Merative Social Program Management 8.1

Working with Intelligent Evidence Gathering (IEG) Guide

Note

Before using this information and the product it supports, read the information in Notices on page 71

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 Working with Intelligent Evidence Gathering..... 9

1.1 Get started..... 9
About IEG..... 9
Evaluating the Use of IEG..... 11
The Basics..... 11
1.2 Capturing Client Information..... 18
Families and Households..... 18
Household Relationships..... 21
Summarizing Client Information..... 22
1.3 Capturing Related Data..... 23
Capturing Composite Data..... 23
Displaying Composite Data on a Summary..... 24
Capturing Associated Data..... 25
Displaying Associated Data on a Summary..... 26
Deleting Associated Data..... 27
1.4 Efficient Ways of Capturing Data..... 28
List Questions..... 28
Code-table questions..... 29
Conditional Elements..... 31
Question Matrices..... 34
Fast Path Navigation..... 35
Implicit Delete..... 39
Three Field Date Picker..... 39
1.5 Other Script Development Considerations..... 40
Displaying Data as Read-Only..... 40
Invoking external functionality by using expressions and custom functions..... 41
Reusing Scripts..... 44
Source Control and Versioning..... 45
Rendering Custom HTML on a Summary Page..... 46
1.6 Integrating IEG into a Cúram Application..... 53
Creating a Script Execution..... 53
Specifying a Redirection URL..... 54
Running the IEG Player in a Tab..... 54
Running the IEG Player in a Modal Dialog..... 56
Cleaning Up Application Data..... 60

Resuming Executed Scripts..... 61
1.7 Managing Captured Data..... 61
Retrieving Captured Data..... 61
Pre-Populating Scripts with Captured Data..... 62
1.8 Using the Resource Store..... 63
Listing all Resources..... 63
Uploading a New Resource..... 64
Removing an Existing Resource..... 64
Updating an Existing Resource..... 65
Downloading an Existing Resource..... 65
Adding Images..... 65
Changing Static Text..... 65
Changing the Default File Encoding..... 66
1.9 Using IBM Rational AppScan to scan IEG..... 66
Preparation..... 66
Relationship Pages..... 66
Scan Configuration..... 67
Test Policy..... 67
Explore Options..... 67
Communications and Proxy..... 67
Test Options..... 68
Multi-Step Operations..... 68
Exclude Paths and Files..... 68
Complete..... 69
Running the Scan..... 69
1.10 Runtime processing in IEG..... 69
Loss of network connectivity during an IEG session..... 69

Notices..... 71

Privacy policy..... 72
Trademarks..... 72

1 Working with Intelligent Evidence Gathering

Use this information to learn how to define and maintain IEG scripts and the associated data store schemas for use in internal or external applications.

IEG can capture data as part of an internal or external application. Typically, the data is client-related data and is required as part of an application for a program or to determine potential eligibility. All such information comes under the general heading of evidence in Merative™ Social Program Management.

This information outlines some considerations when creating an IEG script and Data Store schema and information about how to maintain scripts.

1.1 Get started

Read about the basic principles of IEG and its dependency on the Datastore (DS) and the Resource Store (RS). You are guided through creating a simple IEG script to gather information about a client.

About IEG

IEG is an efficient alternative to traditional information gathering processes. With IEG, information is gathered interactively by displaying a script of questions that a user can provide answers to.

Questions are only displayed if they are consistent with the user's previous answers so that the user is only required to provide answers relevant to his or her needs and situation. This creates a user-friendly environment that can be effectively implemented for a range of processes including client information intake, benefit assessment triage, online eligibility assessment, and so forth.

In contrast to traditional information gathering processes, IEG cuts down on the organization's administrative work by creating the potential for several routes through the same question script. This eliminates the necessity to develop many scripts for gathering information from different types of users.

A further advantage of IEG is the flexibility of its implementation and the range of its potential users. The IEG runtime environment can be set up for access from any UIM page. This means that IEG can be accessed directly from an organization application or remotely by an online user.

The two main components of IEG are the Engine and the Player. IEG scripts are defined in XML and the Engine interprets the script definitions at runtime and evaluates the answers supplied by the user to determine the flow of execution. The Engine determines which pages should be displayed to the user and how many times they should be displayed. The Player presents the pages, questions and other controls to the user. IEG also builds on other elements of the Cúram Application Suite such as the Datastore (DS) and the Resource Store (RS).

Datastore (DS)

The data supplied by a user during script execution is not directly persisted by IEG itself. This task is delegated to the Datastore (DS). The DS is a configurable database.

Just as the questions and question pages that are to be displayed to the user are determined by an IEG script, the data that can be stored in the DS is dynamically determined by an XML schema. The schema describes the structure of the information you want to store and any relationships between the data. Data is stored in the DS in XML format and conforms to the W3C XML Schema Definition Language. More details on the DS and how it works can be found in the *Creating Datastore Schemas* guide.

An IEG script and a DS schema are very closely linked. An IEG script is defined with references to the elements contained in a schema and for that reason a schema must be supplied when editing a script. The same schema is also required when executing a script. Schemas may by reused to edit and execute multiple scripts so the same data structures can be used in different circumstances.

Resource Store (RS)

An IEG script can contain references to images that will be displayed to the user when a script is executed, for example icons representing sections and question pages. The images are stored in the Resource Store (RS).

An IEG script also contains a number of different textual elements, for example page headings, question labels and help text. IEG allows you to enter all the text for your script for the default locale directly into the script definition.

When an IEG script is uploaded into the system via the IEG admin screens, all the text contained within it is automatically extracted into an appropriately named properties files for the script. These properties files are also stored in the RS. The properties files are stored with no locale associated with them (so that they act as the fall-back properties if no properties exist for the locale in which you are running). The RS allows properties files for multiple locales to be uploaded making the localization of scripts a straightforward task. At runtime, the properties files are retrieved for the appropriate locale and presented to the user in the IEG Player.

Script Structure

In its simplest form, an IEG script consists of pages which include questions to be posed to users of IEG. The structure of the IEG script is a logical grouping of these pages so that answers to the questions can be captured effectively.

Sequences of pages can be grouped into logical sections. The purpose of these sections is to give users a higher level view of the kind of information captured by the IEG script.

In addition to including a variable number of pages, each section should contain one summary page. This page is used to give feedback to the user on the information entered on the pages in a section. Summary pages typically contain clusters and lists displaying read-only versions of the answers to questions asked. The summary page will always be the last page displayed within a section and will also be displayed whenever a user clicks on the link for that section in the sidebar of the IEG Player.

To summarize, IEG scripts consist of a hierarchy of elements structured something like this:

- Script
    - Section

- Page
  - Cluster
    - Question
- Summary Page

Evaluating the Use of IEG

There are some key questions to ask when evaluating the use of IEG in any application:

- What information is being captured?
- What is the source of that information?
- How is this information to be used?
- How long will this information live in the application?

Many of the current uses of IEG stem from the need to support an application for products and services offered by agencies either externally or internally. The information captured is generally client related information, such as client personal details, their family or household details and details of their needs.

Often agencies already have data about a client; therefore they can source the information from another system using some key pieces of information like a social security number. This allows them to verify the client information being entered or retrieve to assist with the application.

Some applications are complex and require information from many sources. Clients may have to enter information that is not close to hand. For example, the required information may be held by their employer. They may need the ability to store what they have entered and return to the application at a later time once they have all the required data.

Clients may be exposed to simple screening applications that inform them of their entitlements under current or new legislation. This information is often unreliable and temporary data must be removed from the system after the client logs out or within a set period of time.

These requirements drive the use of IEG and provide important information on the use of the data over its lifetime.

So, let's start with the basics: we want to capture and store information about a client.

The Basics

Create a Schema

The first step in capturing data about a client is to create a DS schema. This section provides an example of how to create a basic schema that defines the capture of some general client data.

The DS stores data collected from users during online screening and intake of applications. The contents of the DS are dynamically definable by way of a schema definition. The requirements for capturing and storing any data about a client can be complex but with appropriate schema design, this data can be efficiently managed over its lifetime.

For the purposes of this example, the requirement is to capture the following:

Table 1: Client Data to Capture

Attributes	Type
First name	String
Middle name	String
Last name/Family name	String
Gender	Male/Female
Date of Birth	Date

There is a minimum set of definitions required in a schema. For a schema to be used in IEG, the following is required:

- Inclusion of Base Domains
- Inclusion of IEG Domains
- A root entity

For more information on the minimum set of definitions required, see the *Creating Datastore Schemas* guide.

The schema would look something like this before adding new content such as the Person entity described above:

```
<xsd:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:d="http://www.curamsoftware.com/BaseDomains">
  <xsd:import namespace="http://www.curamsoftware.com/
BaseDomains"/>
  <xsd:include schemaLocation="IEGDomains"/>
  <xsd:element name="Application">
    <xsd:complexType>
      <xsd:sequence minOccurs="0">
    </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figure 1: Starting Schema

The content of the schema indicates that it is an XMLSchema that imports BaseDomains schema and includes the IEGDomains schema. The first element called `Application` is the root entity for the schema. IEG requires that the root entity is always called `Application`.

The IEGDomains schema contains the domains required to define the attributes of entities to be used with IEG. The types of the attributes must be derived from the IEG Domains rather than the base domains. A Person entity can be defined to represent a client as follows:

```
<xsd:element name="Person">
  <xsd:complexType>
    <xsd:attribute name="firstName" type="IEG_STRING"/>
    <xsd:attribute name="middleName" type="IEG_STRING"/>
    <xsd:attribute name="lastName" type="IEG_STRING"/>
    <xsd:attribute name="gender" type="IEG_GENDER"/>
    <xsd:attribute name="dateOfBirth" type="IEG_DATE"/>
  </xsd:complexType>
</xsd:element>
```

Figure 2: Person Entity

There are a couple of things to note about the above addition for an entity like person:

- Like relational database tables, an ID field is required and a key is defined for this table using this unique ID.
- The person entity is added as a child entity of the root entity.

The schema to capture basic information about a person can be defined as follows:

```
<xsd:element name="Application">
  <xsd:complexType>
    <xsd:sequence minOccurs="0">
       <xsd:element ref="Person" minOccurs="0"
                                 maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="Person">
  <xsd:complexType>
    <xsd:attribute name="personID" type="d:SVR_KEY"/>
    <xsd:attribute name="firstName" type="IEG_STRING"/>
    <xsd:attribute name="middleName" type="IEG_STRING"/>
    <xsd:attribute name="lastName" type="IEG_STRING"/>
    <xsd:attribute name="gender" type="IEG_GENDER"/>
    <xsd:attribute name="dateOfBirth" type="IEG_DATE"/>
  </xsd:complexType>
  <xsd:key name="Person_Key">
    <xsd:selector xpath="./Person"/>
    <xsd:field xpath="@personID"/>
  </xsd:key>
</xsd:element>
```

Figure 3: Basic Schema

Once the schema has been defined you can then create a script to use the schema.

Create a Script

IEG allows you to create dynamic scripts for collecting data. IEG scripts can contain sections, question pages, questions and conditional logic which allows you to decide what information to capture, what pages to display and how many times they are displayed.

Please read the *`Authoring Scripts using Intelligent Evidence Gathering (IEG)`* guide for details on how to define each element of an IEG script.

For the requirements above, where there is a need to capture information about a person, you must define the script and decide how the pages are arranged to capture the information.

A new script can be created in the admin application and the editor can be used to add elements to this script. The content of a newly created script will be similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<ieg-script xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="ieg-schema.xsd">
  <identifier id="WorkingWithIEG" scriptversionnumber="V1"
              type="Intake" />
</ieg-script>
```

Figure 4: New Script

The `ID`, `Type` and `Version` supplied when creating the script are combined to create a script identifier to uniquely identify the script definition.

Once a new script is created, elements such as sections, question pages and summary pages can be added to the script. The examples in the next two sections will show you how to add a section and a question page to a script as well as how to add a summary page that displays information back to the user. Summary pages allow the user to confirm that the data they entered is correct before proceeding and they can also provide the user with the ability to modify the data.

Adding a Section and a Question Page to an IEG Script

A section and a question page need to be added. A section can be used to group related pages together to allow the user to flow through the screens in a logical manner.

Sections can also help to convey to the user their progress through a script. Both the section and the question page can have a title and the question page can optionally have a description.

The following code sample shows a section containing a question page, added to a script:

```
<?xml version="1.0" encoding="UTF-8"?>
<ieg-script xmlns:xsi="http://www.w3.org/2001/XMLSchema-in
 stance"
 xsi:noNamespaceSchemaLocation="ieg-schema.xsd">
  <identifier id="WorkingWithIEG" scriptversionnumber="V1"
              type="Intake" />
  <section>
    <title id="AboutYouSection.Title">
      <![CDATA[About You]]>
    </title>
    <question-page id="AboutYouPage" entity="Person">
      <title id="PrimaryPersonPage.Title">
        <![CDATA[About You]]>
      </title>
      <description id="PrimaryPersonPage.Description">
        <![CDATA[Please enter some information about yourself]]>
      </description>
    </question-page>
  </section>
</ieg-script>
```

Figure 5: New Section

The question page requires the appropriate questions to capture the data. Any data to be stored in the DS has to be associated with an attribute of an entity in the DS schema to be used with this script. If all the questions on a page relate to the same entity, the page can be mapped to that entity type. In the above example the page is mapped to the Person entity.

To add questions to a page, a cluster is required. Clusters help control the layout of the questions on the page. A page can contain many clusters to allow you to logically group questions on the page. Clusters may also contain a title and a description.

In our example below, there are two clusters, one just to display some informational text to the user and another to contain the questions for personal details. Questions and display text can be added to each cluster. Questions must be given an ID which must correspond to one of the attributes of the entity type the page is mapped to. If an answer must be supplied to a question the mandatory indicator of the question can be set to `true`. The script snippet below contains the questions to capture the required data outlined in our example.

```
<question-page ...
```

```
    <cluster>
    <display-text id="RequiredFields.Text">
      <![CDATA[<span style="color: orange;">
                * indicates a required field</span>]]>
    </display-text>
  </cluster>
  <cluster>
    <title id="DetailsCluster.Title">
      <![CDATA[Personal Details]]>
    </title>
    <description id="DetailsCluster.Description">
      <![CDATA[Enter your details here]]>
    </description>
    <question id="firstName" mandatory="true">
      <label id="FirstName.Label">
        <![CDATA[First Name:]]>
      </label>
    </question>
    <question id="middleName">
      <label id="MiddleName.Label">
        <![CDATA[Middle Name:]]>
      </label>
    </question>
    <question id="lastName">
      <label id="lastName.Label">
        <![CDATA[Last Name:]]>
      </label>
    </question>
    <question id="gender" mandatory="true">
      <label id="Gender.Label">
        <![CDATA[Gender:]]>
      </label>
    </question>
    <question id="dateOfBirth" mandatory="true">
      <label id="DateOfBirth.Label">
        <![CDATA[Date Of Birth:]]>
      </label>
    </question>
  </cluster>
</question-page>
```

Figure 6: Clusters, Questions and Display Text

Please note there are more properties of scripts, sections, question pages, clusters, questions and display texts than are covered here. These properties are covered in the *`Authoring Scripts using Intelligent Evidence Gathering (IEG)`* guide some of which will be discussed later in this guide.

Adding a Summary Page to an IEG Script

The final step of this basic example is to display a summary of the information captured. Generally each section will have a summary page.

A summary page is used to display the most important data back to the user in order for them to verify data was captured or calculated correctly. A summary page can display data captured on multiple question pages. A summary page does not have to contain all the information captured in the section as this could be very large making it less useful.

Obviously if the data displayed on a summary page is incorrect the user will more than likely want to modify it. Users may navigate backwards in the script execution by pressing the Back button in the IEG Player until they reach the page where the data was entered, update the data, then proceed forward through the script again. Alternatively you can add edit links to the clusters on the summary page. When the user clicks on an edit link on a summary page the question page specified in the edit link is displayed to the user in the IEG Player. The user can then change the data and depending on whether the changed data is referenced elsewhere in the script, the summary page will be displayed again when the user presses the Next button in the IEG Player.

The summary page in this case will be very simple and similar to the question page previously added. And similar to a question page, if all the attributes referred to on the page relate to the same entity the summary page can be mapped to that entity type, as follows:

```
<section>
...
  <summary-page id="AboutYouSummary" entity="Person">
    <title id="AboutYouSummary.Title">
      <![CDATA[Information about you]]>
    </title>
    <description id="AboutYouSummary.Description">
      <![CDATA
        [Here's the information you've entered about yourself]]>
    </description>
    <cluster>
      <title id="DetailsCluster.Title">
        <![CDATA[Person Details]]>
      </title>
      <description id="DetailsCluster.Description">
        <![CDATA[Enter the details for this person here]]>
      </description>
      <edit-link start-page="AboutYouPage" />
      <question id="firstName">
        <label id="FirstName.Label">
          <![CDATA[First Name:]]>
        </label>
      </question>
      <question id="middleName">
        <label id="MiddleName.Label">
          <![CDATA[Middle Name:]]>
        </label>
      </question>
      <question id="lastName">
        <label id="lastName.Label">
          <![CDATA[Last Name:]]>
        </label>
      </question>
      <question id="gender">
        <label id="Gender.Label">
          <![CDATA[Gender:]]>
        </label>
      </question>
      <question id="dateOfBirth">
        <label id="DateOfBirth.Label">
          <![CDATA[Date Of Birth:]]>
        </label>
      </question>
    </cluster>
```

```
  </summary-page>
</section>
```

Figure 7: Summary Page

This basic script and schema to capture information about a person and display a summary page is now complete and can be run.

Run a Script

In order to run an IEG script the script definition and the associated schema definition must be uploaded into the system. There are a number of ways this can be done which will be covered later in this guide.

The most straightforward way to upload the definitions is via the administration screens in the Intelligent Evidence Gathering section of the Administration Workspace.

To gain access to the IEG administration screens, you will need to log in as an admin user. Once logged in, you will see a section in your shortcuts panel called Intelligent Evidence Gathering and when you click on it you will see a menu for 'IEG' which contains a link called 'Scripts'. If you click on this, you will see a page that contains a list of the IEG scripts currently in the system and various links to allow you to perform activities on these scripts.

At the top of the 'Scripts' page is an 'Import' link which lets you upload, or import, a new IEG script definition.

Similarly, if you click on the 'Datastore Schemas' link of the menu for 'IEG' you will see a page that contains a list of the DS schemas currently in the system. At the top of the 'Datastore Schemas' page, there is also an 'Import' link which lets you upload, or import, a new schema definition.

For convenience, IEG provides a type of test harness that allows IEG scripts to be tested without having to integrate them into the Cúram application. The test harness does have some limitations but it allows most scripts to be tested as soon as they are uploaded into the system. IEG scripts may be run either in a tab or in a modal window via the admin screens.

A script can be run using either the 'Run' or 'Run in Modal' options for the script from the 'Scripts' page. As there is no explicit association between an IEG script and a DS schema, when you select the option to run a script you will then be asked to select a schema from a dropdown with which to execute the script. Clicking on the 'Run Script' button will cause the IEG Player to launch and you will be presented with the first page of the script.

Validating a Script

When a script is executed via the admin screens in this way, the script is validated before it is executed.

You may also choose the 'Validate' option for the script from the 'Scripts' page. All scripts should be validated before they are executed. If the script fails validation, a list of validation errors will be displayed. The validation errors must be addressed before the script can be run from the 'Scripts' page.

Fill in some sample data on the first page of the script and select the Next button. Now this same sample data should be displayed on the summary page. The answers are not modifiable but an edit link is provided to jump back to the page where that data was entered.

Please note, pressing the Next button in the IEG Player on the summary page of the script that has been implemented in this example will cause an error to be displayed. This is because not all

the properties of the script have been defined. The required properties will be covered later in this guide.

1.2 Capturing Client Information

The previous section outlined a basic example of how IEG can be used to capture data for a client. Some application forms for benefits and services can be complex and the information required about applicants can be very detailed. We build on the initial example covered in the previous section by considering a household, where we captured some initial data about a primary member and now want to add details for the other household members.

Families and Households

We currently have a straightforward script, relating to one person. Often applications need more information about the client's circumstance, starting with their living situation.

In general, information is requested about the primary person and this is followed by a simple question that will allow the client to skip to another area of the application. For example, after entering personal details, the client is asked 'Do you live alone?'. If the answer is yes then the person can be treated as single individual who is not living within a household of family or other individuals. Most clients want to get through the application process as quickly as possible, therefore questions such as these provide a good way to move to more relevant parts of the application.

If the client is living with other people, then questions about each person may need to be asked. Loops are used to capture information from each person and depending on how the script author wants to present these questions, they have a choice of loop types: for, while and for-each loop.

IEG also features a Person Tab that allows the client to see who these questions relate to while entering the data. This will appear automatically for a Person entity in the Datastore. Each Person will be represented by an icon (based on the gender and age) and a name. The current Person will be highlighted.

Let's take a scenario for handling family/household data as an extension of the requirements in the basic sample. Here the client is asked if how many people are in the household including the client. Some new question pages need to be added to capture this information.

The first question page will ask about the living situation. For this example there is only one question to ask, as follows: How many people are in the family (excluding yourself)?

```
<question-page id="HouseholdPage" progress="10">
    <title id="LoopControlPage.Title">
        <![CDATA[Household Details]]>
    </title>
    <description id="LoopControlPage.Description">
        <![CDATA[Please tell us some information about your
        household]]>
    </description>
    <icon image="sample_title_household" />
    <cluster>
        <title id="DetailsCluster.Title">
            <![CDATA[Details]]>
        </title>
```

```
        <question id="numPeople" control-question="true"
            control-question-type="IEG_INT32"
            mandatory="true">
            <label id="NumPeople.Label">
                <![CDATA[How many other people are in your
                household?]]>
            </label>
        </question>
    </cluster>
</question-page>
```

Figure 8: Obtaining household size

This question is a control question, i.e. a question used to control the size of a loop and not for data collection purposes. Control questions are not stored in the Datastore schema. It will used in the loop expression of the 'for' loop in the next question page.

The family members question page is within a 'for' loop that will iterate over the number of family members.

```
<loop loop-type="for" loop-expression="numPeople"
    entity="Person" criteria="isPrimary==false">
    <question-page id="PersonDetailsPage"
        show-person-tabs="true"
        progress="20">
        <title id="PersonDetailsPage.Title">
            <![CDATA[Household Member Details]]>
        </title>
        <description id="PersonDetailsPage.Description">
            <![CDATA[Please enter the details for the
                next person in your household]]>
        </description>
        <icon image="sample_title_household" />
        <cluster>
            <title id="DetailsCluster.Title">
                <![CDATA[Person Details]]>
            </title>
            <description id="DetailsCluster.Description">
                <![CDATA[Enter the details for this person
                below]]>
            </description>
            <question id="firstName" mandatory="true">
                <label id="FirstName.Label">
                    <![CDATA[First Name:]]>
                </label>
            </question>
            <question id="lastName">
                <label id="lastName.Label">
                    <![CDATA[Last Name:]]>
                </label>
            </question>
            <question id="gender" mandatory="true">
                <label id="Gender.Label">
                    <![CDATA[Gender:]]>
                </label>
            </question>
        </cluster>
    </question-page>
```

```
</loop>
```

Figure 9: Using 'for' loop to collect household members

The above is an example of how the client enters the number of family members. But the question could have been asked a different way, for example: 'Do you live with your family?' In this case a condition element in the script can be used to check the value of that question. This would display the family member page if they do live with their family. On this question page, a control question is asked to determine if they would like to capture another family member's details.

This control question would be used in a 'while' loop around the family member question page, as follows:

```
<question-page id="HouseholdPage" progress="10">
    <title id="LoopControlPage.Title">
        <![CDATA[Household Details]]>
    </title>
    <description id="LoopControlPage.Description">
        <![CDATA[Please tell us some information about your
        household]]>
    </description>
    <icon image="sample_title_household" />
    <cluster>
        <title id="DetailsCluster.Title">
            <![CDATA[Details]]>
        </title>
        <question id="livesWithFamily" control-question="true"
            control-question-type="IEG_BOOLEAN"
            mandatory="true">
            <label id="NumPeople.Label">
                <![CDATA[Do you live with your family?]]>
            </label>
        </question>
    </cluster>
</question-page>
```

Figure 10: Using 'while' loop to collect household members

Using this approach, the control question is a boolean type, as it is used in a condition expression that indicates whether or not the while loop should be entered. The loop, once entered, is iterated over until details of all the household members have been gathered, as follows:

```
<condition expression="livesWithFamily==true">
  <loop loop-type="while" loop-expression="
    anotherMember==true"
    entity="Person">
      <question-page id="PersonDetailsPage"
       show-person-tabs="true"
         progress="20">
           <title id="PersonDetailsPage.Title">
               <![CDATA[Household Member Details]]>
           </title>
           <description id="PersonDetailsPage.Description">
               <![CDATA[Please enter the details for
                 the next person in your household]]>
           </description>
           <icon image="sample_title_household" />
           <cluster>
               <title id="DetailsCluster.Title">
```

```
                    <![CDATA[Person Details]]>
                </title>
                <description id="DetailsCluster.Description">
                    <![CDATA[Enter the details for this
                    person below]]>
                </description>
            <question id="firstName" mandatory="true">
                <label id="FirstName.Label">
                    <![CDATA[First Name:]]>
                </label>
            </question>
            <question id="lastName">
                <label id="lastName.Label">
                    <![CDATA[Last Name:]]>
                </label>
            </question>
            <question id="gender" mandatory="true">
                <label id="Gender.Label">
                    <![CDATA[Gender:]]>
                </label>
            </question>
            </cluster>
            <cluster>
                <question id="anotherMember"
                control-question="true"
                  control-question-type="IEG_BOOLEAN">
                    <label id="AnotherMember.Label">
                        <![CDATA[Is there another
                        household member?]]>
                    </label>
                </question>
            </cluster>
        </question-page>
    </loop>
</condition>
```

Figure 11: Using while loop to collect household members

Household Relationships

When gathering information about a group of people in a household, it might be necessary to ascertain how those people are related to each other.

IEG provides a mechanism for capturing relationships through the use of relationship pages and a specific Datastore schema structure.

A Relationship entity should be defined in the Datastore schema, taking the following form:

```
<xsd:element name="Person">
  <xsd:complexType>
    <xsd:sequence minOccurs="0">
      <xsd:element ref="Relationship" minOccurs="0"
        maxOccurs="unbounded"/>
    </xsd:sequence>
    ...
</xsd:element>
<xsd:element name="Relationship">
  <xsd:complexType>
```

```
    <xsd:attribute name="relationshipType"
    type="IEG_STRING"/>
    <xsd:attribute name="isNonParentPrimaryCaretaker"
        type="IEG_BOOLEAN" default="false"/>
    <xsd:attribute name="personID" type="D:SVR_KEY"/>
  </xsd:complexType>
</xsd:element>
```

Figure 12: Relationship Entity in Datastore Schema

A relationship page for the household can be defined as follows, provided that the Relationship entity is a child of the Person entity:

```
<relationship-page id="RelationshipPage" show-person-tabs="true"
    progress="40">
    <title id="RelationshipPage.Title">
        <![CDATA[Household Relationships]]>
    </title>
    <description id="RelationshipPage.Description">
    <![CDATA[Please enter the relationships for %1s below]]>
      <argument id="Person.firstName" />
    </description>
    <icon image="sample_title_household" />
    <question id="caretakerInd">
        <label id="CaretakerInd.Label">
            <![CDATA[Is this a non-parent caretaker
            relationship?]]>
        </label>
    </question>
</relationship-page>
```

Figure 13: Relationship Page

It is only necessary to define the relationship page once. IEG will then display the page as many times as is necessary to gather Relationships one person at a time. This equates to one less times than the number of people in the household, as the last person's Relationships will have been collected in their entirety through the process.

By default, the Relationship Type field is presented as a dropdown, populated from a codetable (configurable through the `relationship.type.domain.name` property):

The relationship page will display a Person Tab at the top containing the list of household members and the current Person will be highlighted. Then each relationship between the current Person and the other members will be displayed.

The caretaker indicator is the only question that can be added directly to the relationship page. Questions regarding other attributes of a Relationship entity must be added to clusters that have been added to the relationship page.

Summarizing Client Information

Lists are used on summary pages to display information gathered in loops. The structure of the list should reflect the structure of the loop or hierarchy of loops that collected the data.

This means that the entity and criteria on the list should match the entity and criteria on the loop. For example, to record the members of the family described in Families and Households on page 18, a for loop was used:

```
<loop loop-type="for" loop-expression="numPeople"
    entity="Person" criteria="isPrimary==false">
  ...
</loop>
```

Figure 14: For loop to collect household member information

In the section summary page, the information gathered in this loop is displayed in a list. The list, like the loop, has 'Person' as its entity and 'isPrimary==false' as its criteria:

```
<list entity="Person" criteria="isPrimary==false">
  ...
</list>
```

Figure 15: List of people

Relationship information gathered using a relationship page can be displayed on summary pages in relationship summary lists:

```
<relationship-summary-list>
    <title id="RelationshipSummaryList.Title">
        <![CDATA[Person Relationships Summary]]>
    </title>
    <description id="PersonRelationshipSummaryList.Description">
        <![CDATA[Person Relationship Summary Details]]>
    </description>
    <column id="caretakerInd">
        <title id="CaretakerInd.Title">
            <![CDATA[NPCR]]>
        </title>
    </column>
    <edit-link start-page="RelationshipPage" />
</relationship-summary-list>
```

Figure 16: Relationship Summary List

1.3 Capturing Related Data

Once we have captured information about the household members such as their personal details and their relationships, we might want to capture related data. This can be achieved through composition (the use of nested DS entities) or association (the use of related, non-nested DS entities).

Capturing Composite Data

We have seen that it is possible to capture relationships in IEG. The combination of the Relationship entity and the RelationshipPage provide a convenient mechanism to capture the relationships between the people in a household.

The relationship between people in a household is only one form of relationship. IEG supports other types of relationships. IEG and the DS allow entities to be nested creating a parent child relationship. This can be seen in the example where there is a requirement to capture the incomes for the people in a household. The Income entity is defined as any other entity is defined. It is nested in the Person entity by referencing it in a sequence, as the following sample code snippet shows:

```
<xsd:element name="Person">
  <xsd:complexType>
    <xsd:sequence minOccurs="0">
      <xsd:element ref="Income" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
    ...
    <xsd:attribute name="hasIncome" type="IEG_BOOLEAN"
      default="false"/>
  </xsd:complexType>
  ...
</xsd:element>
<xsd:element name="Income">
  <xsd:complexType>
    <xsd:attribute name="type" type="IEG_STRING" />
    <xsd:attribute name="amount" type="IEG_MONEY" />
  </xsd:complexType>
</xsd:element>
```

Figure 17: Parent/Child Schema

Income information can then be gathered for people in a household by looping over every person that has income. The loop criteria will use a "hasIncome" boolean question that will be asked while gathering the details for each person. A page within the loop can be mapped to the Income entity thus creating the nested relationship, as shown below:

```
<loop loop-type="for-each" entity="Person"
        criteria="hasIncome==true">
  <loop loop-type="while" loop-expression="hasMoreIncome"
          entity="Income">
    <question-page id="IncomePage" entity="Income"
    ...
```

Figure 18: Creating Nested Entities

Displaying Composite Data on a Summary

The information gathered for nested entities can be displayed on a summary page using a nested list. Similarly to regular lists, nested lists must match the entities and criteria used in the nested loops that captured the data.

```
<list entity="Person" show-icons="true"
 criteria="hasIncome==true">
  <title id="IncomeList.Title">
      <![CDATA[Income]]>
  </title>
  <description id="IncomeList.Description">
    <![CDATA[Here's the income details you've entered for all
 the
                  people in your household]]>
  </description>
  <column id="firstName">
    <title id="FirstName.Title">
      <![CDATA[First Name]]>
    </title>
  </column>
  <list entity="Income">
    <column id="type">
```

```
      <title id="IncomeType.Title">
        <![CDATA[Income Type]]>
      </title>
    </column>
    <column id="amount">
      <title id="IncomeAmount.Title">
        <![CDATA[Income Amount]]>
      </title>
    </column>
  </list>
</list>
```

Figure 19: Displaying Nested Entities on Summary Pages

The sample code snippet above of an income summary list will be displayed in the IEG Player as a regular list with incomes grouped per Person. It will also contain Edit and Delete links for each income and an Add link with a dropdown listing all the people.

Capturing Associated Data

IEG allows association relationships to be created between entities. This is useful because a restriction applies to nested entities and nested lists that they can only be nested to two levels. The use of associated relationships provides an effective alternative to nesting entities to three levels.

For example, suppose there is a requirement to record employment information for the people in a household. Employment information may be gathered independently of Income information as there may be multiple incomes for a given employment.

Once the Income and Employment information is gathered and the entities have been created, the association between the entities can be made. The association is made by creating a "relationship" entity. The relationship entity is normally "owned" by one of the entities participating in the relationship and is represented as a sequence as with other relationship types.

Defining a relationship entity requires being able to identify the related entity therefore a key must be defined in the related entity. To apply this to the Income/Employment example, the Employment entity type will have a key, an EmploymentRelationship entity type will be defined and the Income entity will own a sequence of EmploymentRelationships, as follows:

```
<xsd:element name="Employment">
  <xsd:complexType>
    <xsd:attribute name="employmentID" type="d:SVR_KEY" />
    <xsd:attribute name="employer" type="IEG_STRING" />
    <xsd:attribute name="employmentType" type="IEG_STRING" />
  </xsd:complexType>
  <xsd:key name="Employment_Key">
    <xsd:selector xpath="./Employment" />
    <xsd:field xpath="@employmentID" />
  </xsd:key>
</xsd:element>
<xsd:element name="Income">
  <xsd:complexType>
    <xsd:sequence minOccurs="0">
      <xsd:element ref="EmploymentRelationship" minOccurs="0"
                maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attribute name="type" type="IEG_STRING" />
    <xsd:attribute name="amount" type="IEG_MONEY" />
```

```
  </xsd:complexType>
</xsd:element>
<xsd:element name="EmploymentRelationship">
  <xsd:complexType>
    <xsd:attribute name="employmentID" type="d:SVR_KEY" />
  </xsd:complexType>
</xsd:element>
```

Figure 20: Associated Entity Schema

The association can then be captured in the script by defining a list-question and specifying a link-entity attribute which refers to the key of the related entity. Continuing our example, on a page mapped to the Income entity a list-question can be defined specifying the key from the EmploymentRelationship used to identify the Employment entity.

List questions are constructs that allow the user to choose from a list of entities. For more details, see List Questions on page 28.

```
<question-page id="IncomePage" entity="Income"  ...
  <cluster>
    <layout>
      <label-width>0</label-width>
    </layout>
    <list-question link-
entity="EmploymentRelationship.employmentID"
        entity="Employment" single-select="true">
      <label id="SelectEmployer.Label">
        <![CDATA[Select Employer]]>
      </label>
      <item-label>
        <label-element attribute-id="employer" />
      </item-label>
    </list-question>
  </cluster>
</question-page>
```

Figure 21: Creating Association Relationships

Displaying Associated Data on a Summary

The association between entities can be displayed on a summary page by adding a column to the list of entities of one type, in order to display details of the related entity. A link-entity attribute needs to be specified on this column to identify the related entity.

The following example shows how, while listing the Incomes for a Person on a summary page, the associated Employer name can be displayed for each Income:

```
<summary-page id="IncomeSummary"
  ...
  <list entity="Person" criteria="hasIncome==true"
                          show-icons="true">
    <title id="IncomeList.Title">Income</title>
    <description id="IncomeList.Description">Here's the income
        details you've entered for all the people in your
        household</description>
    <column id="firstName">
      <title id="FirstName.Title">First Name</title>
    </column>
```

```
    <list entity="Income" show-icons="false">
      <column id="type">
        <title id="IncomeType.Title">Income Type</title>
      </column>
      <column id="amount">
        <title id="IncomeAmount.Title">Income Amount</title>
      </column>
      <column id="employer"
              link-entity="EmploymentRelationship.employmentID"
              entity="Employment">
        <title id="Employer.Title">Employer</title>
      </column>
    </list>
  </list>
</summary-page>
```

Figure 22: Entity Association Summary Page

Deleting Associated Data

When entities form parent-child relationships, if the parent entity is deleted, all its child entities are also deleted. When an entity that participates in a relationship is deleted, by default, the relationships for that entity are deleted but the related entities are not.

For example, suppose the details of all the people in a household have been collected and Person entities created and the relationships between the people in the household have also been captured and Relationship entities created. If the user chooses to remove a person, the relationships that person participates in will also be removed but none of the other people in the household will be removed.

This default behavior also applies to the income/employment example. If the user chooses to remove an income, any EmploymentRelationships for the income will be removed but none of the Employment entities will be removed.

It is possible to change the default behavior when deleting associated entities so that any entities related to the entity being removed will also be removed.

To change the default behavior, an annotation containing a documentation element may be added to the definition of a relationship entity in the DS schema. A documentation element containing the text `"@curam.ieg.cascading.delete=true"` indicates that related entities should be deleted when the relationship is deleted.

```
<xsd:element name="EmploymentRelationship">
  <xsd:annotation>
    <xsd:documentation>@curam.ieg.cascading.delete=true
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:attribute name="employmentID" type="d:SVR_KEY" />
  </xsd:complexType>
</xsd:element>
```

Figure 23: Cascading Deletes Schema

In the Income/Employment example, if `curam.ieg.cascading.delete` is set to true for the EmploymentRelationship when an Income entity is removed any associated Employment

entity will also be removed. Removing the Employment entities in this way does not cause other Income entities to the removed.

1.4 Efficient Ways of Capturing Data

This section highlights some of the features of IEG that allow information to be gathered more effectively and more intuitively.

List Questions

IEG provides an alternative to asking the same boolean question for a number of entities. A list question can be used to gather all the answers at the same time.

In an earlier example, we saw a requirement to gather income information for the people in a household. In order to only gather income information for the people who actually have income, a question was added to the 'Household Members Details' page to indicate if the person has income or not.

Continuing the previous example where information has been collected about the people in the household, the attribute `hasIncome` has been added to the Person entity to indicate if income information should be collected for the person, as follows:

```
<xs:element name="Person">
  <xs:complexType>
    ...
    <xs:attribute name="hasIncome" type="IEG_BOOLEAN"/>
```

Figure 24: Has Income Person Schema

Like questions, list questions must be added to a cluster. Where list questions differ is that you must specify the type of the entities that will be displayed in the list. The ID of the list question corresponds to the name of the boolean attribute that should be set if the user selects an item in the list. As with questions, a list question should have a label to indicate the purpose of the question. List questions should also have an item label element. The item label specifies which attribute from the entities should be used to identify the entities in the list. In the following example, the first name of the household members is displayed to identify them.

```
<question-page id="AnyoneHaveIncome">
...
  <cluster>
    <list-question id="hasIncome" entity="Person">
      <label id="HasIncome.Label">
             <![CDATA[Which people have income?]]>
      </label>
      <item-label>
        <label-element attribute-id="firstName"/>
      </item-label>
    </list-question>
  </cluster>
</question-page>
```

Figure 25: List question

So rather than adding a question in the loop where the household member details are gathered, once the household member details have been captured a list containing the household members can be displayed. The user can then select the members that have income.

List questions are particularly useful when used in conjunction with a for-each loop, referencing the question that was set in the list-question in the criteria expression of the loop. List questions can also be used with entity types other than Person.

Single-select

List questions can also be used when the selection should be mutually exclusive. When the `single-select` attribute of a list question is set to `true`, only one of the items in the list can be selected.

If for example, the requirement is to indicate which household member is the primary care giver, an attribute can be added to the Person entity and a single-select list question can be added to the script:

```
<xsd:element name="Person">
  <xsd:complexType>
    ...
    <xsd:attribute name="primaryCareGiver" type="IEG_BOOLEAN"/>
```

Figure 26: Primary Care Giver Person Schema

```
<question-page id="PrimaryCareGiver" ...>
...
  <cluster>
    <list-question id="primaryCareGiver" entity="Person"
          single-select="true" criteria="age > 14">
      <label id="PrimaryCareGiver.Label">
        <![CDATA[Which person is the primary care giver?]]>
      </label>
      <item-label>
        <label-element attribute-id="firstName" />
      </item-label>
    </list-question>
  </cluster>
```

Figure 27: Single-select List Question

The above list question will cause list of the household members that are over 14 years old to be displayed with a radio button next to each Person, thus allowing only one to be selected.

Code-table questions

If an attribute is defined in a DS schema as a code table, the default behavior is to display the question as a drop-down menu. You can have single-select or multi-select code-table questions.

When using drop-down menus in your questions, ensure that you do not leave blank responses as this can cause invalid HTML. To produce valid HTML, you must ensure that drop-down menus always have a value for the default selected option. For example, “--Please choose an option--”.

Single-select code-table questions

By default, only one answer can be selected in the menu. For example, to capture a household member's home state, add a domain definition to represent the `AddressState` code table and add a home state attribute to the `Person` entity as follows:

```
<question-page id="AboutYouPage" entity="Person">
...

  <cluster>
    <question id="homeState">
      <label id="State.Label">
        <![CDATA[Please select your home state:]]>
      </label>
    </question>
  </cluster>
```

You can add a question to capture the home state information to the script as follows:

```
...
<xsd:simpleType name="IEG_STATE_ADDRESS">
  <xsd:annotation>
    <xsd:appinfo>
      <D:options>
        <D:option name="code-table-name">AddressState</D:option>
      </D:options>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:restriction base="IEG_CODETABLE_CODE" />
</xsd:simpleType>
...

<xsd:element name="Person">
...
  <xsd:attribute name="homeState" type="IEG_STATE_ADDRESS" />
```

When the script runs, the question is displayed as a drop-down menu.

Multi-select code-table questions

You can also define code-table questions so that the user can make multiple selections.

When a code-table question is single-select, the answer to the question can be stored in a single attribute of an entity. For multi-select code-table questions with multiple answers, you must add a sequence to store the answers and define a new entity type for the answers in the sequence.

```
<xsd:element name="Person">

  <xsd:complexType>
    <xsd:sequence minOccurs="0">
      <xsd:element ref="State" minOccurs="0"
          maxOccurs="unbounded" />
      </xsd:sequence>
...
  </xsd:complexType>
</xsd:element>

<xsd:element name="State">
  <xsd:complexType>
    <xsd:attribute name="stateCode" type="IEG_STATE_ADDRESS" />
  </xsd:complexType>
</xsd:element>
```

To make a code-table question multi-select, set the `multi-select` attribute of the question to `true`.

For a multi-select code-table question, you must map the cluster that contains the question to the new entity type that represents the answers. In the example, the cluster is mapped to the `State` entity.

The page that contains the multi-select question must be mapped to the entity that contains the sequence. In this example, the page must be mapped to the `Person` entity.

Finally, for the options in a multi-select code-table question to be displayed, you must add a `layout` element to the question to specify the number of rows to display. If the number of options exceeds the number of rows that are specified in the layout, a scroll bar is added.

```
<question-page id="AboutYouPage" entity="Person">
...

  <cluster entity="State">
    <question id="stateCode" multi-select="true">
      <label id="State.Label">
        <![CDATA[Please select the states you lived in:]]>
      </label>
      <layout>
        <num-rows>4</num-rows>
      </layout>
    </question>
  </cluster>
```

When the script runs, the question is displayed as a list of the code-table descriptions with one checkbox for each item.

Conditional Elements

IEG scripts can have multiple different conditional elements: sections, pages or clusters. Conditional elements can be shown or hidden based on answers from previous pages or on data pre-populated in the DS.

Conditional Sections

It is possible to remove sections from a script execution by evaluating an expression at the start of the execution: if the section is not visible, it will not be listed in the sections panel and the expression will not be re-evaluated during the script execution.

Using a pre-populated DS as described in Pre-Populating Scripts with Captured Data on page 62, we can set a flag on an entity depending on circumstances external to the script. Let's say we have an entity called IntakeInformation that has a boolean attribute "collectIncomeInformation". We can specify an Income section in our script:

```
...
<section
 visible="IntakeInformation.collectIncomeInformation==true">
   ...
</section>
...
```

Figure 28: Visible Attribute of a Section

This will hide the Income section if the "collectIncomeInformation" attribute is false, as if the section was not present in the script definition.

If a section needs to be enabled or disabled depending on answers from previous sections, it is possible to wrap all the pages of a section in a single condition. Unlike the visible attribute, this condition will be evaluated whenever the section is encountered, which means it is possible to go

back and change an answer that affects the navigability of a section. The section will still appear in the sections panel but will be grayed out so the user cannot click on it.

The preceding example can be modified so that the "collectIncomeInformation" question is asked at the start of the script. The Income section can then be modified as follows:

```
...
<section>
   <condition
     expression="IntakeInformation.collectIncomeInformation">
     ...
   </condition>
</section>
...
```

Figure 29: Conditional Section

Conditional Pages

Pages can be displayed or not based on the value of a condition expression. Loops can be also wrapped in these conditions.

The conditional section previously mentioned where one condition wraps all the section's content is an example of conditional pages.

Conditional Clusters

Clusters can also be wrapped in a condition element. If the expression of the condition element does not refer to any of the questions on the same page the cluster is a static conditional cluster. That is because it can be determined before the pages is displayed whether to display the cluster or not.

For example, if information about household members has been gathered you may wish to add another page to ask further personal details including whether the person is pregnant. A new `isPregnant` attribute should be added to the Person entity to store this information:

```
<xsd:element name="Person">
  <xsd:complexType>
    ...
    <xsd:attribute name="isPregnant" type="IEG_BOOLEAN"/>
```

Figure 30: Additional Person Attribute

Of course, this question is only applicable if the gender is female. Therefore the cluster can be wrapped in a condition and it will only be displayed if the condition expression evaluates to true. The new extra Person Details page can be defined as follows:

```
<question-page id="AboutTheClientContinued" entity="Person" ...>
  <condition expression="Person.gender==&amp;quot;SX2&amp;quot;">
    <cluster>
      <question id="isPregnant" mandatory="true">
          <label id="IsPregnant.Label">
              Are you pregnant?
          </label>
          <help-text id="IsPregnant.HelpText">
              Are you pregnant?
          </help-text>
      </question>
    </cluster>
  </condition>
```

```
</question-page>
```

Figure 31: Static Conditional Cluster

Alternatively, if any of the questions referenced in the condition expression are on the same page, the cluster is then a dynamically conditional cluster. The means that the cluster will be displayed and hidden as the user changes answers to questions on the page. This dynamic feature of IEG requires that JavaScript is enabled in the browser. The expressions of dynamically conditional cluster may not refer to custom functions, as the expressions are evaluated without making a server call.

Without changing the DS schema, if the example above is changed so that the conditional cluster is defined on the same page as the gender question the cluster will be a dynamically conditional cluster.

```
<question-page id="AboutTheClient" entity="Person" ...>
...
  <cluster>
    <title id="DetailsCluster.Title">
      <![CDATA[Personal Details]]>
    </title>
...
    <question id="gender" mandatory="true">
      <label id="Gender.Label">
        <![CDATA[Gender:]]>
      </label>
    </question>
...
  <condition expression="Person.gender==&quot;SX2&quot;">
    <cluster>
      <question id="isPregnant" mandatory="true">
        <label id="IsPregnant.Label">
        <![CDATA[Are you pregnant?]]>
        </label>
      </question>
    </cluster>
  </condition>
</question-page>
```

Figure 32: Dynamically Conditional Cluster

The pregnancy question will dynamically appear or disappear when the value selected for the gender changes. Dynamic behavior on a page can be triggered by text fields, date fields, checkboxes, radio buttons, select elements. Dynamic behavior cannot be triggered by the answer to a multi-select question or a question matrix, due to the restrictions of the expression syntax.

It should be noted that only one level of condition is allowed around a cluster, i.e. conditional clusters cannot be nested in other conditions. The condition expression for a dynamically condition cluster may refer to questions on the same page that are themselves defined in dynamically conditional cluster. This creates a cascading dependency between clusters.

Question Matrices

A question matrix will display a list of questions based on a codetable and for each of these codetable values and each entity, a checkbox will be displayed to allow the user to select all the values that apply to a particular entity.

The list questions presented in List Questions on page 28 ask the same boolean question about a group of entities. It is possible to ask the same codetable question for a group of entities using question matrices.

For example, suppose there is a requirement to capture possible levels of substance abuse for each household member, a new domain definition can be added to represent the `SubstanceAbuse` codetable and an attribute to store the level of substance abuse can be added to the Person entity as follows:

```
<xsd:simpleType name="IEG_SUBSTANCEABUSE">
  <xsd:annotation>
    <xsd:appinfo>
      <D:options>
        <D:option name="code-table-name">SubstanceAbuse</
D:option>
      </D:options>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:restriction base="IEG_CODETABLE_CODE" />
</xsd:simpleType>

<xsd:element name="Person">
  <xsd:complexType>
    ...
    <xsd:attribute name="substanceAbuse"
                   type="IEG_SUBSTANCEABUSE" />
```

Figure 33: Substance Abuse Attribute

The question matrix is then defined as a regular list question, only the fact that it is based on a codetable instead of a boolean will cause it to be displayed differently.

```
...
<list-question entity="Person" id="substanceAbuse"
      criteria="age > 14">
  <label id="SubstanceAbuse.Label">
    <![CDATA[Substance Abuse:]]>
  </label>
  <item-label>
    <label-element attribute-id="firstName" />
  </item-label>
</list-question>
```

Figure 34: Question Matrix Code Example

The example above, of a question matrix that collects substance abuse information about multiple household members, will be displayed in the IEG Player as a matrix with each row corresponding to a codetable description and each column to a Person.

Fast Path Navigation

By default, when a user reiterates through a script all the pages are re-displayed which can become arduous especially in large households. Fast Path navigation enables end users to go through IEG scripts more quickly by automatically skipping loop or conditional pages that have already been answered.

This functionality is optional and switched off by default. It can be activated on loops and conditions (to activate Fast Path navigation, see the *Authoring Scripts using Intelligent Evidence Gathering (IEG)*) guide.

The first time a fast path element is encountered, it behaves as normal. When the user navigates through the script subsequently only the new pages within these fast path elements will be displayed. The pages that were previously displayed will now be skipped. This functionality doesn't prevent from editing the data via the edit links on a summary page if necessary.

Fast Path can be used in the following scenarios:

- List Question driving a Loop
- Eligibility Criteria
- Fast Path Conditions
- Condition in Fast Path Loop

List Question driving a Loop

Using the same List Question as described in List Questions on page 28, we want to gather income information for the people in a household. We will use a nested fast path loop as described in the following example:

```
...
<loop loop-type="for-each" entity="Person"
  criteria="hasIncome==true" fast-path="true">
  <loop loop-type="while" loop-expression="hasMoreIncome"
    entity="Income">
    <question-page id="IncomePage" entity="Income"
      show-person-tabs="true">
      <title id="IncomePage.Title">
        <![CDATA[Income Details]]>
      </title>
      <cluster>
        <title id="IncomeDetails.Title">
          <![CDATA[Income Details]]>
        </title>
        <question id="type">
          <label id="Type.Label">
            <![CDATA[Type:]]>
          </label>
        </question>
        <question id="amount">
          <label id="Amount.Label">
            <![CDATA[Amount:]]>
          </label>
        </question>
        <question id="hasMoreIncome"
          control-question="true"
          control-question-type="IEG_BOOLEAN">
          <label id="ContinueQuestion.Label">
```

```
            <![CDATA[Does %1s have any more income?]]>
            <argument id="Person.firstName" />
          </label>
        </question>
      </cluster>
    </question-page>
  </loop>
</loop>
```

Figure 35: Fast Path List Question driving a Loop Code Example

The first time the list question is encountered, the pages following the loop will gather income for the people that have been selected. Then when re-visiting the page containing the list question, the following can occur:

- If the checkboxes are not modified, clicking Next will jump over the income loop and display the page after the loop.
- If some of the checkboxes are unselected, clicking Next will delete the incomes corresponding to the people that were unselected, jump over the income loop and display the page after the loop.
- If new checkboxes are checked, clicking Next will jump over the existing income pages, show new income pages for the newly selected people and then display the page after the loop.
- If new checkboxes are checked and others are unselected, clicking Next will delete the incomes corresponding to the people that were unselected,jump over the existing income pages, show new income pages for the newly selected people and then display the page after the loop.

Eligibility Criteria

Building on the previous scenario, we can filter the people that will be displayed in the list question (the loop does not need to be modified). Only the people over 18 will be eligible to enter income so a criteria is added to the list question. When reiterating through the script people may no longer match the criteria and therefore not appear in the list.

```
...
<list-question id="hasIncome" entity="Person" criteria="age >
 18">
  <label id="HasIncome.Label">
    <![CDATA[Which people have income?]]>
  </label>
  <item-label>
    <label-element attribute-id="firstName" />
  </item-label>
</list-question>
```

Figure 36: Fast Path List Question with Eligibility Criteria driving a Loop Code Example

This will behave as mentioned in the previous scenario, but if the date of birth of a person is modified, the following will happen:

- If the person becomes ineligible (under 18) and income had been entered, the corresponding income will get automatically deleted as soon as the new date of birth is submitted.
- If the person becomes eligible (over 18), it will be displayed in the list question (but not selected) the next time the list question page is displayed.

Fast Path Conditions

We can ask pregnancy details for female household members using a conditional page. If the condition is defined as fast path, the pregnancy details will be hidden when re-iterating over household members as the pages in the condition will only be displayed when reiterating through the script if the condition previously evaluated to false and something has changed so the condition now evaluates to true.

```
...
<question-page id="AboutYouPage" entity="Person">
  <title id="PrimaryPersonPage.Title">
    <![CDATA[About You]]>
  </title>
  <cluster>
    <title id="DetailsCluster.Title">
      <![CDATA[Personal Details]]>
    </title>
    <question id="firstName" mandatory="true">
      <label id="FirstName.Label">
        <![CDATA[First Name:]]>
      </label>
    </question>
    <question id="middleName">
      <label id="MiddleName.Label">
        <![CDATA[Middle Name:]]>
      </label>
    </question>
    <question id="lastName">
      <label id="lastName.Label">
        <![CDATA[Last Name:]]>
      </label>
    </question>
    <question id="gender" mandatory="true">
      <label id="Gender.Label">
        <![CDATA[Gender:]]>
      </label>
    </question>
    <question id="dateOfBirth" mandatory="true">
      <label id="DateOfBirth.Label">
        <![CDATA[Date Of Birth:]]>
      </label>
    </question>
  </cluster>
</question-page>
<condition expression="Person.gender==&amp;quot;SX2&amp;quot;"
  fast-path="true">
  <question-page id="PregnancyPage" entity="Person">
    <title id="PregnancyPage.Title">
      <![CDATA[About You: pregnancy]]>
    </title>
    <cluster>
      <title id="DetailsCluster.Title">
        <![CDATA[Personal Details About Your Pregnancy]]>
      </title>
      <question id="isPregnant" >
        <label id="IsPregnant.Label">
          <![CDATA[Are you pregnant?]]>
```

```
        </label>
      </question>
    </cluster>
  </question-page>
</condition>
```

Figure 37: Fast Path Conditions Code Example

When editing the personal details, the following can occur:

- If no change was made to the gender, clicking on Next will jump over the condition, whether it was displayed the first time or not.
- If the gender has changed from Male to Female, clicking on Next will display the conditional page to enter pregnancy details.
- If the gender has changed from Female to Male, clicking on Next will delete the pregnancy details and display the page after the condition.

Condition in Fast Path Loop

When a condition is defined inside a Fast Path loop, this will behave the same as when a criteria is used on the loop instead of nesting a condition, with the following exception: if the condition becomes true, the page contained within the condition cannot be displayed as the loop doesn't have a new iteration to show and therefore will be skipped. If the condition becomes false, the page and associated data will not be deleted as the condition is not re-evaluated.

It is therefore recommended to use a criteria on the loop instead of a condition.

```
...
<loop loop-type="for-each" entity="Person"
  fast-path="true">
  <condition expression="Person.hasIncome==true">
    <loop loop-type="while" loop-expression="hasMoreIncome"
      entity="Income">
      <question-page id="IncomePage" entity="Income"
        show-person-tabs="true">
        <title id="IncomePage.Title">
          <![CDATA[Income Details]]>
        </title>
        <cluster>
          <title id="IncomeDetails.Title">
            <![CDATA[Income Details]]>
          </title>
          <question id="type">
            <label id="Type.Label">
              <![CDATA[Type:]]>
            </label>
          </question>
          <question id="amount">
            <label id="Amount.Label">
              <![CDATA[Amount:]]>
            </label>
          </question>
          <question id="hasMoreIncome"
            control-question="true"
            control-question-type="IEG_BOOLEAN">
            <label id="ContinueQuestion.Label">
              <![CDATA[Does %1s have any more income?]]>
              <argument id="Person.firstName" />
```

```
            </label>
          </question>
        </cluster>
      </question-page>
    </loop>
  </condition>
</loop>
```

Figure 38: Condition in Fast Path loop Code Example

Implicit Delete

Wherever possible, the IEG engine tries to delete data as soon it finds out that it is no longer relevant.

If an answer is explicitly modified by the user (through a regular question, a list-question or a set-attribute, but not through a custom function call), the engine detects if this answer is used in a condition expression, a list-question criteria or a loop criteria. If that is the case, the expression or criteria is re-evaluated and if it becomes false, the corresponding pages are removed and the associated data gets deleted without the need to go through the script to encounter the expressions or criterion.

Three Field Date Picker

Whenever a date needs to be entered by a user, the default input field shows a plain text field and a date picker. The user is free to either enter the date manually (for example, '5/6/2010') or to click on the relevant date in the calendar widget. The former can be effective for power users and the latter is quite handy for dates that are not too far in the past or in the future. But for non-power users that need to enter dates well in the past (the obvious example is a date of birth), the default date picker can require a lot of clicks to get to the relevant date.

Another date picker exists in the form of three fields displayed side by side, each representing the three date parts (day, month, year). This lets users select dates in a much faster way and prevents invalid dates being typed.

To switch from the default date picker to the three-field one, the date attribute being captured should be defined in the data store schema using the *'IEG_THREE_FIELD_DATE'* type or a domain that extends it:

```
<xs:element name="Person">
  <xs:complexType>
    ...
    <xs:attribute name="dateOfBirth" type="IEG_THREE_FIELD_DATE"/>
```

The order of the drop-down elements and the display values of the month element reflect the date format, as configured by the date format property in the *ApplicationConfiguration.properties* file. The day drop-down is populated with numbers ranging from 1 to 31. Validation at infrastructure level prevents users from selecting an invalid date, for example, February 31, 2015. The year drop-down element is populated with values starting 100 years in the past to 30 years in the future. The range and order of the options are not configurable.

1.5 Other Script Development Considerations

You might need to display data in a read-only mode or to invoke external functionality. Some things must be considered when maintaining IEG scripts, placing scripts under source control and loading scripts into the database.

Displaying Data as Read-Only

Sometimes the answers to some questions need to be displayed to the user in such a way that they cannot be modified. This is already the case on summary pages where users can review the answers and use the back button or edit links to modify them.

On a question page, a "read-only" boolean attribute can be set to true indicating that all the questions displayed on the page will not be editable.

A more sophisticated mechanism exists: "read-only-expression" attributes can be used on different script elements (sections, all types of pages, clusters, questions and list questions). If the expression evaluates to true, this will apply to all the questions contained in the element. At its simplest, the expression will be "true" if the element needs to be unconditionally read-only. On a summary page, the result is that add, edit and delete links are not displayed.

In the case of read-only-expression defined for cluster, question and list question script elements, if any of the questions referenced in the expression are on the same page as the script element the script element is then dynamically enabled or disabled as opposed to just being read-only. This means that questions will be enabled and disabled as the user changes answers to other questions on the page. Where the read-only-expression of a cluster references a question on the same page all the questions contained in the cluster will be enabled and disabled. This dynamic feature of IEG requires that JavaScript is enabled in the browser. The expressions to dynamically enable and disable questions may not refer to custom functions, as the expressions are evaluated without making a server call.

Dynamic read-only-expressions may also refer to questions on the same page that are themselves dynamically enabled and disabled. This creates a cascading dependency between questions. Care should be taken when defining expressions with cascading dependencies as IEG does not take into account whether the questions referred to in the read-only-expression is enabled or not, just the value of the question. This may be confusing for the user as it may not be apparent what is controlling the enabling and disabling of a question.

When a question is displayed if the corresponding Datastore attribute has a value it will be displayed even if the question is initially disabled. The question may then be enabled by the user and the user may change the answer. If the question is disabled its value will set back to the value it had when initially displayed. When a page is submitted the Datastore attribute will not be updated unless the question is enabled. Therefore if the page is redisplayed the original value of the Datastore attribute will be displayed again.

It is not possible to mark a question as mandatory if it also has a dynamic read-only-expression on the question itself or one of its parent elements.

Dynamically enabling and disabling script elements is not supported on Relationship Pages.

The information gathered in loops can be displayed on summary pages using lists, but it is also possible to use this list construct on regular pages without the need to specify a read-only-

expression in one of the elements wrapping the list. The only difference with summary lists is that links are not allowed.

Another possibility is to make a whole script read-only. This is useful, for example, if a case-worker needs to review a script without being able to change any of the answers. The script is set to read-only through the IEGRuntimeAPI by setting a read-only flag on the script execution, as shown below:

```
...
//Set read only flag.
IEGRuntime runtimeAPI = new IEGRuntime();
IEGScriptExecutionID runtimeExecID = new IEGScriptExecutionID();
runtimeExecID.executionID = execution.getExecutionID();
IEGReadOnlyFlag readOnlyFlag = new IEGReadOnlyFlag();
readOnlyFlag.readOnlyFlag = true;
runtimeAPI.setReadOnlyFlag(runtimeExecID, readOnlyFlag);
...
```

Figure 39: Setting the read-only flag on a script execution

Invoking external functionality by using expressions and custom functions

Expressions can be defined in multiple places in scripts to define behavior such as loops or conditions. These expressions can refer to answers and can combine them using various operators, and they can call external functions, which are called custom functions.

Expressions cannot call custom functions when used on dynamic conditional clusters as these expressions are evaluated in the browser.

Custom functions are defined in Java™ code and depending on their usage, they can be one of two types:

- Custom functions that can take parameters, possibly making a call to external functionality, and return a value. They do not change the content of the datastore. They are used in most expressions.
- Custom functions where you want to update the content of the datastore use the stand-alone `callout` element. The returned value is irrelevant but it must be a Boolean value. These custom functions must not update values that were answered before the callout. The IEG Engine is unaware of updates that are made outside the context of the script, and cannot take any actions that might be required by the updates.

For example, on a personal details page, you might need to call external functionality to validate a US ZIP code that a user has supplied or to populate a US State field based on a supplied ZIP code. Let's look at how you might implement these two use cases.

Update the datastore schema to add two extra attributes to the Person entity, as follows:

```
<xsd:attribute name="state" type="IEG_STRING"/>
<xsd:attribute name="zipCode" type="IEG_STRING"/>
```

First, let's try to validate a ZIP code against a state (this is a naive implementation): a ZIP code must be five digits long and the first three digits indicate the state.

The personal details page and the corresponding summary page can be modified with two extra mandatory questions in the script definition: `state` and `zipCode`:

```
<question id="state" mandatory="true">
    <label id="State.Label">
        State:
    </label>
    <help-text id="State.HelpText">
        The state you live in
    </help-text>
</question>
<question id="zipCode" mandatory="true">
    <label id="ZipCode.Label">
        ZIP Code:
    </label>
    <help-text id="ZipCode.HelpText">
        Your ZIP code
    </help-text>
</question>
```

Create the custom function that performs the validation as a Java™ class in the `curam.rules.functions` package. The following custom function validates the ZIP code:

```
public class CustomFunctionValidateZipCode extends CustomFunctor {

  public Adaptor getAdaptorValue(final RulesParameters rp)
  throws AppException, InformationalException {

    final List<Adaptor> parameters = getParameters();
    final String zipCode =
      ((StringAdaptor) parameters.get(0)).getStringValue(rp);
    final String state =
      ((StringAdaptor) parameters.get(1)).getStringValue(rp);
    boolean valid = false;

    if (zipCode.length() == 5) {
      final String prefix = zipCode.substring(0, 3);
      //lookup the state prefixes
      if (prefix.equals("100")
        && state.equalsIgnoreCase("New York")) {
        valid = true;
      }
      if (prefix.equals("900")
        && state.equalsIgnoreCase("California")) {
        valid = true;
      }
    }

    return AdaptorFactory.getBooleanAdaptor(Boolean.valueOf(valid));
  }

}
```

Insert the following metadata for the custom function in *`<yourcomponent>/rulesets/functions/CustomFunctionMetaData.xml`*:

```
<CustomFunctor name="CustomFunctionValidateZipCode">
  <parameters>
    <parameter>
      curam.util.rules.functor.Adaptor$StringAdaptor
    </parameter>
    <parameter>
      curam.util.rules.functor.Adaptor$StringAdaptor
    </parameter>
  </parameters>
  <returns>curam.util.rules.functor.Adaptor$BooleanAdaptor</returns>
</CustomFunctor>
```

In the example, the `ValidateZipCode` custom function doesn't access an external database to look up the corresponding state. Ideally, the custom function would look up and check the returned state against the entered state. For simplicity, two ZIP code prefixes were hardcoded.

The validation is then inserted in the script definition for the personal details page.

```
<validation
  expression="ValidateZipCode(Person.zipCode, Person.state)">
    <message id="InvalidZipCode">
        The ZIP code is invalid.
    </message>
</validation>
```

When the user clicks **Next**, the answers to the `zipCode` and `state` questions are passed to the custom function, which returns true if the answers are valid. The next page is then displayed.

If the custom function returns false, the message that is specified in the validation is displayed at the top of the Person details page, blocking the access to the **Next** page until valid answers are submitted.

The custom function has no side effect as it doesn't alter anything. It performs an operation based on the parameters and returns a result.

You can also remove the mandatory flag on the two new questions and validate the answers only if they have both been supplied. Change the validation expression to use the `isNotNull` custom function that is provided in the application, which checks whether the parameter is null:

```
"not(isNotNull(Person.zipCode) and isNotNull(Person.state))
    or ValidateZipCode(Person.zipCode, Person.state)"
```

Alternatively, you can populate the state question by using the `zipCode`. On the Person details page, ask for the `zipCode` only, with a mandatory flag, and the summary page then displays both `state` and `zipCode`.

You can define the following custom function:

```
...
public class CustomFunctionpopulateState extends CustomFunctor {

  public Adaptor getAdaptorValue(final RulesParameters rp)
  throws AppException, InformationalException {

    final IEG2Context ieg2Context = (IEG2Context) rp;
    final long rootEntityID = ieg2Context.getRootEntityID();
    String schemaName;
    //schemaName has to be hard-coded or retrieved outside of IEG
    Datastore ds = null;
    try {
      ds =
        DatastoreFactory.newInstance().openDatastore(
            schemaName);
    } catch (NoSuchSchemaException e) {
      throw new AppException(IEG.ID_SCHEMA_NOT_FOUND);
    }

    Entity applicationEntity = ds.readEntity(rootEntityID);

    Entity personEntity =
      applicationEntity.getChildEntities(
        ds.getEntityType("Person"))[0];
    String zipCode = personEntity.getAttribute("zipCode");
    String state = "Unknown";
    final String prefix = zipCode.substring(0, 3);
    //lookup the state prefixes
    if (prefix.equals("100")) {
      state = "New York";
    }
    if (prefix.equals("900")) {
      state = "California";
    }
    personEntity.setAttribute("state", state);
    personEntity.update();
    return AdaptorFactory.getBooleanAdaptor(new Boolean(true));
  }

}
```

And its metadata:

```
<CustomFunctor name="CustomFunctionpopulateState">
  <returns>curam.util.rules.functor.Adaptor$BooleanAdaptor</returns>
</CustomFunctor>
```

Between the Person details page and the summary page, insert a `callout` element in the script definition to call this custom function, as follows:

```
<callout id="populateAddress" expression="populateState()"/>
```

This time, the custom function alters the datastore by populating the state on the Person entity. The context contains the root entity ID and executionID, making it easier to update the datastore. If the callout is in a loop, the context also contains the current entity ID.

Reusing Scripts

It is possible to break down a script definition into multiple files thus providing a re-use mechanism.

In order to achieve this, a script definition will have to reference subscripts. Each of these subscripts will be a standalone script that can be run independently.

Here is an example of a script that can be used as a subscript:

```
<?xml version="1.0" encoding="UTF-8"?>
<ieg-script ...>
  <identifier id="Subscript" scriptversionnumber="V1"
 type="Test" />
  <question-page ...>
    ...
  </question-page>
  ...
</ieg-script>
```

Figure 40: Subscript Containing Pages

The script in the above example code snippet can be included in another script as a subscript, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<ieg-script ...>
  <identifier id="Script" scriptversionnumber="V1" type="Test" />
  <section>
    <ieg-sub-script>
         <identifier id="Subscript"
         scriptversionnumber="V1" type="Test" />
    </ieg-sub-script>
  </section>
  <section>
    ...
  </section>
  ...
</ieg-script>
```

Figure 41: Inclusion of a Subscript in a Script

The possible point of insertion of a subscript in a script can be as follows:

- If the script contains sections and the subscript also contains sections, the subscript will have to be inserted at the top level, under the parent ieg-script element.
- If the script contains sections and the subscript doesn't contain sections, the subscript will have to be inserted in a section of the parent script.
- If the script doesn't contain sections, the subscript cannot contain sections. It will be inserted at the top level, under the ieg-script element.

Another limitation to keep in mind is that a subscript can appear only once in a script as the page IDs must be unique within the resulting script.

Note that a script might be used as a subscript elsewhere. When modifying scripts, ensure that any referencing scripts are re-tested to ensure the changes do not have an undesired impact.

Source Control and Versioning

IEG script definitions are stored in the database. When editing an IEG script using the IEG Editor, the script is edited in place and updated directly in the database. IEG script definitions are development artifacts and from a software configuration management point of view it is important that these artifacts are placed under source control as you would with any other artifacts.

It is possible to download a script definition from the IEG script administration screens. When the option to download a script is chosen, the script is first retrieved from the database, then the

properties files associated with the script definition are retrieved from the Resource Store and the textual properties are "injected" into the script definition before it is made available. However downloading a script in this way does not provide all the resources that may be associated with a script definition. For example, it does not provide properties files in multiple locales and it does not provide images and icons. For more information about the database representation of an IEG script, see .

When populating the database with script definitions, it is important to be aware of the differences in functionality between importing a script through the IEG script administration screens and loading a script definitions via DMX files.

Rendering Custom HTML on a Summary Page

You can use the `custom-output` display element to render custom HTML on a summary page. The `custom-output` display element enables data from a data store instance to be retrieved and accessed from a custom renderer so that the data can be rendered by using custom HTML.

Some summary pages might contain much material that is displayed on multiple pages. Therefore, you might want to implement custom layout requirements so that you can use user interface design concepts. For example, you might want to use cards to format Intelligent Evidence Gathering (IEG) summary data in a user-friendly layout that is oriented to data types.

By using the `custom-output` display element to render custom HTML, you can implement custom-rendered clusters that include images and rich text formatting on a summary page. Custom-rendered clusters enable a summary of the data that is provided by a client to be displayed in a more visually appealing format.

Script description

The following sample shows an example instantiation of the `custom-output` display element in a `summary-page` element.

```
…
<summary-page id="HouseholdSummary">
  <title id="HouseholdSummary.Title"><![CDATA[Household Summary]]></title>
  <icon image="sample_title_household" />
  <custom-output
    class-name="curam.ieg.player.custom.IEGSampleCustomRenderer"
    data-accessor="curam.ieg.player.custom.IEGSampleCustomOutputDataAccessor" />
  ...
</summary-page>
…
```

- **`custom-output` element**

  The `custom-output` element is an optional child element of the `summary-page` display element. You can use the `custom-output` element only within a `summary-page` element. You can include multiple `custom-output` elements in a single summary page or in multiple summary pages, in any order, similarly to `cluster`, `list`, `condition`, and `relationship-summary-list` elements.

  The `custom-output` element has two mandatory attributes, a `data-accessor` attribute, and a `class-name` attribute. The `custom-output` element has no child elements.

- **`custom-output` attributes**
  - **`data-accessor` attribute**

    The `data-accessor` attribute represents the name of the data accessor class that is used to retrieve entity data that is rendered as required in the custom renderer class that is specified in the `class-name` attribute. The fully qualified name of the data accessor class must be specified in the attribute field.

    For each `custom-output` element, the data that is retrieved by the specified data accessor class is unique to the associated, specified custom renderer class. For example, if two `custom-output` elements are instantiated on a summary page, a different data-accessor class but the same custom renderer class can be specified for both elements. For the first `custom-output` element, the data accessor class retrieves only person entity data. For the second `custom-output`element, the data-accessor class retrieves only income entity data. Therefore, for the first `custom-output` element, the custom renderer can access only person entity data. For the second `custom-output` element, the custom renderer can access only income entity data.
  - **`class-name` attribute**

    The `class-name` attribute represents the name of the custom renderer class that is used to output custom HTML on a summary page. The `class-name` attribute value must specify the fully qualified name of the custom renderer class.

How to use the custom-output element

To render custom HTML on a summary page, you must configure a data accessor class and a custom renderer class.

Data accessor class

You must create a data accessor class that is used to retrieve entity data from a data store instance. The data can then be processed within a custom renderer class. The data accessor class must implement the `curam.ieg.external.impl.IEGCustomOutputEntityData` interface and its `getRequiredEntitiesForCustomOutput(IEGScriptExecution)` method. The `getRequiredEntitiesForCustomOutput` method returns a list of string objects that represent the entity data and that can be read by a custom renderer on the client side.

Optionally, a data accessor class can also inherit from the `curam.ieg.external.impl.IEGCustomOutputDataAccessor` class. The `curam.ieg.external.impl.IEGCustomOutputDataAccessor` class contains several utility APIs that are provided to enable entity data from the data store to be accessed more easily. The following list gives a brief description of the APIs. For a full description, see the Javadoc documentation for the methods in the `curam.ieg.external.impl.IEGCustomOutputDataAccessor` class.

- **`List<String> getRequiredEntities(List<String>, IEGScriptExecution)`**
  This method returns a list of XML strings that contain entity data for a list of specified entity type names.
- **`List<String> getTopLevelEntities(IEGScriptExecution)`**
  This method returns a list of strings that contains the names of the direct child entities of the root entity.

- **`List<String> getEntityXML(List<Entity>)`**
  This method returns entity data in an XML string format. The following example shows a sample XML string for an income entity with two attributes, type and amount:

  ```
  <Income type="Wages" amount="10000"/>
  ```

- **`List<String> getChildEntityXML(List<Entity>, String, IEGScriptExecution)`**
  This method returns a list of strings that contain entity data that is of a specified child entity type.

Custom renderer class

You must also create a custom renderer class that is used to create and render custom HTML. The custom renderer class must inherit from the `curam.ieg.player.IEGCustomOutputRenderer` class. The `curam.ieg.player.IEGCustomOutputRenderer` class contains several utility APIs that are provided to enable a custom renderer to access and process data that is retrieved by a data accessor class. The following list gives a brief description of the APIs. For a full description, see the Javadoc documentation for the methods in the `curam.ieg.player.IEGCustomOutputRenderer` class.

- **`List<Node> getEntityXMLData(Component, RendererContext)`**
  This method gets the required entity data that was retrieved by the associated data accessor class for a custom renderer. Entity data for a custom renderer is retrieved in an XML string format and returned as a list of node objects.
- **`List<String> getEntityData(Component, RendererContext)`**
  This method gets the required entity data was retrieved by the associated data accessor class for a custom renderer. Entity data for a custom renderer is retrieved and returned as a list of strings.
- **`List<Node> parseXMLString(String)`**
  This method parses an XML String into a list of node objects.
- **`String getLocalizedText(Component, RendererContext, String, List<String>)`**
  This method returns the localized text for a specified property.
- **`String getLocalizedCodeTableItemValue(String, String, RendererContext)`**
  This method returns the localized value of the specified code table item.
- **`String getLocalizedMoneyString(String)`**
  This method returns a localized string that represents a money value.
- **`String getLocalizedBooleanString(RendererContext, String)`**
  This method returns a localized string that represents a Boolean value.
- **`String getLocalizedDateString(String)`**
  This method returns a localized string that represents a date value.
- **`String getLocalizedImageFromResourceStore(String, RendererContext)`**
  This method returns the relative URI for an image from the resource store for the current locale.

Both data accessor classes and custom renderer classes can be reused multiple times by different `custom-output` elements.

Guidelines for using the custom-output element

When you add a `custom-output` element to a summary page, if custom properties are required to be used within a custom renderer class, you must add the properties to the properties file for the summary page. The custom renderer class must then reference the required properties directly.

For example, you can retrieve text from a properties file within a custom renderer by calling the `getLocalizedText(Component, RendererContext, String, List<String>)` method. If required, pass the property name and a list of arguments for the property as parameters into the method. The localized text for the property is returned. Similarly, you can also retrieve images from the resource store by calling the `getLocalizedImageFromResourceStore(String, RendererContext)` method. Pass the name of the image as a parameter into the method, and the relative image URI is returned. You can set the URI value as the source for an image element.

The `custom-output` element enables data from a data store instance to be retrieved and then rendered on a summary page by using custom HTML. When you use the `custom-output` element, use the following guidelines:

- **API methods**
  Several API methods are provided in the `curam.ieg.player.IEGCustomOutputRenderer` class to handle the localization of the different data types that can be used within custom renderers. It is recommended that you use the methods to return localized versions for each of the data types within a custom renderer. No API methods are supplied for number formatting. The following table lists the data types for which localization APIs are provided:

  Table 2: Data types and associated API localization methods

Data type	Method
String	`getLocalizedText(Component, RendererContext, String, List<String>)`
Codetable **Note:** Codetable hierarchies are not supported.	`getLocalizedCodeTableItemValue(String, String, RendererContext)`
Money	`getLocalizedMoneyString(String)`
Boolean	`getLocalizedBooleanString(RendererContext, String)`
Date	`getLocalizedDateString(String)`

- **Retrieving entity data**
  - You must use data accessor classes to retrieve entity data that can then be rendered in a custom renderer class. Data accessor classes support only the reading of data from the data store. The editing of data store data within a data accessor class might cause adverse effects and is not supported.
  - Retrieving a large amount of entity data from the data store within a data accessor class can affect the overall performance of an IEG page. Therefore, take care when you consider how many `custom-output` elements are included on a summary page.
  - Within custom renderer classes, where it is possible to retrieve data such as images and properties, ensure that you retrieve data only from the resource store.

- **Instantiating the `custom-output` element**
  Import IEG script definitions that contain the new `custom-output` element into the database by using one of the following options:

  - `ieg.importscript` command line build option
  - `database` command line build option

  Do not use the option in the administration section of the application to import an IEG script that contains `custom-output` elements. Any associated properties that are specified for a custom renderer are not included in the script definition. Instead, the properties are defined directly in the associated property file of the summary page where the `custom-output` elements are instantiated.
- **Rendering custom HTML**

  - The HTML that is output by a custom renderer can contain only read-only fields. The HTML output must not contain any editable fields because no infrastructure is provided that allows data to be either entered or modified through the mechanism of the custom output feature.
  - The custom HTML that is created must be valid HTML within the context of an IEG page. For example, it is valid to insert a `div` tag as a starting point for the custom HTML, but inserting an `html` tag as a starting point within a custom renderer might cause adverse effects.
  - Ensure that the custom renderer does not delegate to other renderers to output HTML on a summary page.
  - Ensure that the custom HTML complies with accessibility standards.
  - If specific custom HTML for right-to-left languages and high contrast mode is required, use the custom renderer to create the custom HTML.

Sample code for rendering custom HTML

The following sample code shows an example of how to set up the `custom-output` element and its required classes. The sample code outputs the first names of all the people in the household in a custom cluster at the top of a summary page. The example also demonstrates how text from a properties file and images can be retrieved from the resource store and used in a custom renderer.

Data accessor class

The following sample data accessor code implements the supplied `curam.ieg.external.impl.IEGCustomOutputEntityData` interface and its `getRequiredEntitiesForCustomOutput(IEGScriptExecution)` method. The code then specifies that person entity data is required for the particular `custom-output` element. To retrieve the person entity data, the `getRequiredEntities(List<String>, IEGScriptExecution)` API method is

called. The data accessor class can access the API method because the class inherits from the `curam.ieg.external.impl.IEGCustomOutputDataAccessor` class.

```
…

package curam.ieg.player.custom;

public class IEGSampleCustomOutputDataAccessor extends
IEGCustomOutputDataAccessor implements IEGCustomOutputEntityData {

public IEGSampleCustomOutputDataAccessor() {
}

public List setRequiredEntities() {

// Create a list of required entity types

List entityListForCustomOutput = new ArrayList();

entityListForCustomOutput.add("Person");

return entityListForCustomOutput;

}

@Override
public List getRequiredEntitiesForCustomOutput(final IEGScriptExecution
 execution)
  throws AppException, InformationalException {

// Returns a list of XML strings for Entities of the required entity types.

final List entityList = getRequiredEntities(setRequiredEntities(), execution);

return entityList;
}
}
```

Custom renderer class

The sample custom renderer class accesses the data that was retrieved by the data accessor class, processes it, and then creates HTML to render the data. To render the data, the custom renderer

inherits from the supplied `curam.ieg.player.IEGCustomOutputRenderer` class. You can style HTML in custom renderers by using either CSS, or inline styling.

```
…
package curam.ieg.player.custom;

public class IEGSampleCustomRenderer extends IEGCustomOutputRenderer {

@Override
public void render(final Component component,
final DocumentFragment fragment, final RendererContext context,
final RendererContract contract) throws ClientException,
DataAccessException, PlugInException {

// Get the owner document from the fragment
final Document ownerDocument = fragment.getOwnerDocument();

// CSS classes for customOutput divs
final String customOutputClass = "customOutput";

// Create the required HTML elements
final Element customOutputDiv = ownerDocument.createElement(kDiv);
customOutputDiv.setAttribute(kClass, customOutputClass);

final Element customOutputHouseholdContentDiv =
ownerDocument.createElement(kDiv);

final Element customOutputHouseholdTitle =
ownerDocument.createElement(kH2);

/*
* Calls the provided API method to retrieve the localized title text from
* a properties file.
*/
final String householdTitle = getLocalizedText(component,
context, "CustomHousehold.Title",
new ArrayList());

customOutputHouseholdTitle.setTextContent(
householdTitle);

// Create a div to hold the household icon
final Element customOutputHouseholdImageDiv =
ownerDocument.createElement(kDiv);

// Create the household icon using an image from the resource store.
final Element customOutputHouseholdImage =
ownerDocument.createElement(kImg);

/*
* Calls the provided API method to retrieve the image from the resource
* store
*/
customOutputHouseholdImage.setAttribute(kSrc,
getLocalizedImageFromResourceStore("household.png", context));

customOutputHouseholdImageDiv.appendChild(
customOutputHouseholdImage);
customOutputHouseholdContentDiv.appendChild(
customOutputHouseholdTitle);

// Set the attribute name for the entity
final List personAttributeNames = new ArrayList();
personAttributeNames.add("firstName");
/*
* Calls the provided API method to access the entity data the data
* accessor class retrieved.
*/
final List nodeList = getEntityXMLData(component, context);

Element element;

// Navigate through the node list
for (final Node node: nodeList){
if(node.getNodeName().equals("Person")) {
element = processPersonDetails(
personAttributeNames, node.getAttributes(), fragment, context,
component, customOutputHouseholdTitle);
```

Sample properties file for a summary page with a `custom-output` element

The following sample code shows an example of how to add a custom property to the properties file for the summary page that contains the `custom-output` element:

```
…
CustomHousehold.Title=Household Details:
```

Sample IEG script

In the following IEG Script, the `custom-output` element has been added before a list element on a summary page:

```
…
<summary-page id="HouseholdSummary" progress="45"
  show-back-button="true" show-exit-button="false"
  show-save-exit-button="true" show-next-button="true" set-focus="true">
  <title id="HouseholdSummary.Title"><![CDATA[Household Summary]]></title>
  <icon image="sample_title_household" />
  <custom-output
    class-name="curam.ieg.player.custom.IEGSampleCustomRenderer"
    data-accessor="curam.ieg.player.custom.IEGSampleCustomOutputDataAccessor" /
>
  <list entity="Person">
    <title id="PersonList.Title"><![CDATA[People in your household]]></title>
    <column id="firstName">
      <title id="FirstName.Title"><![CDATA[First Name]]></title>
    </column>
  </list>
</summary-page>
…
```

1.6 Integrating IEG into a Cúram Application

This section outlines how IEG can be integrated into an application. IEG can be integrated in two ways: either by opening the player in a tab or in a modal dialog. The integration tasks that are dealt with here include creating the script execution; setting finish and quit pages; running in a tab; running in a modal; cleaning up application data; and resuming scripts.

Creating a Script Execution

It is recommended that, before opening the IEG Player from an application, the script execution is created using the public API. The execution ID can then be passed to the player.

The following example code snippet shows the creation of a script execution using the public API:

```
...

// create the script execution
final IEGRuntime runtimeAPI = new IEGRuntime();
final IEGScriptExecutionIdentifier executionIdentifier =
  runtimeAPI.createScriptExecution(iegScriptID, schemaName);
```

Figure 42: Creation of a script execution

Specifying a Redirection URL

The `finish-page` and `quit-page` attributes in an IEG Script indicate what URL to redirect to when leaving the IEG Player. In this way they provide a connection between the IEG Player and an application.

These attributes are detailed in the *`IEG Script Element Reference`* chapter of the *`Authoring Scripts using Intelligent Evidence Gathering (IEG)`* developer's guide.

Modify the example script to include these attributes as shown below:

```
<ieg-script
  finish-page="IEG2_listAllIEG2Scripts"
  quit-page="IEG2_listAllIEG2Scripts"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="ieg-schema.xsd">
    ...
<ieg-script>
```

Figure 43: Script with finish-page and quit-page defined

In the example above, completion or exit from the script will result in redirection to the list of all IEG scripts provided in the administration screens.

Running the IEG Player in a Tab

Running the IEG Player in a tab requires less effort than running it in a modal. It necessitates that the 'opening' link points to `ieg/Screening.do` and passes in the `executionID`. `Screening.do` invokes the IEG Player.

The parameters are as follows:

Here is an example of a resolve UIM that opens the IEG Player in a tab:

```
<?xml version="1.0" encoding="UTF-8"?>
  <PAGE PAGE_ID="System_IEGResolver">
    <JSP_SCRIPTLET>
    <![CDATA[

      String scriptID = request.getParameter("scriptID");
      String scriptType = request.getParameter("scriptType");
      String scriptVersion = request.getParameter(
          "scriptVersion");
      String schemaName = request.getParameter("schemaName");
      String name = request.getParameter("name");

      String executionIDParam =
          request.getParameter("executionIDParam");
      String url = null;

      curam.omega3.request.RequestHandler
          rh = curam.omega3.request.
      RequestHandlerFactory.getRequestHandler(request);

      String context = request.getContextPath() + "/";

      if (executionIDParam == null) {
```

```
        // Need to check to see if there are any script
validation
        // errors before running the script.

        String contextWithUserPreferences = context +
          curam.omega3.user.UserPreferencesFactory
          .getUserPreferences(
              pageContext.getSession()).getLocale() + "/";

        curam.interfaces.IEGScriptAdminPkg.
        IEGScriptAdmin_checkForScriptErrors_TH
            iegScriptAdminCheckForErrors
                = new curam.interfaces.IEGScriptAdminPkg.
        IEGScriptAdmin_checkForScriptErrors_TH();

        iegScriptAdminCheckForErrors.setFieldValue(
        iegScriptAdminCheckForErrors.key$scriptID_idx, scriptID);
        iegScriptAdminCheckForErrors.setFieldValue(
        iegScriptAdminCheckForErrors.key$scriptType_idx,
            scriptType);
        iegScriptAdminCheckForErrors.setFieldValue(
        iegScriptAdminCheckForErrors.key$scriptVersion_idx,
            scriptVersion);
        iegScriptAdminCheckForErrors.setFieldValue(
        iegScriptAdminCheckForErrors.key$schemaName_idx,
            schemaName);
        //Call the method.
        iegScriptAdminCheckForErrors.callServer();

        String errorsPresentInScript =
            iegScriptAdminCheckForErrors.getFieldValue(
                iegScriptAdminCheckForErrors
                    .result$errorsExist_idx);
        boolean errorsPresent =

Boolean.valueOf(errorsPresentInScript).booleanValue();

        if (errorsPresent) {

          // If there are errors, redirect to the validation
error
          // page.
          String redirectTo = contextWithUserPreferences
          + "System_listValidationErrorsForRunPage.do"
          + "?name=" + name
          + "&scriptID=" + scriptID
          + "&scriptType=" + scriptType
          + "&scriptVersion=" + scriptVersion
          + "&schemaName=" + schemaName;
          url = redirectTo + "&amp;" + rh.getSystemParameters();


        } else {

          // Call the run script method and redirect to the IEG
          // player.
          curam.interfaces.IEGScriptAdminPkg.
          IEGScriptAdmin_runScript_TH iegScriptAdminRunScript
```

```
            = new curam.interfaces.IEGScriptAdminPkg.
            IEGScriptAdmin_runScript_TH();

            iegScriptAdminRunScript.setFieldValue(
            iegScriptAdminRunScript.key$dtls$scriptID_idx,
                scriptID);
            iegScriptAdminRunScript.setFieldValue(
            iegScriptAdminRunScript.key$dtls$scriptType_idx,
                scriptType);
            iegScriptAdminRunScript.setFieldValue(
            iegScriptAdminRunScript.key$dtls$scriptVersion_idx,
                scriptVersion);
            iegScriptAdminRunScript.setFieldValue(
            iegScriptAdminRunScript.key$schemaName_idx,
                schemaName);
            //Call the method.
            iegScriptAdminRunScript.callServer();

            String executionID = iegScriptAdminRunScript.
                getFieldValue(
                    iegScriptAdminRunScript.result
$executionID_idx);
            url = context + "ieg/Screening.do?" + "executionID="
                + executionID + "&" + rh.getSystemParameters();

          }
       } else {

            url = context + "ieg/Screening.do?" + "executionID="
                + executionIDParam + "&"
                + rh.getSystemParameters();

      }

      // Redirect to the correct page.
      response.sendRedirect(response.encodeRedirectURL(url));
  ]]>
  </JSP_SCRIPTLET>
</PAGE>
```

Figure 44: Resolve UIM to open IEG Player

Running the IEG Player in a Modal Dialog

The IEG Player can be opened in a modal dialog, and there are some specific considerations a script developer needs to account for pertaining to this.

Opening the IEG Player in a Modal Dialog

To open the IEG Player in a modal dialog, open `Screening.do`, in the modal, passing the `executionID` and system parameters, using a resolve UIM.

`System_IEGResolverModal.uim` is provided out-of-the-box to perform this processing:

```
<PAGE PAGE_ID="System_IEGResolverModal">
          <JSP_SCRIPTLET>
    <![CDATA[
```

```
String scriptID = request.getParameter("scriptID");
String scriptType = request.getParameter("scriptType");
String scriptVersion =
  request.getParameter("scriptVersion");
String schemaName = request.getParameter("schemaName");
String name = request.getParameter("name");

// Need to check to see if there are any script
// validation errors before running the script.
curam.omega3.request.RequestHandler
  rh = curam.omega3.request.
    RequestHandlerFactory.getRequestHandler(request);

String context = request.getContextPath() + "/";
String contextWithUserPreferences = context +
 curam.omega3.user.UserPreferencesFactory
    .getUserPreferences(
        pageContext.getSession()).getLocale() + "/";

String url = null;

curam.interfaces.IEGScriptAdminPkg.
  IEGScriptAdmin_checkForScriptErrors_TH
      iegScriptAdminCheckForErrors
    = new curam.interfaces.IEGScriptAdminPkg.
      IEGScriptAdmin_checkForScriptErrors_TH();

iegScriptAdminCheckForErrors.setFieldValue(
  iegScriptAdminCheckForErrors.key$scriptID_idx,
      scriptID);
iegScriptAdminCheckForErrors.setFieldValue(
  iegScriptAdminCheckForErrors.key$scriptType_idx,
      scriptType);
iegScriptAdminCheckForErrors.setFieldValue(
  iegScriptAdminCheckForErrors.key$scriptVersion_idx,
      scriptVersion);
iegScriptAdminCheckForErrors.setFieldValue(
  iegScriptAdminCheckForErrors.key$schemaName_idx,
      schemaName);
//Call the method.
iegScriptAdminCheckForErrors.callServer();

String errorsPresentInScript =
   iegScriptAdminCheckForErrors.getFieldValue(
     iegScriptAdminCheckForErrors.result$errorsExist_idx);
boolean errorsPresent =
    Boolean.valueOf(errorsPresentInScript).
        booleanValue();

if (errorsPresent) {

  // If there are errors, redirect to the validation
  // error page.
  String redirectTo = contextWithUserPreferences
    + "System_listValidationErrorsForModalPage.do"
    + "?name=" + name + "&scriptID=" + scriptID
          + "&scriptType=" + scriptType
    + "&scriptVersion=" + scriptVersion
```

```
            + "&schemaName=" + schemaName;
          url = redirectTo + "&&" + rh.getSystemParameters();


        } else {

          // Call the run script method and redirect to
          // the IEG player.
          curam.interfaces.IEGScriptAdminPkg.
            IEGScriptAdmin_runScript_TH iegScriptAdminRunScript
              = new curam.interfaces.IEGScriptAdminPkg.
                IEGScriptAdmin_runScript_TH();

          iegScriptAdminRunScript.setFieldValue(
            iegScriptAdminRunScript.key$dtls$scriptID_idx,
                scriptID);
                iegScriptAdminRunScript.setFieldValue(
            iegScriptAdminRunScript.key$dtls$scriptType_idx,
                scriptType);
          iegScriptAdminRunScript.setFieldValue(
            iegScriptAdminRunScript.key$dtls$scriptVersion_idx,
                scriptVersion);
          iegScriptAdminRunScript.setFieldValue(
            iegScriptAdminRunScript.key$schemaName_idx,
                schemaName);
          //Call the method.
          iegScriptAdminRunScript.callServer();

          String executionID =
              iegScriptAdminRunScript.getFieldValue(
            iegScriptAdminRunScript.result$executionID_idx);
          executionID = executionID.replaceAll(",", "");

          url = context + "ieg/Screening.do?"
                + "executionID=" + executionID
                + "&" + rh.getSystemParameters();
        }

        // Redirect to the correct page.
        response.sendRedirect(
            response.encodeRedirectURL(url));
      ]]>
    </JSP_SCRIPTLET>
</PAGE>
```

Exiting a Script Execution in a Modal Dialog

There are two broad approaches a script developer can take to complete or exit an IEG script execution in a modal dialog:

- Directly closing the modal dialog, and refresh or redirect in the parent tab.
- Progressing to further UIM screen/s in the modal dialog.

Directly Closing the Modal on Script Completion

To close a modal dialog directly upon completion of; or exit (Exit, Save & Exit actions) from an IEG script execution, the script developer must specify a resolve UIM as the finish-page and/or quit-page.

That resolve UIM must in turn invoke a custom JSP that calls the appropriate JavaScript function to close the dialog.

For example, to redirect to the `IEG2_listAllIEG2Scripts` administration screen, include the following JSP scriptlet in your UIM file:

```
<PAGE
   PAGE_ID="IEG2_resolveFinishScript"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="file://Curam/UIMSchema.xsd"
 >
 <JSP_SCRIPTLET>
   <![CDATA[

     curam.omega3.request.RequestHandler
       rh = curam.omega3.request.RequestHandlerFactory
           .getRequestHandler(request);

     String context = request.getContextPath() + "/";
     context += curam.omega3.user.UserPreferencesFactory
         .getUserPreferences(
           pageContext.getSession()).getLocale() + "/";

     String url = "";
     url = context + "IEG2_listAllIEG2ScriptsPage.do";

     String forwardParams =
         request.getParameter("forwardParams");

     if (screenContext != null && screenContext
         .hasContextBits(
         curam.omega3.taglib.ScreenContext.MODAL)) {
       url += "?" +  rh.getSystemParameters();
       String encodeRedirectURL = response.encodeURL(url);
       response.sendRedirect(response.encodeRedirectURL(
           request.getContextPath() +
               "/ieg/CloseAndRedirect.jspx?redirect="
               + encodeRedirectURL));
     } else {
       url += "?" +  rh.getSystemParameters();
       response.sendRedirect(
           response.encodeRedirectURL(url));
     }

     ]]>
 </JSP_SCRIPTLET>
</PAGE>
```

`CloseAndRedirect.jspx` is provided out-of-the-box for closing the modal dialog and redirecting to a specified UIM (if provided) in the parent.

Progressing to Further UIM Screen/s in the Modal Dialog

To keep the modal dialog open to display further UIM screens after script execution has completed, specify the required UIM page as the finish-page and/or quit-page in the IEG script definition.

Once that UIM has loaded, you have moved out of IEG and standard UIM processing in a modal dialog applies.

Cleaning Up Application Data

Cleaning up application data involves removing data from the IEGEXECUTIONSTATE database table and the Datastore(DS) where appropriate. This section details the manual and automatic data clean-up tasks that script authors should be aware of, and makes some recommendations to ensure cleaning up application data can proceed smoothly.

In order to support execution of an IEG script, information about individual script executions must be maintained by the IEG engine. For example the IEG engine must keep track of the current page for the script execution. The IEG engine must also maintain a list of the pages that have been presented to the user to support navigation. The answers to control questions are not persisted in the DS and the IEG engine must also keep track of these. All the information to support the execution of an IEG script is persisted in the IEGEXECUTIONSTATE table. When a new IEGScriptExecution object is created via the IEGScriptExecutionFactory API a corresponding entry is created in the IEGEXECUTIONSTATE table. The IEGEXECUTIONSTATE table is an "internal" table only intended to be used by the IEG engine and it should not be modified or extended. Script authors should not become dependent on or make assumptions about the contents of this table as they can be subject to change without notice.

IEG has no way of knowing when an entry in the IEGEXECUTIONSTATE table is no longer required and therefore the entries will persist until they are explicitly deleted. To avoid the IEGEXECUTIONSTATE table becoming unnecessarily cluttered, if a script execution has completed or will not be resumed or re-executed its entry in the table should be removed via the removeScriptExecutionObject method of the IEGScriptExecutionFactory API.

IEG cannot make any inference as to what data can be used to logically and uniquely identify a particular script execution as this can vary from script definition to script definition. The only way for IEG to identify a script execution is via the generated ID that is assigned to the script execution when it is initially created. It is highly recommended that script authors implement a mechanism to identify script executions by associating unique data with the script execution IDs. A new table can be created to maintain the relationship between the data that identifies the execution and the execution ID to make it easy for script executions to be resumed. When they are no longer required they can be removed. Removing a script execution does not cause any of the gathered data that is persisted in the DS to be removed.

Similarly to IEGEXECUTIONSTATE, the IEG engine and the DS have no way of knowing when the data that is gathered during a script execution and persisted in the DS is no longer required. Again, the DS can become unnecessarily cluttered with entities that are no longer required. It is intended that entities will not persist in the DS indefinitely but that the gathered data be moved to application tables and then removed from the DS. When an entity is deleted from the DS its child entities are also deleted. Therefore when the data that is gathered during a script execution has been moved to application tables and is no longer required it is sufficient to delete the root entity for the execution.

The following example code snippet shows the deletion of the root entity:

```
final Long applicationID = execution.getRootEntityID();
      final Entity rootEntity =
 datastore.readEntity(applicationID);
      rootEntity.delete();
```

Figure 45: Deleting the Root Entity

Resuming Executed Scripts

It is possible to stop a script execution and resume it later. To do so, the application must take care of storing the execution ID in a custom table and associate it with some user ID.

See Cleaning Up Application Data on page 60 for more details.

Provided the IEGEXECUTIONSTATE table hasn't been cleaned up and the script definition hasn't been modified, a script execution will be resumed by passing the executionID parameter to the IEG Player in the same way it is done when starting a new script execution.

1.7 Managing Captured Data

The data captured during script execution is persisted in the Datastore (DS). This section explains how you can retrieve the captured data from the DS. This section also explains how data can be inserted into the DS so that it is available to IEG while executing scripts.

Retrieving Captured Data

The Datastore (DS) has a public API which you may use in your application code. This API is most often used to retrieve information from a populated schema but it can also be used to pre-populate a schema.

For example, once a client has completed an application, they can submit their information. At this point, the API can be used to extract the data from the schema and populate tables in the relational database.

An example of pre-population is where some information about the client is known in advance of starting their application. If some of that information is required to navigate through the application, the DS can be pre-populated with the information.

To read any data from a schema, the appropriate execution of the script needs to be known. This means you are retrieving the correct application information for a client. Therefore, the executionID and schema name are vitally important to gain access to the data.

The following example code snippet shows the obtaining of the root entity:

```
final IEGRuntime runtimeAPI = new IEGRuntime();
final IEGRootEntityID rootEntityID =
  runtimeAPI.getScriptExecutionRootEntityID(executionID);

Datastore ds = DatastoreFactory.newInstance()
  .openDatastore(kSchemaName);

final Entity rootEntity =
```

```
  ds.readEntity(rootEntityID.entityID));
```

Figure 46: Obtaining root entity

From here, the root entity can be used to retrieve other entities under this root entity.

Pre-Populating Scripts with Captured Data

It is possible to pre-populate the values that will be displayed to the user so that the answers only need to be confirmed or modified.

For example, we can pre-populate the name and date of birth of a user on a Personal Details page assuming that the user has already logged in and another database holds the personal details.

The DS can be pre-populated prior to the start of script execution as follows:

```
...
Datastore ds = null;

try {
  // open the data store and create the root entity
  ds = DatastoreFactory.newInstance().openDatastore(schemaName);
} catch (NoSuchSchemaException e) {
  throw new AppException(IEG.ID_SCHEMA_NOT_FOUND);
}

final EntityType appType = ds.getEntityType("Application");
final Entity rootElement = ds.newEntity(appType);

ds.addRootEntity(rootElement);

final EntityType personType = ds.getEntityType("Person");
final Entity person = ds.newEntity(personType);

person.setAttribute("firstName", "TestFirstName");
person.setAttribute("lastName", "TestLastName");
person.setAttribute("dateOfBirth", "19700101");
//...

rootElement.addChildEntity(person);
```

Figure 47: Code Snippet that Populates the DS

The root entity can then be used in creating a new script execution as follows:

```
...

// create the script execution
final IEGRootEntityID rootEntityID = new IEGRootEntityID();
rootEntityID = rootElement.getUniqueID();
final IEGRuntime runtimeAPI = new IEGRuntime();
final IEGScriptExecutionIdentifier executionIdentifier =
  runtimeAPI.createScriptExecutionExistingRootEntity(
    iegScriptID, schemaName, rootEntityID);
```

Figure 48: Creation of a Script Execution

The IEG Player can then be run using this new script execution as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<PAGE PAGE_ID="IEGScriptLauncher">
  <JSP_SCRIPTLET>
    <![CDATA[
curam.omega3.request.RequestHandler rh =
  curam.omega3.request.RequestHandlerFactory.getRequestHandler(
    request);

String context = request.getContextPath() + "/";

String url =
  context + "ieg/Screening.do?" + "executionID=" + executionID
    + "&" + rh.getSystemParameters();

// Redirect to the correct page.
response.sendRedirect(response.encodeRedirectURL(url));
    ]]>
  </JSP_SCRIPTLET>
</PAGE>
```

Figure 49: Launching the IEG Player

Note that it is only possible to pre-populate the DS, and not the control questions or other script-related information as they are stored in the script execution and not in the DS. This means that it is not possible to pre-populate the data displayed in the first section of the script and start at the second section. The first section will be displayed and the user will be able to confirm the pre-populated data.

1.8 Using the Resource Store

The Resource Store is an area of the infrastructure database which is used to store resources used in a live application. Resources can be of any type but the most common used by IEG are images and properties file resources.

Listing all Resources

To gain access to the resource administration screens, you will need to log in as an admin user. Once logged in, you will see a section in your navigation panel called IEG. When you click on the section, you will see a menu which contains a link called 'Application Resources'. If you click on this, a list of resource will be displayed with a search box based on the category.

Resources are organized into categories. Existing resources are displayed by selecting a category in the filter criteria and selecting 'search'. The resource categories used by IEG are as follows:

- CSS

  Stylesheet templates that can be modified to customize the look-and-feel of the IEG Player.
- Image

  Images used in the IEG Player and IEG Scripts.
- Property

  Properties files containing locale specific text for Scripts and Question Pages.

Uploading a New Resource

At the top of the screen which lists all resources is a link which lets you add a new resource. When you click on this link, you will be presented with a screen where the resource details should be entered.

You must enter the following information:

- Name

  This is a unique name for the resource which can be used within an IEG script to reference it. Depending on the resource type, a naming convention may be enforced for use within an IEG script. The sections on Adding Images on page 65 and Changing Static Text on page 65 have more details.
- Content Type

  When serving a resource to a web browser, a content type is required to instruct the browser how to handle the resource. The most common content types used in an IEG script would be `image/png` for a PNG image and `text/plain` for a properties file.
- Content

  The file chooser allows the user to pick the resource to upload.

The following information is optional:

- Category

  The category in which the new resource is to be added.
- Content Disposition

  For resources used in IEG scripts, this can be left empty.
- Locale

  If you wish to have a locale specific version of a resource, enter the locale code here. When the system searches for a resource, it uses a fall-back mechanism similar to Java. For example, if the current locale is `en_US` the system will attempt to locate the resource for the `en_US` locale, then `en` and finally the “default” resource. The “default” resource is specified by leaving the locale field empty when uploading the resource.
- Internal

  This indicates if the resource is for internal use only and should never be served to the web browser. In this first release of IEG, this setting can be ignored.
- Description

  A description of the resource.

Removing an Existing Resource

To delete an existing resource, select the 'view' link on the resource and from the 'View Resource Page' select 'delete' to remove this resource from the system.

When you click on this link, a confirmation dialog will be displayed asking you to confirm that you want to remove this resource from the system.

Updating an Existing Resource

To update an existing resource, select the 'edit' link on the 'Application Resources' page or on the 'View Resource' page. You can then browse to the updated resource on your file system in the 'New content' field.

Downloading an Existing Resource

Each entry in the resources list can be downloaded by clicking the 'download' link on the 'Application Resources' page. This link will open the browser file download dialog to allow the user to save the resource or open it directly.

Adding Images

IEG scripts allow you to specify images to use for both your sections (in the navigation panel to the left of a page, by default) and pages (in the page title area for the page), and also comes with some images which are built in to the system (like the various person images used in person tabs, and so on).

All of these images must be stored in the resource store so that new images can be added and existing ones updated without having to rebuild and re-deploy your application. When uploading an image resource, set the “Content Type” appropriately for the image (e.g., image/png, image/gif etc.) and leave the “Content Disposition” field empty.

Changing Static Text

The IEG engine allows you to enter all the text for your script for the default locale directly into the script definition. However, this is not where the text displayed on the screens is actually read from. Instead, all text referenced from within a script is stored in locale-specific properties files within the resource store.

For each script, there will be a minimum of one properties file for the script itself and one properties file for each page within the script. In order to ensure the uniqueness of these files, the following naming convention is used (the last part is obviously only applicable to the page-specific properties files):

```
scriptID_scriptVersion_scriptType_pageID
```

When you use the IEG admin screens to upload a new script into the system, all the static text contained within it (e.g., all the labels, titles, descriptions, etc.) are automatically extracted into the appropriately named properties files for your script and stored in the resource store with no locale associated with them (so that they act as the fall-back properties if no properties exist for the locale in which you are running). Any of this text can then be changed by simply downloading the current properties file keeping in mind the naming convention described above to locate the resource in the resources list. Then make the necessary changes and update the resource as described in Updating an Existing Resource on page 65. No changes to the script itself are required.

Equally, versions of these files for other locales can be easily added and will be picked up in preference to the default locale properties the next time the script is run in that locale. When

uploading a properties file resource, set the “Content Type” to `text/plain` and leave the “Content Disposition” field empty.

Changing the Default File Encoding

When uploading a plain text resource, the file will be expected to be in UTF-8 encoding. If you wish to use a different encoding when uploading the file, the "Content Type" field can be used to specify this through the use of the optional `charset` parameter.

For example:

```
text/plain; charset=ISO-8859-1
```

1.9 Using IBM Rational AppScan to scan IEG

This section describes the steps required to perform security scans of IEG style applications using the IBM®Rational AppScan® tool.

Preparation

In IEG the communication between the Player and the Engine is coordinated by means of a sync token. The sync token is used to ensure that the page submitted by the browser is consistent with the page the IEG engine is expecting to be submitted.

This facilitates detecting when the user uses the browser navigation buttons rather than the navigation buttons in the Player itself. The sync token changes for every question page that is displayed in the IEG Player. This makes it very difficult to scan IEG question scripts executing in the IEG player.

For this reason, prior to scanning it is recommended that the script configuration property appscan.mode.enabled should be set to true. When this property is set to true, the Engine does not check the value o the sync token that is passed by the Player. Disabling sync token checking is acceptable when performing a scan but sync token checking should always be enabled in a production environment.

Also, in order to reduce the amount of superfluous information reported in a scan the stack trace should be disabled. To disable the stack trace:

- Go to the folder webclient\JavaSource\curam\omega3\
- Rename Initial_ApplicationConfiguration.properties to ApplicationConfiguration.properties
- Open ApplicationConfiguration.properties
- Add the entry: `errorpage.stacktrace.output=false`

Relationship Pages

Relationship pages are a special feature of IEG which facilitate gathering information about the relationships between the people of a household.

Unlike the other pages of an IEG script Relationship pages have a more dynamic nature and contain a variable number of fields. Currently the names of the fields that are generated for

Relationship pages vary from execution to execution. This means that currently it is not possible to run a scan on an IEG question script that contains a relationship page.

Scan Configuration

Once AppScan is launched a new scan can be created by selecting the 'Create New Scan...' option on the Welcome screen.

Then select 'Regular Scan' from the Predefined Templates on the next screen.

Choose 'Web Application Scan' on the first page of the Configuration Wizard, click 'Next'.

On the 'URL and Servers' page of the wizard enter the starting URL of the application. Once the URL is entered it can be verified by clicking the icon beside the input field. This will cause the AppScan browser to be displayed and it will attempt to open the URL. Confirm that the URL is correct and accessible. (Click yes on security warning if necessary). Close the browser and click 'Next' in the Configuration Wizard.

Enter the necessary Login Management details. Applications running under Eclipse/Tomcat do not require the user to login, so the option 'None' can be selected. Click 'Next'.

On the 'Test Policy' screen click on the 'Full Scan Configuration' link in the 'General Tasks' panel. This presents the 'Scan Configuration' dialog.

Test Policy

Ensure that 'Test Policy' is selected in the view selection pane on the left-hand side of the configuration dialog. The most straight forward approach while configuring a scan is to enable all the tests and then disable the low value tests which increase the time required to run the scan.

Select 'Enabled/Disabled' from the 'sort tests by' dropdown. First check the 'Partially Enabled' then the 'Disabled' boxes. The only entry displayed should be 'Enabled'. Select 'Severity' from the dropdown. Uncheck the 'Low' and 'Informational' boxes. For the purposes of scanning IEG it is not required to perform invasive tests as these tests are more concerned with testing the platform. Select 'Invasive' from the dropdown. Uncheck the 'Invasive' box.

Explore Options

Select 'Explore Options' in the view selection pane.

Set 'Redundant Path Limit' to 1. Choose 'Breadth First' as the 'Explore Method'.

Communications and Proxy

Select 'Communications and Proxy' in the view selection pane. Set 'Number of Threads' to 1.

Test Options

Select 'Test Options' in the view selection pane. Uncheck 'Use Adaptive Testing based on application behavior'.

Multi-Step Operations

IEG requires correctly formatted data be used in certain parameters. As such AppScan must be 'trained' to use the application being tested. Select 'Multi-Step Operations' in the view selection pane.

Click the record button. This will cause the AppScan browser to be displayed and it will attempt to open the URL specified on the 'URL and Servers' page of the Configuration Wizard. You should then navigate through the application as required, entering relevant data. AppScan will record the values entered and use these values for each test that it runs later. Once you have finished, simply close the browser. The Scan configuration dialog will be updated with the sequence that has just been recorded. Check the 'Enable playback of this sequence' checkbox and uncheck the 'Allow play optimization' checkbox.

Take note of all the sequence steps that contain `Screening.do`. You will have to turn these sequence steps into regular expressions and add them as exception paths to the exclude path options of AppScan. AppScan can easily be thrown out of sync when it comes to recorded operations, so you have to ensure that AppScan will ignore the wrong path and keep to the recorded script when running its tests. This is achieved by telling AppScan to ignore all sequence steps containing `screening.do`, except those that you specify in regular expressions. Take note of each `__u=x` value found in the list of sequence steps.

Exclude Paths and Files

Select 'Exclude Paths and Files' in the view selection pane.

Click the button to add an Exclude Path. Choose 'Exclude' as the 'Type' and select 'Regular Expression' from the 'Match' dropdown. Enter `.*/Curam/ieg/Screening.do.*` for the 'Path' and click 'OK'.

Add another Exclude Path. Choose 'Exception' as the 'Type' and select 'Regular Expression' from the 'Match' dropdown. Enter `.*/Curam/ieg/Screening.do?executionID=.\d*` for the 'Path' and click 'OK'.

An Exception should also be added for each `__u=x` value found in the list of sequence steps. Again, select 'Regular Expression' from the 'Match' dropdown and enter an expression in the following format for 'Path': `.*/Curam/ieg/Screening.do?executionID=.*&__u=[value shown in summary screen]`

Click 'OK'.

Click 'OK' to be returned to the configuration wizard.

Click 'Next' in the Configuration Wizard.

Complete

At this point the configuration of the scan is complete. Choose the 'I will start my scan later' option so that the configured scan will be saved rather than allowing AppScan randomly scan the whole application. Click 'Finish'.

Running the Scan

To start the scan, select the 'Scan' menu item in the AppScan main window and select 'Test Multi-Step Operations Only'.

Depending on the application to be tested a scan may take a number of days to complete. Once the scan is complete AppScan will display the results of the scan on a summary screen. These results should then be investigated to determine which reported issues are real vulnerabilities and which are false positives.

1.10 Runtime processing in IEG

This section describes the runtime processing and behaviors that occur when you use an IEG application.

Loss of network connectivity during an IEG session

When a loss of network connectivity occurs while you are running an IEG application, if a user starts an operation that requires a server call, such as clicking a navigational button or clicking a link that performs an action, the user is presented with a modal dialog box, that informs them of a network connectivity error.

By default, the modal dialog box informs the user that a network error occurred while the request was being processed and to check the internet connection and try again. This default text within the modal dialog box is configured by using the following properties:

Property	Purpose
network.connection.error.dialog.title.text	Modal dialog title text
network.connection.error.dialog.message.text	Modal dialog message
network.connection.error.dialog.ok.button.test	Modal dialog OK button text

The modal dialog box displays at the center of the user's application window and any access to the page content behind the modal is disabled.

The modal dialog box also contains an **OK** button which users are able to click to close it. The users can then continue to interact with the IEG features that do not require a server call to function, for example, form data, widgets such as drop-down menus and date pickers, help links and expand or collapse clusters and lists on summary pages.

When the modal dialog box is dismissed by the user, all navigation actions and links are reset and respond as before. No limit exists for the amount of times that the modal dialog box can be shown and dismissed while in an offline state.

IEG features that require a server call to function, such as code table hierarchies, do not function in an offline state. When the connection is restored, the IEG features work as before. The modal dialog box is not displayed when a user clicks a hyperlink that brings them to an external source or for non-navigation actions such as clicking a print link.

When the network connection is restored and the network connectivity error modal dialog box is dismissed, users are able to start operations that require server calls as normal without the modal dialog opening, and features such as form validations resume seamlessly. All keyboard shortcuts and gesture navigation operate as normal and respond gracefully when any network resources are requested.

For a session-timeout warning dialog box that is configured to be displayed before an application session times out, and if the IEG application is in an offline state when the session-timeout warning dialog box displays, the session-timeout warning dialog box contains no content and is not dismissable. The reason is that the session-timeout warning dialog box requires a server call to populate its content. If the session-timeout warning dialog box displays while IEG is offline, a user needs to refresh the browser when the network connection is restored to restart the session.

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.