidence Compare Interface

One of the main benefits of using the evidence generator is that developers do not have to provide an implementation for the Evidence2Compare interface. Without the evidence generator, this can be a time consuming task, particularly when sharing a large number of evidence types.

Values

Developers must write code which gets the relevant values, i.e., attributes from the evidence entity, and put them into a struct that can be transformed into xml by the Evidence Broker API for evidence comparison purposes.

Labels

Developers must create an entity.properties file, <Entity>Labels.properties, per evidence type. This should contain the attribute name and label for that name which will be displayed on the evidence comparison screen. Like all property files, the label is localizable.

```
readDtls.clientDtls.name=Client Name
readDtls.sportingActivityType=Sporting Activity Type
readDtls.sportingAwardType=Sporting Award Type
readDtls.paymentAmount=Payment Amount
readDtls.startDate=Start Date
readDtls.endDate=End Date
readDtls.comments=Comments
```

Domains

Customers don't need to implement domains with a resource bundle. They could just as easily use java constants. Labels however, must be localizable, so it makes sense for them to do it this way. The generated naming convention for domains is <Entity>Domains.properties. These are generated to the service layer impl code package (alongside the code that uses them). An example of a domains file is shown below

```
readDtls.clientDtls.name=FULL_NAME
readDtls.sportingActivityType=SAMPLE_SPORT_ACT_TYPE
readDtls.sportingAwardType=SAMPLE_SPORT_AWRD_TYPE
readDtls.paymentAmount=CURAM_AMOUNT
readDtls.startDate=CURAM_DATE
readDtls.endDate=CURAM_DATE
readDtls.comments=COMMENTS
```

Names

The Names attribute refers to the name that is used internally in the implementation to reference a particular attribute on an evidence record. The Names attribute is distinct from the Labels attribute, which references the text in the attribute label that is displayed to users.

Sample Implementation

Here is a sample implementation of the Evidence2Compare interface

```
//_____
/**
 * Return details that will comprise the XML blob
 * used to populate the evidence comparison screen
 * inside the Evidence Broker.
 *
 * @param key Identifies an evidence entity
 * @return Evidence entity details
 */
public EvidenceComparisonDtls getComparisonData(EvidenceCaseKey
  key) throws AppException, InformationalException {

  EvidenceComparisonDtls evidenceComparisonDtls =
    new EvidenceComparisonDtls();

  SampleSportingActivityKey sampleSportingActivityKey =
    new SampleSportingActivityKey();
  sampleSportingActivityKey.sportingActivityID =
    key.evidenceKey.evidenceID;
```

```
SampleViewSportingActivityDtls readDtls =
  readSampleSportingActivityEvidence(
    sampleSportingActivityKey);

EvidenceDescriptorKey evidenceKey =
  new EvidenceDescriptorKey();
evidenceKey.evidenceDescriptorID =
  readDtls.evidenceDescriptorID;

EvidenceDescriptorDtls evidenceDtls =
  EvidenceControllerFactory.newInstance()
    .readEvidenceDescriptorDtls(evidenceKey);

evidenceComparisonDtls.descriptor.assign(evidenceDtls);

evidenceComparisonDtls.descriptor.updatedBy =
  readDtls.updatedBy;
evidenceComparisonDtls.descriptor.updatedDateTime =
  readDtls.updatedDateTime;

ResourceBundle domainTypes =
 ResourceBundle.getBundle(
  SampleSportingGrantConst.kSampleSportingActivityDomainsFile,
    new Locale(TransactionInfo.getProgramLocale()));

ResourceBundle labels =
 ResourceBundle.getBundle(
  SampleSportingGrantConst.kSampleSportingActivityLabelsFile,
    new Locale(TransactionInfo.getProgramLocale()));

Object[] valueObjects = {
        readDtls.clientDtls.name
      , readDtls.sportingActivityType
      , readDtls.sportingAwardType
      , readDtls.paymentAmount
      , readDtls.startDate
      , readDtls.endDate
      , readDtls.comments
      };

EvidenceComparisonHelper helper =
  new EvidenceComparisonHelper();

// populate the return struct one attribute at a time
for (int i = 0;
  i < SampleSportingGrantConst.kSampleSportingActivityNames
      .length
  && i < valueObjects.length; i++) {

  EvidenceAttributeDtls attribute =
    new EvidenceAttributeDtls();

  try {
   attribute.domain =
    domainTypes.getString(
     SampleSportingGrantConst.kSampleSportingActivityNames[i]);
  } catch (MissingResourceException mrException) {
    // missing domain causes widget to fail
    // insert SVR_STRING by default
    attribute.domain = CuramConst.kDomainSVR_STRING;
  }

  try {
    attribute.label =
     labels.getString(
```

```
    SampleSportingGrantConst.kSampleSportingActivityNames[i]);
  } catch (MissingResourceException mrException) {
    // labels are blank by default
    attribute.label = CuramConst.gkEmpty;
  }
  attribute.value =
    helper.objectToString(valueObjects[i]);
  attribute.name =
    SampleSportingGrantConst.kSampleSportingActivityNames[i];
  evidenceComparisonDtls.details.addRef(attribute);
}

  return evidenceComparisonDtls;
}
```

Implementing Transfer Evidence

Introduction

The purpose of this chapter is to provide instructions on how to implement the transferEvidence Evidence Interface function. This is only necessary when dealing with handcrafted evidence as this function is automatically generated when using the Cúram Evidence Generator.

Identify Evidence Types Requiring Specialized Transfer Evidence Code

Some evidence entities contain one or more case participant role fields. These are foreign keys to the Case Participant Role entity. When this evidence is broadcast to one or more target cases, the evidence will initially be inserted with the case participant roles of the source case. These must be handled by specialized code in the transferEvidence Evidence Interface function so these fields are updated with case participant roles on the target case. An example of such code is shown below:

```
// _____
/*
 * Method that does any entity adjustments for moving the
 * evidence record to a new caseID
 *
 * @param details Contains the evidenceID / evidenceType
 *   pairings of the evidence to be transferred
 * @param fromCaseKey The case from which the evidence is being
 *   transferred
 * @param toCaseKey The case to which the evidence is being
 *   transferred
 */
public void transferEvidence(EvidenceTransferDetails details,
  CaseHeaderKey fromCaseKey, CaseHeaderKey toCaseKey)
  throws AppException, InformationalException {

  EIEvidenceKey key = new EIEvidenceKey();

  CaseParticipantRoleKey caseParticipantRoleKey =
    new CaseParticipantRoleKey();
  CaseParticipantRoleDtls caseParticipantRoleDtls;
  CaseIDParticipantRoleKey caseIDParticipantRoleKey =
    new CaseIDParticipantRoleKey();

  CaseParticipantRoleDtlsList caseParicipantRoleDtlsList;
  CaseParticipantRole caseParticipantRoleObj =
    CaseParticipantRoleFactory.newInstance();

  // Read the "from" Evidence entity details
  key.evidenceID = details.fromEvidenceID;
  key.evidenceType = details.fromEvidenceType;
```

```
fromClaimParticipantDtls =
  (ClaimParticipantDtls)readEvidence(key);

// Read the "to" evidence entity details
key.evidenceID = details.toEvidenceID;
key.evidenceType = details.toEvidenceType;
toClaimParticipantDtls =
  (ClaimParticipantDtls)readEvidence(key);

// Get the case participant details
curam.core.sl.intf.CaseParticipantRole
  caseParticipantServiceLayerObj =
  curam.core.sl.fact.CaseParticipantRoleFactory.newInstance();

CaseParticipantRoleDetails caseParticipantRoleDetails =
  new CaseParticipantRoleDetails();

caseParticipantRoleDetails.dtls.caseID = toCaseKey.caseID;
caseIDParticipantRoleKey.caseID = toCaseKey.caseID;
caseParticipantRoleDetails.dtls.fromDate =
  Date.getCurrentDate();
caseParticipantRoleDetails.dtls.recordStatus =
  RECORDSTATUS.NORMAL;

if (fromClaimParticipantDtls.caseParticipantRoleID != 0L) {

  // Find the ParticipantRoleID by using the existing
  // CaseParticipantRoleID
  caseParticipantRoleKey.caseParticipantRoleID =
    fromClaimParticipantDtls.caseParticipantRoleID;

  caseParticipantRoleDtls =
    caseParticipantRoleObj.read(caseParticipantRoleKey);

  // Need to search for the CaseParticipantRole that have the
  // to CaseID and the existing ParicipantRoleID. There should
  // only be one.
  caseIDParticipantRoleKey.participantRoleID =
    caseParticipantRoleDtls.participantRoleID;

  caseParticipantRoleDtlsList =
    caseParticipantRoleObj.searchByParticipantRoleAndCase(
      caseIDParticipantRoleKey);

  caseParticipantRoleDetails.dtls.participantRoleID =
    caseParticipantRoleDtls.participantRoleID;

  // If the list is empty, it means the participant to whom the
  // evidence belongs is not a CPR on the toCase
  if (caseParticipantRoleDtlsList.dtls.isEmpty()) {

    // never create a PRIMARY in transferEvidence
    if (caseParticipantRoleDtls.typeCode.equals(
      CASEPARTICIPANTROLETYPE.PRIMARY)) {

      caseParticipantRoleDetails.dtls.typeCode =
        CASEPARTICIPANTROLETYPE.MEMBER;
    } else {
      // use the 'from' type
      caseParticipantRoleDetails.dtls.typeCode =
        caseParticipantRoleDtls.typeCode;
    }

    // Create a new record
    caseParticipantServiceLayerObj.insertCaseParticipantRole(
```

```
        caseParticipantRoleDetails);

      toClaimParticipantDtls.caseParticipantRoleID =
        caseParticipantRoleDetails.dtls.caseParticipantRoleID;

    } else {

      // MEMBER takes precedence
      if (fromClaimParticipantDtls.caseParticipantRoleID
        == toClaimParticipantDtls.caseParticipantRoleID) {

        for (int i = 0;
          i < caseParticipantRoleDtlsList.dtls.size(); i++) {

          if (caseParticipantRoleDtlsList.dtls.item(
            i).typeCode.equals(CASEPARTICIPANTROLETYPE.MEMBER)
            || caseParticipantRoleDtlsList.dtls.item(
            i).typeCode.equals(CASEPARTICIPANTROLETYPE.PRIMARY)) {

            toClaimParticipantDtls.caseParticipantRoleID =
              caseParticipantRoleDtlsList.dtls.item(
                i).caseParticipantRoleID;
            break;
          }

        }

      }

      // If there are still no matches, use the MEMBER type to
      // create a new record
      if (fromClaimParticipantDtls.caseParticipantRoleID
        == toClaimParticipantDtls.caseParticipantRoleID) {

        caseParticipantRoleDetails.dtls.typeCode =
          CASEPARTICIPANTROLETYPE.MEMBER;

        caseParticipantServiceLayerObj.insertCaseParticipantRole(
          caseParticipantRoleDetails);

        toClaimParticipantDtls.caseParticipantRoleID =
          caseParticipantRoleDetails.dtls.caseParticipantRoleID;
      }

    }

  }

  claimparticipantKey.evidenceID = details.toEvidenceID;
  modify(claimparticipantKey, toClaimParticipantDtls);
}
```

Using the Broadcast Evidence Hook

Introduction

The purpose of this chapter is to provide instructions on how to use the broadcast evidence hook.

Identify Evidence Types with Pre or Post Processing

Some evidence types require pre and / or post processing when evidence is created, whether through an insert or through sharing. The purpose of the broadcast evidence hook is to allow developers to include

this processing when sharing evidence. Before using the broadcast evidence hook, developers must first identify the evidence types with pre and / or post create processing which need to be invoked as part of evidence sharing.

Developers then need to provide a second create business process whose signature will accept the additional parameters required for evidence sharing. Keeping the existing create business process will ensure there is no impact on existing functionality and existing tests. The simplest way to achieve this is to move the code from the original business process into the new business process and get the original process to call the new one. Here is a sample of the signature for the new business process:

```
//_____
/**
 * Creates a <custom> evidence record.
 *
 * @param dtls Contains <custom> evidence creation details
 * @param sourceEvidenceDescriptorDtls If this function is
 *   called during evidence sharing, this parameter will be
 *   non-null and it represents the header of the evidence
 *   record being shared (i.e. the source evidence record)
 * @param targetCase If this function is called during evidence
 *   sharing, this parameter will be non-null and it represents
 *   the case the evidence is being shared with.
 * @param sharingInd A flag to determine if the function is
 *   called in evidence sharing mode. If false, the function
 *   is being called as  part of a regular create.
 *
 * @return the new evidence ID and warnings.
 */
public ReturnEvidenceDetails create<Custom>Evidence(
  <Custom>EvidenceDetails dtls,
  EvidenceDescriptorDtls sourceEvidenceDescriptorDtls,
  CaseHeaderDtls targetCase, boolean sharingInd)
  throws AppException,InformationalException {
```

Provide Implementation for the Evidence Broadcast Hook

The Broadcast Evidence hook is used to route the processing for specific evidence types to their respective create business processes. Here is a sample implementation of the Broadcast Evidence hook which includes comments to clearly describe what needs to be done:

```
/**
 * Sample implementation of the Broadcast Evidence hook.
 */
public abstract class CustomBroadcastEvidence extends
  custom.evidencebroker.sl.base.CustomBroadcastEvidence {

  //_____
  /**
   * Delegates the evidence broadcast through the custom service
   * layer processing.
   *
   * @param sourceDescriptor The source evidence descriptor
   * @param targetCase The case the evidence is being broadcast
   *   to
   * @return The evidence descriptor of the broadcast record on
   *   the target case
   */
  public EvidenceDescriptorDtls processBroadcast(
    EvidenceDescriptorDtls sourceDescriptor, CaseHeaderDtls
    targetCase) throws AppException, InformationalException {

    if (sourceDescriptor.evidenceType.equals(
      CASEEVIDENCE.ALIEN)) {

      // Read the Alien evidence details (through the service
```

```
    // layer)
    AlienKey alienKey = new AlienKey();
    alienKey.alienID = sourceDescriptor.relatedID;

    ReturnAlienDetails alienDetails =
      AlienFactory.newInstance().readAlienDetails(alienKey);

    // Assign these details to the alien creation struct,
    // e.g.
    // Note: a number of assignments may be required here
    // depending on the number of aggregated structs
    // within ReturnAlienDetails and CreateAlienDetails
    CreateAlienDetails createAlienDetails =
      new CreateAlienDetails();
    createAlienDetails.assign(alienDetails);

    ReturnCreateAlien returnCreateAlien =
      AlienFactory.newInstance().createAlienEvidence(
        createAlienDetails,
        sourceDescriptor,
        targetCase,
        true);

    RelatedIDAndEvidenceTypeKey key =
      new RelatedIDAndEvidenceTypeKey();

    key.relatedID = returnCreateAlien.alienID;
    key.evidenceType = CASEEVIDENCE.ALIEN;

    // Read the EvidenceDescriptor and return the details
    EvidenceDescriptor evidenceDescriptorObj =
      EvidenceDescriptorFactory.newInstance();

    return evidenceDescriptorObj.readByRelatedIDAndType(key);
  }

  // null will be returned for all other evidence types
  return null;
  }

}
/**
  * Delegates the external evidence broadcast through the
  * custom service layer processing.
  *
  * @param descriptorDetails Contains the evidence descriptor
  *          details received from remote system.
  * @param targetCase Contains the case the evidence is being
  *          broadcast to.
  *
  * @return The evidence descriptor of the broadcast record on
  * the target case.
  */

  public EvidenceDescriptorDtls processExternalBroadcast(
      SharedEvidenceDescriptorDetails descriptorDetails,
      CaseHeaderDtls targetCase) throws AppException,
      InformationalException {

    if (descriptorDetails.details.evidenceType.
        equals(CASEEVIDENCE.ALIEN)) {
      EvidenceDescriptorDtls evidenceDescriptorDtls =
      EvidenceControllerFactory.newInstance().
      shareExternalEvidence(descriptorDetails, targetCase);
```

```
      // Perform Alien evidence specific processing here
      // . . .
      // . . .

      return evidenceDescriptorDtls;
    }
    // null will be returned for all other evidence types
    return null;
  }
/**
  * Returns the structure with a true value set if the evidence being
  * passed has been auto accepted onto the target case else false would
  * be returned.
  *
  * @param sourceDescriptor
  *           Contains source evidence descriptor details.
  * @param targetCase
  *           Contains the case identifier of the evidence is being
  *           broadcast to.
  *
  * @return True would be returned if the evidence being passed has
  *           been auto accepted onto the target case else false.
  */
  public EvidenceAutoAcceptanceInd isAutoAccepted(
      EvidenceDescriptorDtls sourceDescriptor,
      CaseHeaderDtls targetCase) throws AppException,
      InformationalException {
    return null;
  }
```

Register Custom Override of OOTB Broadcast Evidence Hook

The Cúram Evidence Broker comes with an OOTB Broadcast Evidence hook as part of the Evidence Broker. After creating a custom version of the hook, it is necessary to associate the custom version with the evidence type. The custom hook is managed by a combination of the BroadcastEvidenceManager.

This can be done in two ways: using Google Guice module or by a registrar. Though both approaches achieve the same goal, the Google Guice route is preferred over the registrar route. Typically the registrar route should be used when overriding the OOTB implementation. Both approaches are outlined below:

Using Google Guice

This can be done by creating a Guice module class and adding a corresponding entry in the MODULE table. A Guice module class is created by deriving a class from com.google.guice.AbstractModule and overriding the configure method to add the following statement:

```
public void configure() {

    MapBinder<String, Method> broadcastEvidenceHookMapBinder
    = MapBinder.newMapBinder(binder(), String.class, Method.class,
    new RegistrarImpl(RegistrarType.EVIDENCE_BROKER));

    broadcastEvidenceHookMapBinder.addBinding(CASETYPECODE.
    INTEGRATEDCASE).toInstance(BroadcastEvidenceFactory.class.
    getMethod(ReflectionConst.kNewInstance, new Class[0]));
    }
```

Where 'RegistrarType.EVIDENCE_BROKER' is an annotation which is used to differentiate between various registrar maps.

It is not necessary to create a new module for each of such hooks you have to bind. Single module class per component will work well.

Using Registration

This is done by adding the custom hook to the ENV_BROADCASTEVIDENCE_REGISTRARS_LIST environment variable. The BroadcastEvidenceRegistrar contains an interface which must be implemented by a custom registrar class in order to register the class which implements the hook. This is looked up via the BroadcastEvidenceManager class inside the Evidence Broker.

```
/**
  * Registers the custom evidence broadcast hook
  */
public class CustomBroadcastEvidenceRegistrar
   extends custom.sl.base.CustomBroadcastEvidenceRegistrar
   implements BroadcastEvidenceRegistrar {

   public void register() {
     HookMap map = BroadcastEvidenceManager.get();
     map.addMapping(CASETYPECODE.INTEGRATEDCASE,
       BroadcastEvidenceFactory.class);
   }
}
```

Evidence Broker Sharing Strategy

Introduction

The purpose of this chapter is to provide high level instructions on how to implement the EvidenceBrokerSharingStrategy and EvidenceSharingStrategy interface and usage of hook which skips evidence sharing on case/case member creation.

Provide Implementation for the EvidenceBrokerSharingStrategy Interface

This interface needs to be implemented in scenarios where the customer wants to change the conditions under which an evidence type for a specific case is shared or the target systems it is shared with.

Please see the code of class curam.evidencebroker.sl.infrastructure.impl.EvidenceBrokerSharingStrategyImpl as an interface implementation sample.

Map the custom strategy to a case type
Cúram Evidence Broker provides an EvidenceBrokerSharingStrategy implementation that acts as the default strategy for all case types without a specific sharing strategy.

After you create a custom strategy (for example, CustomEvidenceBrokerSharingStrategyImpl) for a case type, you must bind the strategy with the case type. Create a Guice module class and add a corresponding entry in the MODULE table. You can create a Guice module class by deriving a class from com.google.guice.AbstractModule and overriding the configure method to add the following statement:

```
public void configure() {

    MapBinder<String, CaseTypeEvidence< mapbinder =
          MapBinder.newMapBinder(binder(), String.class,
             CaseTypeEvidence.class);
      mapbinder.addBinding(CASETYPECODE.INTEGRATEDCASE).to(
          IntegratedCaseTypeEvidence.class);

   }
```

You don't need to create a new module for each hook that you must bind. You can use a single module class per component.

Using a hook to skip evidence sharing on case or case member creation

You can use a hook to stop and restart evidence sharing on the case or case member creation process.

Use the stopSharing() and restartSharing() APIs in curam.evidencebroker.sl.intf.EvidenceBroker.java to stop and restart the evidence sharing process. Evidence Broker triggers evidence sharing based on the 'StopEvidenceSharing' flag. The stopSharing() API sets the 'StopEvidenceSharing' flag, which skips evidence sharing in EvidenceBroker. You can call the API at the start of the case or case member creation process. You can restart evidence sharing by calling the restartSharing() API, which resets the 'StopEvidenceSharing' flag and allows evidence sharing to proceed.

Provide Implementation for the EvidenceSharingStrategy Interface

A new hook is provided which would be called before evidence sharing in order to filter/finalize on the evidences that should be shared on the target case. With this hook the customer will be able to determine evidences they wish to share between the source and the target cases. EvidenceSharingStrategy interface must be implemented when the customer wants to filter/finalize the evidences for a specific case type that must be shared between the source and the target case. On customizing the strategy (for example, CustomEvidenceSharingStrategyImpl) for a specific case type, it is necessary to bind the strategy with the case type. This can be done by creating a Guice module class and adding a corresponding entry in the MODULE table.

Using Evidence Activation Hook

Introduction

The purpose of this chapter is to provide instructions on how to use the evidence activation hook.

Provide Implementation for the Evidence Activation Hook

A new hook is provided to determine the automation strategy of activating the evidences. EvidenceActivation interface must be implemented if the customer wishes to determine how the evidence was activated.On customizing the strategy (for example, CustomEvidenceActivationImpl) for a specific case type, it is necessary to bind the strategy with the case type. This can be done by creating a Guice module class or adding a corresponding entry in the MODULE table..

Implementing Evidence Broker Task Hook Interfaces

Introduction

The purpose of this chapter is to provide high level instructions on how to implement the EvidenceBroker Task Hook interfaces.

Implementing the EvidenceBrokerTaskGenerationStrategy Interface

A hook has been provided to determine the strategy of generating tasks on brokering of evidences. The interface EvidenceBrokerTaskGenerationStrategy must be implemented, if the customer wishes to change the conditions under which tasks are generated.

Customers who wish to customize the task generation has to implement EvidenceBrokerTaskGenerationStrategy interface and bind the new implementation in their corresponding Module class.

For example, in their own Module class, *bind(EvidenceBrokerTaskGenerationStrategy.class).toInstance(new MockEvidenceBrokerTaskGenerationStrategy());*

If a custom implementation class is bounded, the same will be called instead of the default implementation. To load their own Module class, customers should add a row to MODULECLASSNAME table in the database using a corresponding custom DMX file of their own.

Implementing the EvidenceBrokerTaskStrategyHook Interface

A hook has been provided to determine the strategy of generating tasks when brokering of evidences fails. The interface EvidenceBrokerTaskStrategyHook must be implemented, if the customer wishes to customize the task which is generated when brokering of evidence fails.

Customers who wish to customize the task which is generated due to failure of evidence brokering has to implement EvidenceBrokerTaskStrategyHook interface and bind the new implementation in their corresponding Module class.

For example, in their own Module class, *bind(EvidenceBrokerTaskStrategyHook.class).toInstance(new MockEvidenceBrokerTaskStrategyHook());*

If a custom implementation class is bounded, the same will be called instead of the default implementation. To load their own Module class, customers should add a row to MODULECLASSNAME table in the database using a corresponding custom DMX file of their own.

Evidence Broker Web Service

Introduction

This chapter provides a high level overview of the evidence broker web service that facilitates evidence sharing with remote systems.

Receive Change Notification Service

Cúram Evidence Broker uses web services to accept change notifications from remote systems. The web service calls are implemented on an Axis2 stack for improved performance, security, and flexibility.

When a remote system calls the Receive Change Notification service (EvidenceBrokerWS.receiveChangeNotification), the service layer class verifies that the structure of the incoming XML is correct and then creates an ExternalCaseHeader record with some basic information as a representation of the source case that exists in the remote system. The incoming XML is translated into structures and the sharing process starts based on the configured sharing strategy.

Incoming Parameters

The parameters are used to populate the internal struct: curam.core.sl.struct.SharedEvidenceDescriptorDetails:

Table 1: Minimum Requirements		
Intake Element	**Map to Parameter**	**Schema Type**
caseID	caseID	se:caseReference
participantNumber	participantID	se:personReference
evidenceType	evidenceType	se:evidenceType
caseType	sourceType	se:caseType
caseSubType	sourceID	se:caseSubType
sourceSystemName	sourceSystemID	se:sourceSystemName
sharedInstanceID	sharedInstanceID	se:sharedInstanceID
operation	operation	se:OperationName

Table 1: Minimum Requirements (continued)		
Intake Element	**Map to Parameter**	**Schema Type**
receivedDate	effectiveFrom	se:date
effectiveDate	effectiveDate	se:date
dataObjects	see below	see below

The parameters caseID, participantID, sourceSystemID are internal ID determined by querying the database using the attributes CaseHeader.caseReference, ConcernRole.primaryAlternateID, TargetSystem.systemName.

The parameter sourceID is determined using the API curam.core.sl.impl.CaseTypeEvidence.getSubTypeID(final String caseSubType) using the caseSubType value.

Each Incoming Evidence schema has an object structure defined for the incoming data. The dataObjects structure is:

```
<dataItem name="{data item name}"
        >{value}</dataItem>
```

- Data Item name: The name of the attribute within the struct that is passed to the entity object.
- Value: The value to populate the struct field with. This will be passed to the entity object.

Note: DataItem to struct mapping controls all data type conversions and checks.

Incoming Parameter Descriptions

Table 2: Parameter Descriptions		
Parameter	**Domain**	**Description**
caseID	CASE_ID	This is the case identifier to identify the case with which this evidence is associated.
sourceType	CASE_TYPE_CODE	The source case type code from which evidence being shared. Code table:CaseTypeCode
effectiveDate	CURAM_DATE	The date from which this Evidence applies. Format:ddMMyyyy
sharedInstanceID	INTERNAL_ID	Unique identifier that will be common to all evidence records which have been shared from the same initial piece of evidence.
evidenceType	EVIDENCE_TYPE_CODE	This is the evidence type code to identify the type of Evidence record. Code table:EvidenceType
operation	OPERATION_NAME	This corresponds to evidence create or remove operations. Type:string
sourceID	INTERNAL_ID	The unique identifier of the source product from which evidence is being shared.

Table 2: Parameter Descriptions (continued)		
Parameter	**Domain**	**Description**
sourceSystemID	INTERNAL_ID	The unique identifier of the source system from which evidence is being shared.
participantID	CONCERN_ROLE_ID	Identifier of the participant to whom the evidence relates; this could be the primary client of the case or a member of the integrated case.

The following figure displays an example of the inbound Share Evidence xml message:

```xml
<evidence xmlns="http://ws.curam/EvidenceShare" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ws.curam/EvidenceShare">
  <evidenceData>
    <evidenceDetails>
      <caseID>512</caseID>
      <participantNumber>30000001</participantNumber>
      <evidenceType>ET500</evidenceType>
      <caseType>CT5</caseType>
      <caseSubType>PC9</caseSubType>
      <sourceSystemName>Bird</sourceSystemName>
      <sharedInstanceID>-3381077420248399872</sharedInstanceID>
      <operation>ON2001</operation>
      <receivedDate>2011-01-20</receivedDate>
      <effectiveDate>0001-01-01</effectiveDate>
    </evidenceDetails>
    <dataObjects>
      <dataItem name="sportingActivityID" type="long">2527645290861690880</dataItem>
      <dataItem name="caseParticipantRoleID" type="long">-2084040727565697024</dataItem>
      <dataItem name="sportingActivityType" type="string">SA8</dataItem>
      <dataItem name="sportingAwardType" type="string">SAT1</dataItem>
      <dataItem name="paymentAmount" type="money">40.00</dataItem>
      <dataItem name="comments" type="string" />
      <dataItem name="startDate" type="date">2008-08-04</dataItem>
      <dataItem name="endDate" type="date">2008-09-28</dataItem>
      <dataItem name="versionNo" type="int">1</dataItem>
    </dataObjects>
  </evidenceData>
</evidence>
```

Figure 1: Inbound Example : Share Evidence

Configuring Non-Dynamic to Dynamic Evidence Attribute Mapping

To support non-dynamic to dynamic evidence mapping at attribute level, an application property "Evidence Broker - Static Dynamic Attribute Mapping" is available. By default, this property is set to "NO". Which means that non-identical evidence brokering for non-dynamic to dynamic evidence will continue to support the existing behavior. This means that only evidence attributes with the same name and data type are mapped to one another between the source and target evidence types. To enable mapping between specific attributes on non-dynamic to dynamic evidence, the property should be enabled (set to "YES") and following set of entities must be updated after the non-identical evidence configuration has been created from within admin.

EBATTRIBUTEMAPPING
This entity maintains the details of the evidence type and the attribute mapping between evidences. Identified attributes to be mapped should be populated in this entity along with the evidence type.

EBCODETABLEMAPPING

This entity maintains the code table name mapping for the mapped evidence attributes which have code table data types. For every code table data type that is mapped, this entity has to be populated with the corresponding code table value name.

EBCODETABLEVALUEMAPPING

This entity maintains the code table code mapping for the mapped evidence attributes which have code table data types. The code table code values that need to be mapped between evidences is populated in this entry.

Non-Dynamic to Dynamic Evidence Attribute Mapping Example

Consider the following attribute and code table mapping configuration example between non-dynamic evidence 'Earning' and dynamic evidence 'Income'

Table 3: Attribute Mapping		
S.No	**Earning**	**DynamicEvidence_Income**
1	amount (Money)	amount (Money)
2	startDate (Date)	fromDate (Date)
3	endDate (Date)	toDate (Date)
4	type (Code Table Code)	incomeType (Code Table Code)

Table 4: Code Table Mapping		
S.No	**type (Sample_Income code table)**	**incomeType (Sample_Income code table)**
1	Net Self Employment Income (SIT2001)	Net Self Employment Income (SIT2001)
2	Annuities and Pensions (SIT2002)	Annuities and Pensions (SIT2002)

Table 5: EBATTRIBUTEMAPPING Entity Records				
EBAttributeMappingID	**LHSEvidenceType**	**LHSAttribute**	**RHSEvidenceType**	**RHSAttribute**
101	Earning	amount	DynamicEvidence_Income	amount
102	Earning	startDate	DynamicEvidence_Income	fromDate
103	Earning	endDate	DynamicEvidence_Income	toDate
104	Earning	type	DynamicEvidence_Income	incomeType

Table 6: EBCODETABLEMAPPING			
EBCodeTableMapping	**EBAttributeMappingID**	**LHSCodeTableName**	**RHSCodeTableName**
201	104	Sample_Income	Sample_Income

Table 7: EBCODETABLEMAPPING					
EBCodeTableValueMappingID	**EBCodeTableMappingID**	**LHSCodeTableName**	**LHSCodeTableValue**	**RHSCodeTableName**	**RHSCodeTableValue**
301	201	Sample_Income	SIT2001	Sample_Income	SIT2001

Notices

This information was developed for products and services offered in the United States.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing IBM Corporation North Castle Drive, MD-NC119 Armonk, NY 10504-1785 US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing Legal and Intellectual Property Law IBM Japan Ltd. 19-21, Nihonbashi-Hakozakicho, Chuo-ku Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing IBM Corporation North Castle Drive, MD-NC119 Armonk, NY 10504-1785 US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBMproducts. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Privacy Policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies or other similar technologies that collect each user's name, user name, password, and/or other personally identifiable information for purposes of session management, authentication, enhanced user usability, single sign-on configuration and/or other usage tracking and/or functional purposes. These cookies or other similar technologies cannot be disabled.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at " Copyright and trademark information " at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other names may be trademarks of their respective owners. Other company, product, and service names may be trademarks or service marks of others.

IBM®

Part Number:

(1P) P/N: