Oracle® Real-Time Decisions

Platform Developer's Guide

Version 3.0.0.1

E13854-02

April 2011

ORACLE®

Oracle Real-Time Decisions Platform Developer's Guide, Version 3.0.0.1

E13854-02

Contents

9 Using the .NET Smart Client

10 Web Service Client Example

11 Using the Oracle RTD PHP Client

Part III Decision Studio Reference

12 About Decision Studio

13 About Decision Studio Elements and APIs

17 Externalized Objects Management

Preface

Oracle Real-Time Decisions (Oracle RTD) enables you to develop adaptive enterprise software solutions. These adaptive solutions continuously learn from business process transactions while they execute and optimize each transaction, in real time, by way of rules and predictive models.

This document is divided into four parts:

- Part I, "Getting Started" is a tutorial that provides information and examples to help you get started using Oracle RTD.
- Part II, "Integration with Oracle RTD" provides information about integrating with Oracle RTD, including information about Oracle RTD Smart Clients, Web services, and directly messaging Real-Time Decision Server.
- Part III, "Decision Studio Reference" identifies each of the elements used to configure Inline Services, including the properties of each Inline Service and the available APIs.
- Part IV, "Miscellaneous Application Development" provides an in-depth look at the concepts, components, and processes involved in Oracle RTD application development that require special processing, such as batch framework and external editors that enable modification of Oracle RTD application objects.

Audience

Part I of this document is designed to help technical users of Oracle RTD get acquainted with the capabilities, terminology, tools, and methodologies used to configure Inline Services.

Part II of this document is intended for developers who will use the Java-based API, .NET component, or Web services to integrate enterprise applications with Oracle RTD Inline Services.

Part III of this document is designed for technical users configuring Inline Services using Decision Studio.

Part IV of this document is designed for technical users who will develop batch jobs or external editors that interface with Oracle RTD application objects.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to

facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at http://www.oracle.com/accessibility/.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

To reach AT&T Customer Assistants, dial 711 or 1.800.855.2880. An AT&T Customer Assistant will relay information between the customer and Oracle Support Services at 1.800.223.1711. Complete instructions for using the AT&T relay services are available at http://www.consumer.att.com/relay/tty/standard2.html. After the AT&T Customer Assistant contacts Oracle Support Services, an Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process.

Related Documents

For more information, see the following documents in the Oracle RTD Version 3.0 documentation set:

- *Oracle Real-Time Decisions Release Notes*
- *Oracle Real-Time Decisions Installation and Administration Guide*
- *Oracle Real-Time Decisions Decison Center User's Guide*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I

Getting Started

The chapters in Part I are a tutorial that provide an introduction to using Oracle RTD. The examples in these chapters assume the reader has installed Oracle RTD on a Windows system.

Part I contains the following chapters:

1

About Oracle Real-Time Decisions

Oracle Real-Time Decisions (Oracle RTD) provides a new generation of enterprise analytics software solutions that enable companies to make better decisions in real time at key, high-value points in operational business processes.

Oracle RTD easily integrates with enterprise applications both on the front end (such as CRM applications) and on the back end (such as enterprise data stores). Oracle RTD also includes other helpful load testing and debugging tools.

This chapter contains the following topics:

1.1 Terminology

Oracle RTD consists of five components:

- Decision Studio
- Real-Time Decision Server
- Decision Center
- Administration (JMX)
- Load Generator

Inline Service refers to the configured application that is deployed.

Inline Services are configured and deployed using Decision Studio and analyzed and updated using Decision Center. Inline Services run on Real-Time Decision Server.

An Inline Service can gather data and analyze characteristics of enterprise business processes on a real-time and continuous basis. It also leverages that data and analysis to provide decision-making capability and feedback to key business processes.

Elements are one of the following types of objects:

- **Application:** The application object identifies application level settings including default model parameters, and any parameters needed for the Inline Service globally.
- **Performance Goals:** Performance Goals identify the Key Performance Indicators (KPIs) used for setting the decision criteria for the scoring of choices in Oracle RTD.

- **Choices:** Choices represent the offers that will be presented through the Inline Service or the targets of study to be tracked by the self-learning models of Oracle RTD.
- **Rules:** Rules are graphically configured rules used to target segments of population, decide whether a choice is eligible or score a particular choice.
- **Decisions:** Decisions score and rank eligible choices based on the weighted scores for each associated performance goal.
- **Selection Functions:** Selection Functions can be used by Decisions as a custom way to select which choice to send back through the Oracle RTD Advisors.

  An Advisor is an Integration Point. For more information, see the topic Integration Points that follows in this list.
- **Entities:** Entities represent the actors in the system. Entities are a logical representation of data used for Oracle RTD modeling and decisioning. The attributes of an entity can be populated via data sources, as incoming parameters from integration points, or derived in real time through custom logic.
- **Data sources:** Data Sources retrieve data from tables or stored procedures.
- **Integration Points:** Integration Points serve as the touchpoints with outside systems interacting with Oracle RTD. There are two classes of Integration Points: *Informants* and *Advisors*. Informants receive data from outside systems, whereas Advisors receive data and also send recommendations back to outside systems.
- **Models:** Self-learning, predictive models that can be used for optimizing decisions and providing real-time analysis for desired targets of study.
- **Statistics Collectors:** Statistic Collectors are special models that track statistics about entities.
- **Categories:** Categories are used to segment data for display in Decision Center.

Note: The following terms are referenced throughout the Oracle RTD documentation:

- *RTD_HOME*: This is the directory into which Oracle RTD is installed. For example, `C:\OracleBI\RTD`.
- *RTD_RUNTIME_HOME*: This is the application server specific directory in which the application server runs Oracle RTD.

For more information, see the section "About the Oracle RTD Run-Time Environment" in *Oracle Real-Time Decisions Installation and Administration Guide*.

1.2 About Decision Studio

Decision Studio is the development tool for configuring Inline Services, the services that are created to model business processes, gather statistics, and make recommendations.

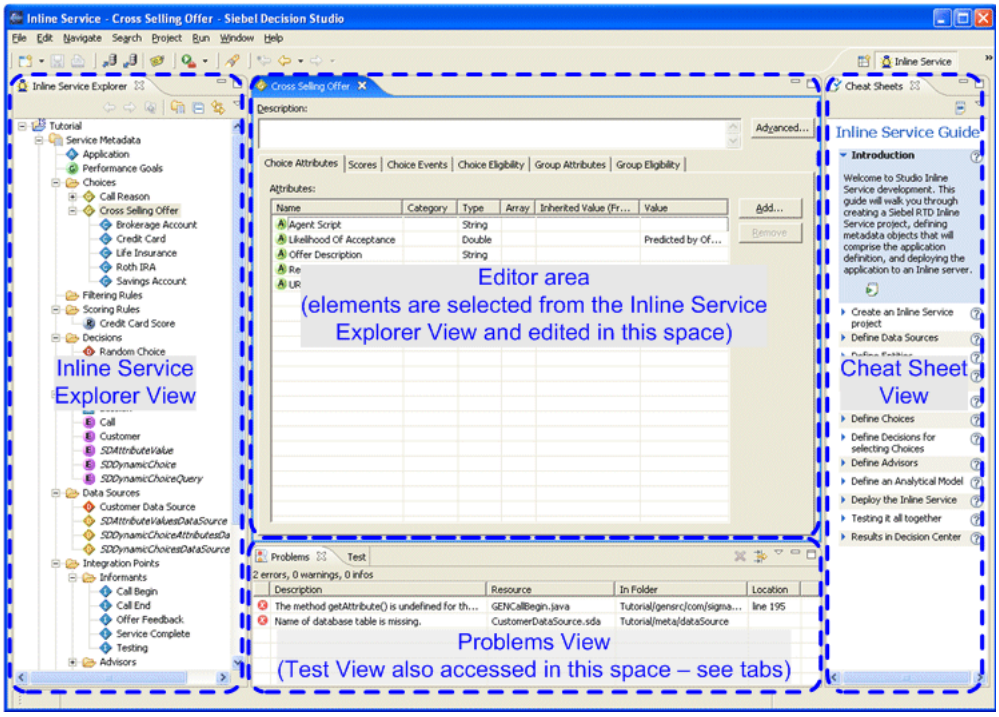
Decision Studio is fully integrated with Eclipse, an open source Java IDE produced by the Eclipse Foundation. Decision Studio exists as a standard plug-in to the Eclipse environment. If you are using Eclipse, you have the advantage of using the environment for additional development and advanced features. If you are not familiar with Eclipse, it is completely transparent to using Decision Studio. Eclipse and Decision Studio both have online help available through the Help menu.

Decision Studio allows you to work with an Inline Service from several *perspectives*. A perspective defines the initial set and layout of *views* and *editors* for the perspective. Each perspective provides a set of functionality aimed at accomplishing a specific type of task or works with specific types of resources. Perspectives control what appears in certain menus and toolbars.

To select or change to a different perspective (such as Inline Service, Java, Resource, and so on), click the **Window** menu in Decision Studio and choose **Open Perspective**, then choose from the list of available perspectives. The default perspective when starting Decision Studio for the first time is **Inline Service**. We will use this perspective in this tutorial. In general, this will be the perspective you use to develop Inline Services.

The default Inline Service perspective contains four views and an editor area, as shown in Figure 1–1.

Figure 1–1 Inline Service Perspective in Decision Studio



This section contains the following topics:

- Section 1.2.1, "Inline Service Explorer View"
- Section 1.2.2, "Problems View"
- Section 1.2.3, "Test View"
- Section 1.2.4, "Cheat Sheets View"
- Section 1.2.5, "Editor Area"
- Section 1.2.6, "Arranging Views and Resizing Editors"

1.2.1 Inline Service Explorer View

The **Inline Service Explorer View** organizes all of the elements of the Inline Service that are configured by the user.

1.2.2 Problems View

The **Problems View** shows validation (.sda files) and compilation errors (.java files) as you build and compile your Inline Service. If you double-click a validation error, Problems View opens the metadata/element-editor at the point of the error. If you double-click a compilation error, Problems View opens the generated source code (`.java` files) at the point of the error. You should *not* edit generated source code files directly; instead, fix related metadata/element problems, which will then regenerate and recompile the source code.

1.2.3 Test View

The **Test View** allows you to test your Inline Services directly from Studio after you deploy them to the server.

1.2.4 Cheat Sheets View

The **Cheat Sheets View** provides step-by-step instructions for common tasks. After installation, it is located on the right-hand side of the window.

> **Tip:** You may want to close the Cheat Sheets View to give more editor space. The Cheat Sheets are not used in this tutorial.

1.2.5 Editor Area

The center area of the Inline Service Perspective is the *editor* area, and shows an editor that is specific to the node on the project tree you have selected. To change to a new editor, double-click the element you want to edit from the Inline Service Explorer View.

1.2.6 Arranging Views and Resizing Editors

Tabs on the editors indicate the names of resources that are currently open for editing. An asterisk (*) indicates that an editor has unsaved changes.

You can drag the views and editors of a perspective to any space on the screen. Views and editors will resize themselves to fit the area in which they are placed. Occasionally, portions of an editor (where you do your main work) or view will become covered by other views, or resized to an area that is not convenient to use. To resize the editor or view, either close other open views and the remaining will automatically resize, or maximize the editor or view by double-clicking the editor tab.

Both editors and views can be toggled between Maximize and Minimize by double-clicking the tab, or by using the right-click menu item.

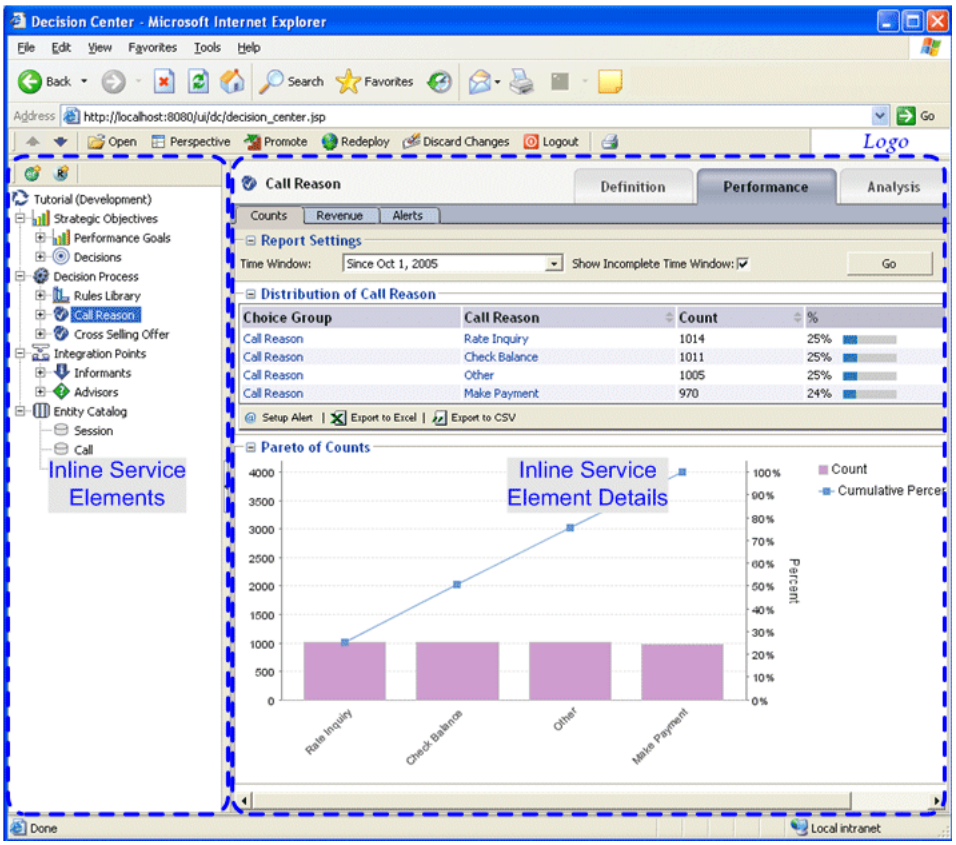
To show additional views or open views that were closed, click the **Window** menu in Decision Studio and choose **Show View**, then choose from the list of available views.

1.3 About Decision Center

Decision Center is a Web-based application that allows the business analyst to monitor and optimize deployed Inline Services. From Decision Center, you can view statistics gathered from the models and fine-tune campaigns such as cross-selling, as well as adjust how decisions are made.

The Decision Center user interface displays Inline Services in two panes. The left pane shows the list of Inline Service elements, while the right pane displays detailed information related to the selected element.

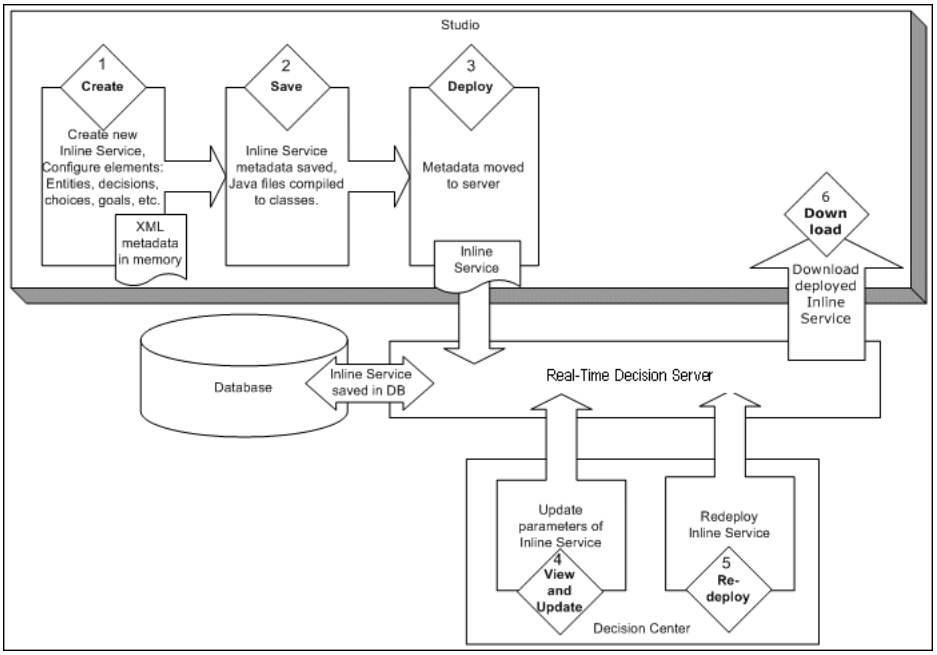
Figure 1–2 Decision Center



1.4 About the Inline Service Lifecycle

Inline Services are created using Decision Studio. Figure 1–3 shows the Inline Service Lifecycle.

Figure 1–3 Inline Service Lifecycle

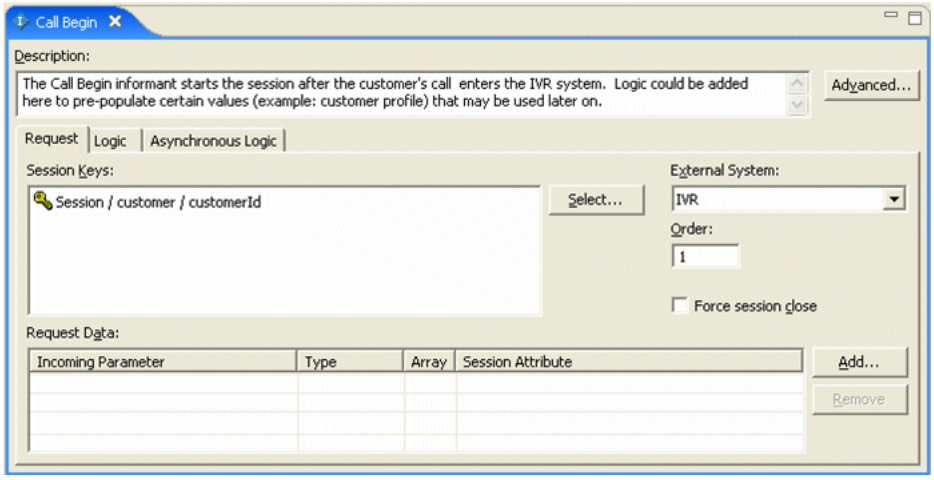


The following steps outline the overall process by which Inline Services are created, deployed, and downloaded:

1. **Create:** Using Decision Studio, elements are configured to align with the business process into which Oracle RTD is to be integrated. Examples of elements are: Choice Groups, Performance Goals, Decisions, Informants, Advisors, Entities, and Data Sources.

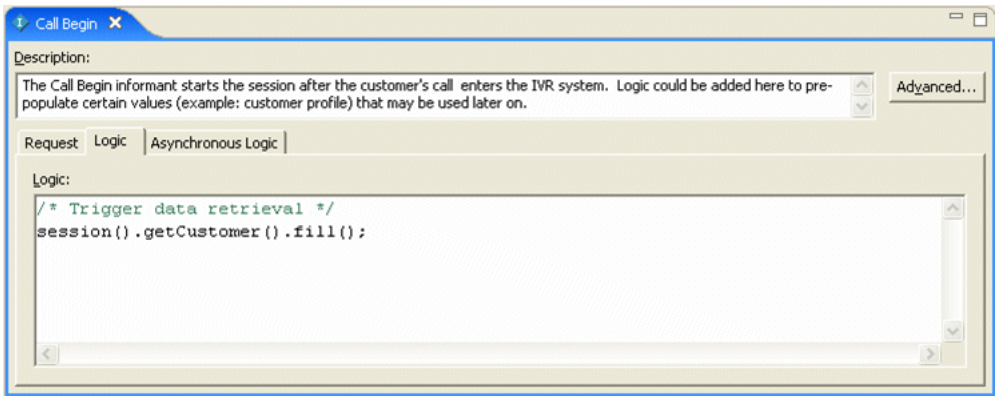
   Some elements allow the use of Java scriptlets in Logic and Asynchronous Logic attributes. For instance, an Informant element is shown in Figure 1–4. This element is named 'Call Begin.' In addition to the Description and the Advanced button, there are three tabs, each with a set of attributes for the Informant.

Figure 1–4 Call Begin Informant



In the Logic tab of this 'Call Begin' Informant, we can write optional Java code to perform specific tasks, as shown in Figure 1–5.

Figure 1–5 Logic Tab of Call Begin Informant



As elements are created and saved, XML metadata is created in memory that describes the object.

2. **Save:** By saving the Inline Service in Decision Studio, the metadata is written to an Inline Service directory on the local file system, in XML files with the extension `*.sda`. The metadata that follows is an example of the content saved for the Informant 'Call Begin', in a file called `CallBegin.sda`.

```
<?xml version="1.0" encoding="UTF-8"?>
<sda:RTAPType xmlns:sda="http://www.sigmadynamics.com/schema/sda"
internalName="CallBegin" lastModifiedTime="1133228616435" name="Call Begin"
schemaVersion="20050818" forcesSessionClose="false" order="1.0">
  <sda:description>The Call Begin Informant starts the session after the
  customer's call enters the IVR system. Logic could be added here to pre-
  populate certain values (example: customer profile) that may be used later
  on.</sda:description>
  <sda:system ref="Ivr"/>
  <sda:sessionKey path="customer.customerId" relativeTo="session"/>
  <sda:requestMapper internalName="RequestMapper">
    <sda:entity type="ApplicationSession" var="session"/>
    <sda:dataSource type="RequestDataSource" var="result">
      <sda:arg>
        <sda:path localVarName="session" path="request" relativeTo="local"/>
      </sda:arg>
    </sda:dataSource>
  </sda:requestMapper>
  <sda:requestData internalName="RequestDataSource">
    <sda:param internalName="message" dataType="object"
objectType="com.sigmadynamics.client.wp.SDRequest"/>
    <sda:request>
      <sda:resultSet/>
    </sda:request>
  </sda:requestData>
  <sda:body>
    <sda:java order="0">/* Trigger data retrieval
*/&#xD;&#xA;session().getCustomer().fill(); </sda:java>
  </sda:body>
  <sda:postOutputBody/>
</sda:RTAPType>
```

The attributes that were assigned to the element in Decision Studio, such as **Session Key** and **External System**, are represented here. Note that the Java scriptlet is also inserted into the body of the XML file.

As Inline Service elements are added, configured, and saved, Decision Studio automatically generates the necessary Java code and compiles them into Java class files. Two classes of Java code are generated. The first set is the base Java files used by the Inline Service; these files are named the element id preceded by `GEN`. For example, the CallBegin element will produce a file called `GENCallBegin.java`.

The second set of Java files is created to allow overriding of the generated code. These files are named the same as the element ID. For instance, the CallBegin element will produce a file named `CallBegin.java`. Note that by default, the Java class `CallBegin` simply extends the class `GENCallBegin`.

When the Inline Service is compiled, the generated code is used unless we specifically instruct that the override code be used. To do this, update and move the override Java file (for example, `CallBegin.java`) from the generated source files folder:

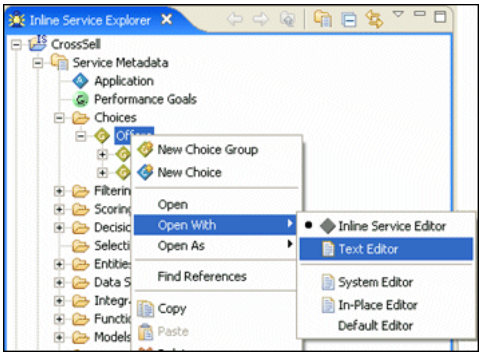
Inline_Service_project_root_folder`\gensrc\com\sigmadynamics\sdo`

to the override source files folder:

Inline_Service_project_root_folder`\src\com\sigmadynamics\sdo`

Decision Studio will now compile using the override Java file instead of the generated Java file.

> **Tip:** The XML for any Inline Service object can be viewed with Decision Studio's built-in Text Editor. Right-click an Inline Service object in the Inline Service Explorer View, then select **Text Editor** from the **Open With** menu. To switch back to normal editor format, select the option **Inline Service Editor**.



> Note that you should not edit the XML (`*.sda`) files directly to modify the Inline Service objects; instead, use the corresponding Inline Service Editors.

3. **Deploy:** The Inline Service is deployed to Real-Time Decision Server using Decision Studio. The Management Service on the server receives the metadata and compiled Inline Service files, stores the Inline Service in the database, and loads the Inline Service into memory. The Inline Service can now be utilized to process requests, view reports, and so on.
4. **View and Update:** Reports and learnings are viewed through the browser-based Decision Center interface. Selected elements and parameters of your Inline Service can be updated from Decision Center. Updated Inline Services are not available for run-time use until they are redeployed.

5. **Redeploy:** If updates are made to the Inline Service in Decision Center, the changes can be made available for use by redeploying the Inline Service in Decision Center. The Management Service will regenerate all necessary metadata and Java files, recompile the Inline Service, store the Inline Service in the database, and load it in memory.
6. **Download:** Using Decision Studio, you can download a deployed Inline Service from the server. Downloading involves copying the Inline Service that resides in the database and placing all of the metadata, Java, and class files into a Decision Studio project on the hard drive. This is useful if you were not the original developer of the Inline Service and thus do not have the metadata files. Even if you had originally developed and deployed the Inline Service from Decision Studio, if your business process allows other users to make changes and redeploy the Inline Service through Decision Center, then to make additional changes to the Inline Service in Decision Studio, you would first need to download the latest version from the server.

2

Creating an Inline Service

This section is designed to demonstrate how to build an Inline Service that acts as an Observer. Observer Inline Services are aimed at analyzing characteristics of target process on a real-time and continuous basis. An Observer Inline Service guides business users in their analysis of those various business events and how they change over time.

The Inline Service for this tutorial is based around a credit card company's call center. The Inline Service will collect data about the customer and the call center operational system and will analyze information about the call and its resolution.

The goal of this Inline Service is to analyze the patterns about calls, reasons for calling, and customers. In later sections, we will extend the capability of this Inline Service to provide recommendations to the CRM system on cross selling offers and then to add feedback to the service on the success of its recommendations.

This chapter contains the following topics:

2.1 About the Inline Service Tutorial

An Inline Service is created using the Decision Studio development tool. In general, an Inline Service is created in the following fashion:

- A project is started in Decision Studio.
- Elements are added to that project and then configured to support the desired business process.
- Logic is added in the form of Java scriptlets to certain elements that perform operations.
- The Inline Service is deployed to Real-Time Decision Server, where it runs.

- Reports generated from the use of the Inline Service are viewed through Decision Center.

In this tutorial the following elements are added and configured:

1. **Application:** The Application element establishes any application level settings that are needed, as well as defines security for the Inline Service. An Application element is automatically created for every Inline Service.
2. **Performance Goals:** Performance Goals represent organizational goals composed of metrics that are optimized using scoring. For instance, revenue and call duration are performance metrics. An organizational goal would be to maximize revenue while minimizing call duration.
3. **Data source:** The data source element acts as a provider of data from an outside data source. The structure and format of data from data sources can vary. For example:
   - Rows and columns of a RDBMS table
   - Output values and result row sets from a stored procedure

   A data source is a provider of data that you can map to Entity elements to supply the data for those elements.

   For example, in this tutorial we add a data source that connects to a table in the database. This table contains customer data.
4. **Entity:** The Entity is a logical representation of data that can be built from one or more data sources. Entities serve the following purposes:
   - To organize the data into objects for organizational, analytical, and modeling purposes.
   - To allow relatively easy and intuitive access from Java code of data from various sources.
   - To hide the details by which the data is obtained so that those details can change without requiring the logic to change.
   - To hide the mechanisms by which the data is obtained to save the user of this data from needing to deal with the APIs that are used to obtain the data.
   - To support sharing of objects when objects need to be shared. For example, an object representing a service agent could be used in multiple sessions.

   Attributes of an entity can be *key* values. The entity key is used to identify an instance of an entity.

   For example, in this tutorial we create an entity to represent a customer. The attributes of that entity are mapped to the data source for values. The customer ID is chosen as the key for the customer entity.

   Later we will also create an entity that represents a Call.
5. **Session Entity:** The Session entity is a special entity of an Inline Service. The Session entity represents a container for attributes that are specific to a particular defined Session. The *Session key* uniquely identifies each individual session.

   Entities that have been defined can be associated with the session by being made attributes of the Session Entity. Only Entities that are Session attributes can have their keys marked as session keys.

For example, in this tutorial we add the Customer entity to the Session entity as an attribute, and then we choose the Customer key value, Customer ID, as a Session key.

6. **Informant:** An Informant is an Integration Point within the Inline Service that identifies the business interactions as they occur and triggers business logic that continuously identifies relevant statistical patterns in the data. Informants watch a process; they do not interact with it.

   In this tutorial, we first create a testing Informant, and then create an Informant that gathers completion of service data from a CRM system.

   Later in the tutorial, we create an Informant that provides feedback to the Inline Service on the success or failure of the predictions of the model.

7. **Choice Groups:** Choice Groups are useful for organizing choices. Choice Groups can be used in one of two ways: they provide a way to organize the observations that are collected and analyzed; they are also a way to organize the feedback we will give to the business process through the Advisor Integration Points.

   For example, in this tutorial we first create Choice Group that organizes the reason for calls. When we extend the Inline Service to include an Advisor, a Choice Group is used to organize cross sell offers that are recommended to the service center agent.

8. **Models:** Built-in analytical models allow self-learning and automatic analysis. Models can be used to simply analyze data, or to make recommendations to the business process.

   In this tutorial, we create a model that analyzes the reasons for calls, and then later a model which helps to determine the most likely cross sell offer to be accepted by the customer.

9. **Decision:** A Decision is used by an Advisor to determine eligible Choices, score those Choices dynamically, weight the scoring according to segments of the population and its designated performance goals, and present to best-fit choice.

10. **Advisors:** An Advisor is an integration point that returns information back to the business process that calls it. Advisors call Decisions in the Inline Service that returns one or many ranked choices.

    In this tutorial, we will create a Choice Group of offers that can be made to callers to the credit card service center. The Advisor calls on a Decision to determine the best offer for the caller based on information about the call and caller. The Advisor passes that cross sell recommendation to the front end application, so that the call center agent can make the offer.

2.2 About Deployment and Decision Center Security

To be able to deploy your Inline Service to Real-Time Decision Server, and to see the results from the Inline Service in Decision Center, you must have the necessary roles, which are provided through a user name and password.

For this tutorial, it is assumed that an available user login and password has been created with the assigned roles of RTDUsers, RTDDecisionCenterUsers, and RTDStudioDeployers.

If necessary, contact the administrator responsible for installing and setting up your Oracle RTD system.

2.3 About Naming and Descriptions

Element names and descriptions are used extensively in Decision Center, the user interface for business users. Therefore, it is very important that as you create elements you take the time to name them intuitively and to write good descriptions for all elements.

Before you begin, ensure that Real-Time Decision Server is started. See *Oracle Real-Time Decisions Installation and Administration Guide* for more information about how to start Real-Time Decision Server.

To configure the Application element:

1. Open Decision Studio by running *RTD_HOME*`\eclipse\eclipse.exe`. After Decision Studio opens, choose **File > New > Inline Service Project** to begin a new project.

   Note: This tutorial assumes you are using a new installation, with the original preferences set. If Decision Studio or Eclipse has been used in the past, you may want to switch to a new workspace. To switch to a new workspace, choose **File > Switch Workspace** and choose a new workspace folder.

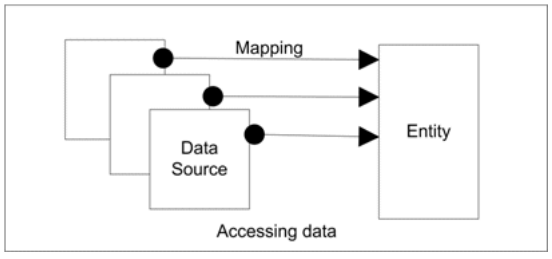
2. Enter the name for the project, `Tutorial`, and choose the **Basic** template. Click **Finish**. If you are asked about upgrading the Inline Service, select **Yes**. The Inline Service project is created in the Inline Service Explorer. By default, the files for the project/Inline Service are stored in the Decision Studio workspace, in a folder with the same name as the project (for example, `C:\Documents and Settings\`*`Windows_user`*`\Oracle RTD Studio\Tutorial`).
3. In Decision Studio, expand the **Tutorial > Service Metadata** folder. Double-click the **Application** element to bring up the element editor. In the element editor, type a description for the Tutorial Inline Service.

2.4 Accessing Data

In order to access organizational data, we will configure two elements:

- **Data source:** The data source is the element that represents the structure of the data in the database.
- **Entity:** The entity is a logical representation of data that can be populated by one or more data sources or contextual data retrieved by an Informant.

Figure 2–1 Data Source/Entity Mapping



This section contains the following topics:

- Section 2.4.1, "Adding a Data Source"

- Section 2.4.2, "Adding an Entity"

2.4.1 Adding a Data Source

Adding a data source involves creating the new data source, then importing the outputs for a data source.

This section contains the following topics:

- Section 2.4.1.1, "Creating the New Data Source"
- Section 2.4.1.2, "Importing the Outputs for a Data Source"

2.4.1.1 Creating the New Data Source

To create a data source:

1. In Decision Studio, select **Data Sources** in the Inline Service Explorer and right-click it. Select **New SQL Data Source**. For the **Display Label**, enter `Customer Data Source`, and click **OK**. The data source Editor appears.
2. Under **Description**, add the following description for the data source: `Customer data from a database table`.

   Good descriptions are very important. These descriptions are used in Decision Center and are essential for business users to identify components of reports and analysis.

   Note: You may notice that there are some other data sources already defined. These are part of the Inline Service framework and are not used in this tutorial.

2.4.1.2 Importing the Outputs for a Data Source

The outputs of a data source are the columns that are retrieved from the database. Outputs do not have to include all the columns in the table.

To import the outputs for a data source:

1. Click **Import**. Import Database Table appears. You server should appear next to **Server**. Click **Next** to connect to the server. Select Table or View appears.
2. For **JDBC Data Source**, select **SDDS**. Then, select **CrossSellCustomers** under **Tables and Views**. This table was created and populated by the default standard installation.
3. Click **Finish**.
4. All of the columns of the table are imported into the **Output** columns table.
5. Set the input column for the data source. The input is the column on which you will be matching to retrieve the data record. In this case, we can select the column name **Id** from the **Output** columns table and click the right arrow to move **Id** to the **Input** columns table. The data type is **Integer**.
6. Set the output columns for the data source. In the **Output** columns table, select and use **Remove** to remove all except the columns listed in Table 2–1.

Table 2–1 Output Columns to Retain in CrossSellCustomers Table

Name	Type
Age	Integer

Table 2–1 (Cont.) Output Columns to Retain in CrossSellCustomers Table

Name	Type
HasCreditProtection	String
Language	String
LastStatementBalance	Double
MaritalStatus	String
NumberOfChildren	Integer
Occupation	String

7. Save your work by choosing **File > Save All**. If there are errors in your work, you will receive notification in the Problems View.

> **Note:** You can use **Import** to import the column names and data types to the Outputs for the data source. Remove any columns you will not be using with **Remove**.

2.4.2 Adding an Entity

Now that we have the data source defined, we can proceed to define a corresponding Entity. Entities are the objects that are used by the other elements in the configuration. Entities provide a level of abstraction from sources of data such as Data Sources or Informants. A single entity can have data coming from many data sources, or even computed values. For now, we will create a simple entity that maps directly to the structure of the data source.

This section contains the following topics:

- Section 2.4.2.1, "Creating the New Entity"
- Section 2.4.2.2, "About Additional Entity Properties"
- Section 2.4.2.3, "Adding an Entity Key"

2.4.2.1 Creating the New Entity

To create the new entity:

1. In the Inline Service Explorer, right-click the **Entities** folder and select **New Entity**. For **Display Label**, enter the name `Customer` and click **OK**. The Entity Editor appears. Enter `Customer entity` for **Description**.

   Good descriptions for entity attributes are very important. Make sure you add a good description for every entity.

   > **Note:** Object IDs are automatically made to conform to Java naming conventions: variables are mixed case with a lowercase first letter; classes are mixed case with an uppercase first letter. If you have spaces in your label name, they will be removed when forming the object ID.
   >
   > Use the **Toggle** icon on the Inline Explorer task bar to toggle between the label of the object and its object ID:
   >
   > 

2. Click **Import** to import the attributes names and data types from **Customer Data Source**. Leave the option **Build data mappings for the selected data source** selected.
3. In the column **Default Value** of the **Definition** tab, click to get an insertion point and add a default value for each of the attributes listed in Table 2–2. Values for String data types will be automatically surrounded by double quotes.

Table 2–2 Attributes for Default Value Column of Definition Tab

Name	Type	Default Value
Age	Integer	35
HasCreditProtection	String	No
Language	String	English
LastStatementBalance	Double	1000
MaritalStatus	String	Single
NumberOfChildren	Integer	0
Occupation	String	Student

2.4.2.2 About Additional Entity Properties

You can modify additional settings about the attributes of an entity. For example, in more complex Inline Services, you may want to define categories of attributes. To do this, create a category element and assign it using the **Category** on the attribute's **Properties**. To view the properties of an attribute, select the attribute in the **Definition** tab, then right-click and choose **Properties** from the menu.

You may also want to indicate that an attribute should not be used for learning. For example, if you have the phone number of the customer, it does not make sense to have analytics on the number, so in that case you would deselect **Use for Analysis**.

The **Show in Decision Center** option is used to control whether the attribute is visible in Decision Center. This is useful when an attribute has only technical meaning and no direct or interesting business meaning.

2.4.2.3 Adding an Entity Key

In order to fully map the entity object to the data source, we need an entity attribute to map to the key value of the data source and complete the mapping. To do this:

1. On the **Definition** tab of the Customer Entity, click **Add Key** to add a key attribute. Add Key appears. Enter `customerId` for **Display Label**, add a description for the key value, change the data type to **Integer**, and click **OK**.
2. Save your work using **File > Save All**. You may see several errors in the Problems View - this is expected because the mapping definition of the Customer entity attributes to its data source is incomplete. Proceed to the next section in order to complete the mapping definition.

2.5 About the Session Entity

The Session is the root of run time data for a unit of a process. Data is kept in memory during the duration of the session. In order to track data about an Entity, we associate it with the Session entity that is part of the Oracle RTD framework. To associate the Entity to the session, make it an attribute of the Session entity. A key is chosen for the session. When a unique instance of that key is detected, the session begins.

As an example, consider a call center process being tracked by Oracle RTD. The Session contains entities that represent the Caller and the Agent. For the duration of the session (in other words, the call in this case) the data defined by those entities and the interaction between them is kept in memory and available for analysis and decision making.

This section contains the following topics:

2.5.1 Adding an Attribute to the Session Entity

To add an attribute to the Session entity:

1. In the Inline Service Explorer, double-click **Session** under **Entities**.
2. From the **Definition** tab, click **Add Attribute**. For **Display Label**, enter an attribute name, `customer`, then add a **Description**. Note that the initial data type is type **String**. We'll change this in the next step.
3. For **Data type**, select **Other**. A **Type** selection dialog appears. Under **Entity Types**, choose **Customer**. Click **OK**.

2.5.2 Creating a Session Key

To create a session key:

1. In **Session Keys from Dependent Entities**, click **Select**.
2. Expand the tree to see the keys of all entities associated with the Session. Expand **customer** and select **customerId** as a session key by checking the box. Click **OK**.

> **Tip:** Oracle RTD supports multiple session keys to enable the tracking of a session when different systems are sending Informants and Advisors to the same Inline Service. In this tutorial and in many real installations, only one session key is needed.

2.5.3 Mapping the Entity to the Data Source

We associate the Customer entity with the Customer Data Source through mappings defined in the Entity object editor. Our mapping of the Customer entity's attributes to the Customer Data Source *output* columns was automatically done when the attributes were imported from the Customer Data Source (see Section 2.4.2, "Adding an Entity" for more information). We need to now map the *input* column value for the Customer Data Source in the Customer entity.

To map the input column value for the Customer Data Source:

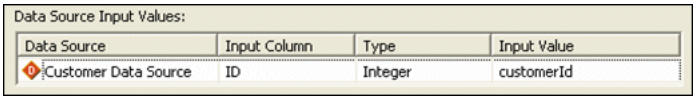
1. Open the Customer Entity and select the Mapping tab. Entity editors are identified by an **E** icon:

2. Because we used Import, the Customer Data Source attributes are already mapped to the Customer entity attributes. For attribute **Age**, the **Source** value should look like `Customer Data Source / Age` (or `Customer Data Source.Age` if the **Show Object ID** icon is selected). If you had added additional attributes beyond

the import, they are mapped by clicking the ellipsis under **Source** and locating the data source attribute.

3. We need to identify the input column values for each Data Source (in this case, Customer Data Source) in the **Attributes** table. The input columns for the data source are the identifier (the `WHERE` clause in a SQL select statement) by which records are retrieved. In the Customer Data Source, we had only one input column, **Id**, thus in the **Mapping** tab, we will see only one entry in the **Data Source Input Values** table, located below the **Attributes** table. In the **Input Value** cell of this entry, click the ellipsis to open the Edit Value dialog.
4. For this Inline Service, we will select the Customer entity's key. Choose **Attribute or Variable**. Expand **Customer**, select **customerId**, then click **OK**.

Figure 2–2 Edit Value Dialog for Id Column



5. Save the Inline Service by choosing **File > Save All**.

2.6 Adding an Informant

Informants are a type of Integration Point that can send a message to Real-Time Decision Server containing information about a specific unit in a process.

To test the Inline Service at this stage, we will create a testing Informant that prints out the age of the customer. To view the actual printed statement, we will need to deploy the Inline Service to Real-Time Decision Server and then call the Informant.

If we get a number back, we will know that the entity, mapping, and data source are working.

This section contains the following topics:

- Section 2.6.1, "Creating an Informant"
- Section 2.6.2, "Adding Testing Logic to the Informant"

2.6.1 Creating an Informant

To create an informant:

1. In the Inline Service Explorer, go to **Integration Points** and then select **Informants**. Right-click and select **New Informant** from the menu. Enter an object name, `Testing`, then click **OK**.
2. In the **Testing** Editor, add a description under **Description**.
3. Click **Advanced** next to **Description**. Deselect **Show in Decision Center**. This will make this Informant invisible to the business users of Decision Center. Click **OK**.

2.6.2 Adding Testing Logic to the Informant

To add testing logic to the Informant:

1. On the Testing Informant Editor, select the **Request** tab. Informants are identified by an **i** icon:

2. To add a session key, click **Select** under **Session Keys**. Choose **customerId** from **Customer**. Click **OK**.

> **Note:** When you configure an entity in Decision Studio, a class is generated. The generated classes have a property, getter, and setter for each attribute.

3. Choose the **Logic** tab. Under **Logic**, add the following scriptlet:

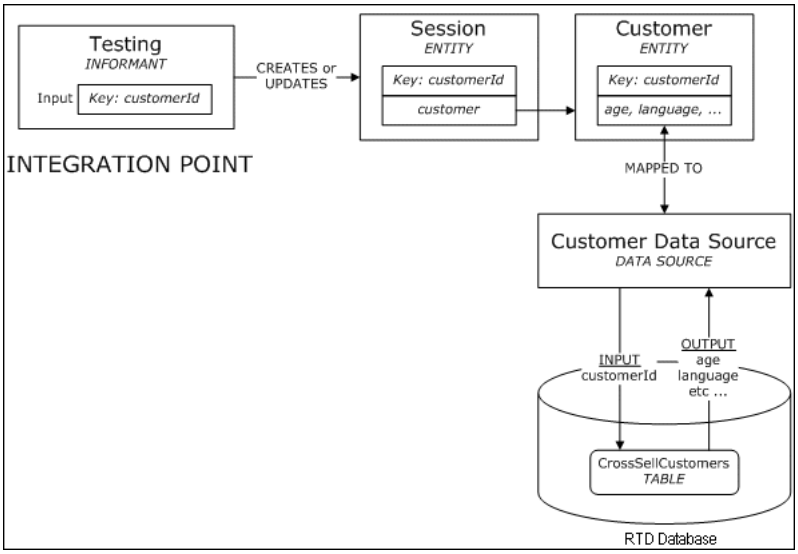
```
logInfo("Customer age = " + session().getCustomer().getAge() );
```

The logInfo call allows us to output information to the **Log** subtab of the Test view and also the server log file (usually in *RTD_HOME*\log). We will use the session to access the Customer object and get the contents of the age attribute.

4. Now we should be ready to deploy. Save the configuration by choosing **File > Save All**.

Figure 2–3 shows how the Testing Informant will access customer data when the Informant is called and a session created.

Figure 2–3 Tutorial Inline Service Objects



2.7 Testing the Inline Service

To test the Inline Service, we deploy it, call the Informant with test data, and use the Test View to observe the results. Because Informants do not return value to their callers, the results will be seen in the Log tab of Test View.

To deploy the Inline Service for testing:

1. Click the **Deploy** icon on the taskbar to deploy the Inline Service:

You can also use the menu item **Project > Deploy** to deploy your Inline Service.

2. Click **Select** to select the server where you want to deploy. Deploy to the location of your Real-Time Decision Server. This defaults to `localhost`, as does the default configuration of the installation. Enter the user name and password provided for you, as described in Section 2.2, "About Deployment and Decision Center Security." Use the drop-down list to select a deployment state, **Development**. Select **Terminate Active Sessions (used for testing)**. Click **Deploy**.

   Deployment takes anywhere from about 10 seconds to a few minutes. A message 'Tutorial deployed successfully' will appear below the Inline Service Explorer when deployment is complete.

   Note: The reason we terminate active sessions is that we want to make sure we are testing against the latest deployed Inline Service. If there were active sessions and we used the same session id (in this case, it is also the customerId), testing of the Informant would be against an earlier version of the deployed Inline Service. If we terminate the currently active sessions, then we are guaranteed to be testing against the latest deployed Inline Service, regardless of the session id used.

3. In the Test View at the bottom of the screen, select **Testing** as the Integration Point to test. Enter a value for **customerId** by typing `7` in the field. Click the **Send** icon:

4. Select the **Log** tab within Test View to see the results. Every printout coming from a `logInfo` command will be printed out with a timestamp.

   Your results should look similar to the following:

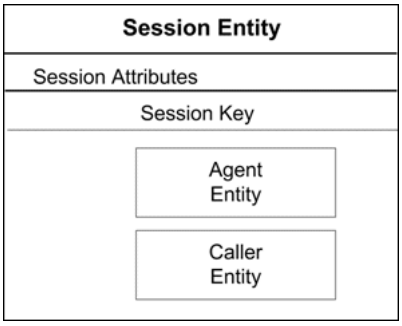
```
11:53:54,102 Customer age = 38
```

2.8 Adding Functionality

We will now create an entity to hold information specific to the call. This is contextual information about the nature of the interaction with the customer. The data in this entity will come from Informants or be computed, but it will not be retrieved from any database.

First we create an entity to represent a call, then an Informant that gathers data from calls. Choices are created as the targets of our analysis of the calls. In our case we are interested in focusing our analysis on the reasons for the calls.

Figure 2–4 Session Entity



Using this entity, we will explore the factors related to the reasons for calls, like the call lengths for each call reason, the most likely customer characteristics for these calls, and so on. In order to gather and analyze the call reasons gathered by the Informant, a self-learning analytical model will be added and reports will be displayed in Decision Center.

This section contains the following topics:

2.8.1 Creating a Call Entity

To create a call entity:

1. In the Inline Service Explorer, right-click the **Entities** folder and select **New Entity**. Enter the object name `Call` and click **OK**.
2. For each attribute listed in Table 2–3, do the following:
   - On the **Definition** tab of the Entity Editor, click **Add Attribute**. Add Attribute appears. Enter the values from the table and click **OK**.
   - Click in **Type**. Choose the proper data type for each attribute using the drop-down list.

Table 2–3 Attributes for Call Entity

Name	Type
agent	String
length	Integer
reason code	Integer

3. In the Inline Service Explorer, double-click **Session** under **Entities**.
4. From the **Definition** tab, click **Add Attribute**. Enter an object name, `call`. Note that the default type is **String**. We will change the default type in the next step.
5. For **Data type**, select **Other**. In the Type dialog box, expand **Entity types** and select **Call** as the type, then click **OK**. Add a **Description** for 'call'. Click **OK**.
6. Save the changes to the Inline Service by choosing **File > Save All**.

2.8.2 Creating the Call Begin Informant

We will now create three Informants that will be called by the CRM application: **Call Begin**, **Service Complete**, and **Call End**. The first, Call Begin, will start the session. In this Informant, we could optionally preload and cache certain session attribute values so they can be accessed more quickly later. For example, we may want to preload the customer's profile if this information will be used later on and the loading of this information is expected to be slow due to database calls or other constraints.

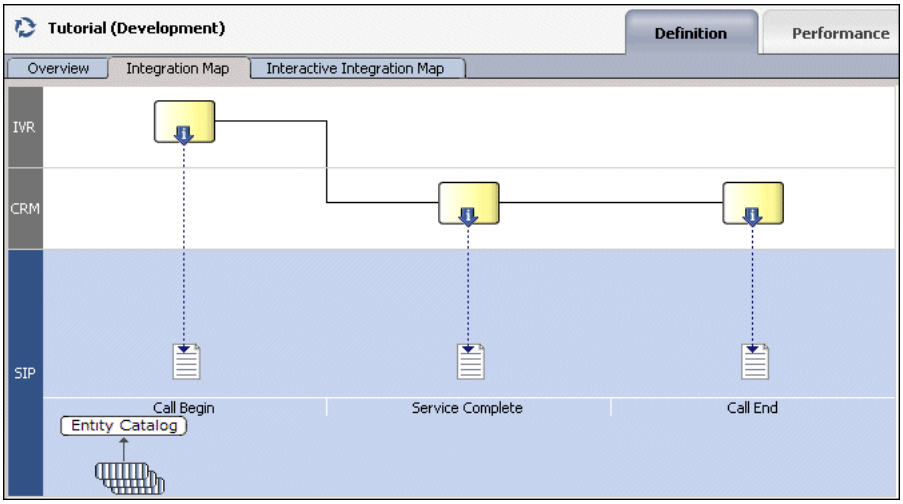
Note that it is not necessary to preload session attribute values as they are automatically loaded whenever they are needed. For example, when we want to print

the customer's Age, as the **Testing** Informant did in the previous section, Real-Time Decision Server will automatically populate the entire session's Customer entity attribute and return the Age value. In this Tutorial Inline Service, our Call Begin Informant will simply start the session, but will not pre-populate any session attribute values.

To create the Call Begin Informant:

1. In the Inline Service Explorer, under **Integration Points**, right-click the **External Systems** folder and select **New External System**. Object Name appears. Name the system `IVR` and click **OK**. Give the element a description. Save this object.
2. In the Inline Service Explorer, under Integration Points, right-click the **Informants** folder and select **New Informant**. Object Name appears. Name the Informant `Call Begin` and click **OK**.
3. Using the Informant Editor, enter a description for Call Begin.
4. To add a session key to the Call Begin Informant, click **Select** next to **Session Keys** in the **Request** tab. Choose **customerId**. Click **OK**.
5. While still in the **Request** tab, choose **IVR** from the **External System** drop-down list and enter `1` in the **Order** box. Do not select **Force session close**. The **External System** and **Order** determine the display layout and order in Decision Center's Integration Map (see Section 4.8, "Viewing the Integration Map" for more information). When we have finished defining the three Informants and deployed the Inline Service, the Integration Map in Decision Center will look like the one shown in Figure 2–5.

Figure 2–5 Decision Center Integration Map



****

6. In the Logic tab, add the following code:

```
/*
Prepopulate customer data during start of call even though the information may
not be used until much later.  This is not explicitly necessary since the
server will automatically retrieve the information whenever logic in the
Inline Service needs it.
*/
//session().getCustomer().fill();
```

```
int CustomerID = session().getCustomer().getCustomerID();
logInfo("Integration Point - CallBegin: Start Session for customerID = " +
CustomerID);
```

7. Save the changes to the Inline Service by choosing **File > Save All**.

2.8.3 Creating the Service Complete Informant

The second Informant will report on call information such as the agent that handled the call, the length of the call, and the reason for the customer's call. This Informant is called by the CRM application when the call center agent has responded to the customer's need, or in other words, when service is complete. The data that is gathered by the Informant will populate the Call entity.

To create the Service Complete Informant:

1. In the Inline Service Explorer, under **Integration Points**, right-click the **External Systems** folder and select **New External System**. Object Name appears. Name the system `CRM` and click **OK**. Give the element a description. Save this object.
2. In the Inline Service Explorer, under **Integration Points**, right-click the **Informants** folder and select **New Informant**. Object Name appears. Name the Informant `Service Complete` and click **OK**.
3. Using the Informant Editor, enter a description for **Service Complete**.
4. To add a session key to the Service Complete Informant, click **Select** adjacent to **Session Keys** in the **Request** tab. Select **customerId** and click **OK**.
5. While still in the **Request** tab, choose **CRM** from the **External System** drop-down list and enter `2` in the **Order** box. Do not select **Force session close**.
6. To add the additional pieces of data to the Informant, do the following *for each incoming parameter* listed in Table 2–4:
   - On the **Request** tab of the Informant Editor, click **Add**. Enter the name and then select the data type using the drop-down list. Click **OK**.
   - Under **Session Attribute**, click the ellipsis to use **Assignment**. Expand the **Session** folder, then expand **call** and then the select the call attribute that matches the incoming item.

Table 2–4 Data Types and Session Attributes for Incoming Parameters

Incoming Parameter Name	Type	Session Attribute
agent	String	call / agent (or call.agent if the Show Object ID icon is selected)
length	Integer	call / length (or call.length if the Show Object ID icon is selected
reason code	Integer	call / reason code (or call.reason code if the Show Object ID icon is selected)

7. In the **Logic** tab, add the following code:

```
logInfo("Integration Point - Service Complete");
logInfo("  Reason Code: " + session().getCall().getReasonCode());
logInfo("  Agent: " + session().getCall().getAgent());
logInfo("  Call Length: " + session().getCall().getLength());
```

8. Save the changes to the Inline Service by choosing **File > Save All**.

2.8.4 Creating the Call End Informant

The third Informant will close the session and could be the last Informant called by the CRM application. In this Tutorial Inline Service, we will only use this Informant to close the session, but in a real system, you might perform additional processing or trigger learning for a model.

To create the Call End Informant:

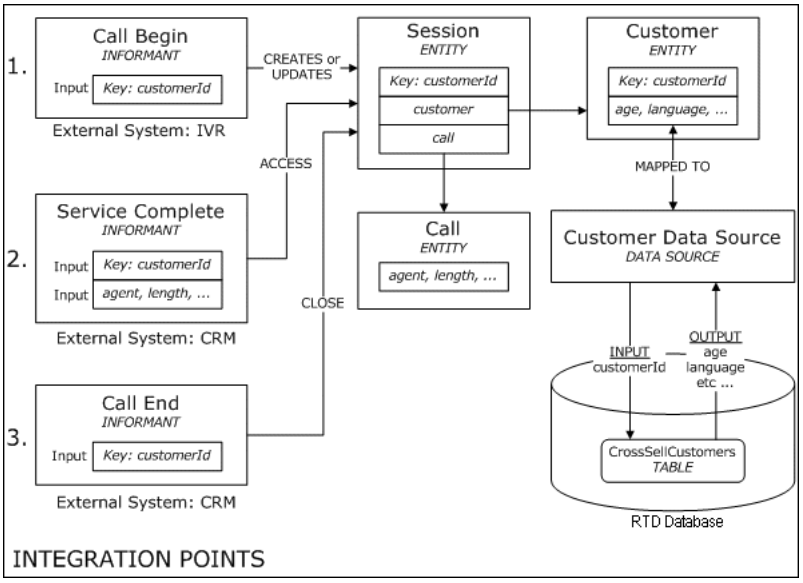
1. In the Inline Service Explorer, under **Integration Points**, right-click the **Informants** folder and select **New Informant**. Object Name appears. Name the Informant `Call End` and click **OK**.
2. Using the Informant Editor, enter a description for **Call End**.
3. To add a session key to the Call End Informant, click **Select** next to **Session Keys** in the **Request** tab. Select **customerId** and click **OK**.
4. While still in the **Request** tab, choose **CRM** from the **External System** drop-down list and enter `5` in the **Order** box. We set the **Order** to `5` because we will add two more integration points (an Advisor and another Informant) later in this tutorial.
5. Make sure the option **Force session close** is selected. Choosing this option will explicitly close the session once the Informant has been called and its logic processed. Note that if we do not explicitly close a session, the session will automatically close after some period of time - the default is 30 minutes and can be changed using JConsole.
6. In the Logic tab, add the following code:

```
logInfo("Integration Point - CallEnd");
logInfo("**************************");
```

7. Save the changes to the Inline Service by choosing **File > Save All**.

Figure 2–6 shows how the three Informants access and update the same session.

Figure 2–6 Tutorial Inline Service Objects: Integration Points



2.8.5 Testing the Informants

We will now test a simple scenario where three Informants you just created are called, corresponding to 1) start of a call, 2) service completion, and 3) end of the call. We will use the Test View to call the Informants and view the log messages we had placed in the logic portions of the Informants.

To test the Informants:

1. Deploy to the server. Click the **Deploy** icon on the taskbar to deploy the Tutorial Inline Service:

   Remember to select **Terminate Active Sessions (used for testing)**.

2. In the Test view, located in the bottom portion of Decision Studio, select the Integration Point **Call Begin**. For the request input **customerId**, enter an integer value, say `7`. Click the **Send** icon to send the request to the server:

   In the **Log** tab within the Test View, you should see a message indicating that the Call Begin integration point was called.

3. Repeat the process for the other two integration points, **Service Complete** and **Call End**, in order and with the input values as shown in Table 2–5. Examples of what you should see in the Log tab are also shown in Table 2–5.

Table 2–5 Input Values for Integration Points with Log Results

Integration Point	Request Inputs	Log Tab
Call Begin	customerId: 7	`09:15:41,753 Integration Point - CallBegin: Start Session for customerID = 7`

Service Complete	customerId: 7
	agent: John
	length: 21
	reason code: 18
		`09:17:51,845 Integration Point - Service Complete`
		`09:17:51,845 Reason Code: 18`
		`09:17:51,845 Agent: John`
		`09:17:51,845 Call Length: 21`

Call End	customerId: 7	`09:20:17,342 Integration Point - CallEnd` `09:20:17,342 **********`

Note that you could have called the Informants in any order. The Call Begin Informant is not needed to start a session and Call End is not needed to end the session. If we had called only the Service Complete Informant, the session would still be started correctly and the response would have been the same, although the session would then remain open. We are working with three Informants in this tutorial to demonstrate the different locations of a call center process that could make calls to Real-Time Decision Server.

> **Tip:** If there are errors in compilation, a dialog in Decision Studio shows these errors in the Problems View. Double-clicking the error takes you to the editor of the element that has the error.
>
> Make sure that the server with which you are communicating is `localhost`. Because Decision Studio remembers in the drop-downs the values previously entered, the default may not be `localhost`.

2.9 Analyze Call Reasons

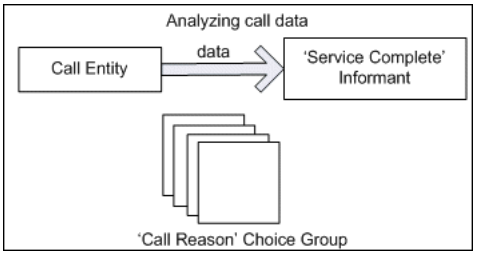
In the previous sections, we created three Informants. The second Informant, Service Complete, sends call information from the CRM application to Real-Time Decision Server. One of the pieces of call information is the call reason, or in other words, why the customer called. In this section, we will analyze the call reasons registered through the use of choices and a model. The objective is to able to view basic reports on call reasons - how many of each reason were recorded and how/if there were correlations between each call reason and session attributes.

This section contains the following topics:

2.9.1 About Using Choices for Analysis

Choices are used to create targets for analysis. In our case, we are first interested in focusing our analysis on the reasons for the calls. We will first create a choice group for the call reasons. Then, we will define an attribute for this choice group called `code`. The individual choices created within this group will then inherit the attribute definition, although the values can differ for each choice.

Figure 2–7 Analyzing Call Data



2.9.2 Adding a Choice Group

To add a choice group:

1. In the Inline Service Explorer, under **Service Metadata**, right-click the **Choices** folder and select **New Choice Group**. Name the group `Call Reason` and click **OK**. Add a description.
2. In the Choice Group Editor for **Call Reason**, in the **Choice Attributes** tab, click **Add** next to the **Attributes** table. For **Display Label**, enter `code`. Select the data type **Integer**, then select **Overridable**. Add the description `Choice codes` and click **OK**.

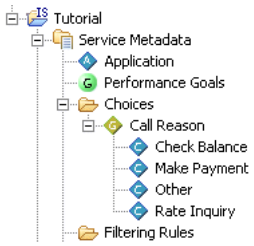
> **Note:** We made this a choice attribute as opposed to a group attribute. The difference between the two is that choice attributes are meant to be given values for each of the choices in the hierarchy, while group attributes are only given to the current group.

3. To create choices underneath the group, right-click the **Call Reason** choice group in the Inline Service Explorer and select **New Choice**. Add a Choice called `Check Balance`.

   Repeat this step for the choices `Make Payment`, `Rate Inquiry`, and `Other`. Add a description for each.

4. In the Inline Service Explorer, under **Choices**, expand the **Call Reason** group to show the choices.

Figure 2–8 Call Reasons in Inline Service Explorer



5. For *each of the four choices*, select the Choice in the Inline Service Explorer. In the Editor for that choice, under the **Attribute Values** tab, for the attribute **code**, set the **Attribute Value** as shown in Table 2–6.

Table 2–6 Attribute Values for Call Reason Choices

Choice	Attribute Value
Check Balance	17
Make Payment	18
Other	20
Rate Inquiry	19

6. Save the changes to the Inline Service by choosing **File > Save All**.

2.9.3 Adding an Analytical Model

A self-learning analytical Model is created to perform the automatic analysis of the reasons for calls. This model will track the reason for each call and correlate all the session attributes with these outcomes. Decision Center uses this model to build reports.

To add an analytical model:

1. In the Inline Service Explorer, under Service Metadata, right-click the **Models** folder and select **New Choice Model**. Name the model `Reason Analysis` and click **OK**. Make sure to create a Choice Model, and not a Choice Event Model.
2. Deselect **Use for prediction**.
3. To indicate that the target of analysis is the **choice** model attribute, select the **Choice** tab and choose **Call Reason** from the **Choice Group** drop-down list.
4. Select the **Learn Location** tab, then select **On Integration Point.**
5. Click **Select**, then select **Service Complete** and click **OK**.
6. Save the changes to the Inline Service by choosing **File > Save All.**

2.9.4 Adding Logic for Selecting Choices

When the Service Complete Informant is received, we need to select the choice that represents the corresponding reason for the call. We will do so by adding reasons to the model's choice array using the method of the Choice Model **addToChoice**.

To add reasons to the choice array of the model:

1. In the Inline Service Explorer, expand **Integration Points**. Under **Informants**, double-click **Service Complete**.
2. Select the **Logic** tab and enter the following logic. This adds the Object ID of the Choice that represents the reason for call to the model.

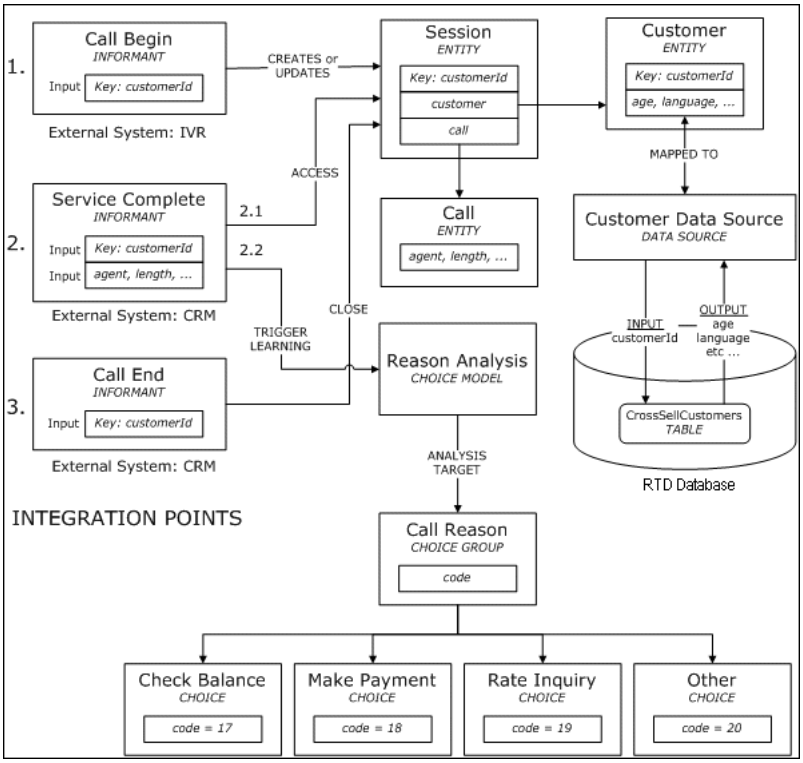
```
logInfo ("Integration Point - Service Complete. ");
logInfo ("  Reason Code: " + session().getCall().getReasonCode());
logInfo ("  Agent: " + session().getCall().getAgent());
logInfo ("  Length: " + session().getCall().getLength());

int code=session().getCall().getReasonCode();
switch (code) {
  case 17:
      ReasonAnalysis.addToChoice("CheckBalance");
      logInfo ("  CheckBalance was added to the model");
    break;
  case 18:
      ReasonAnalysis.addToChoice("MakePayment");
      logInfo ("  MakePayment was added to the model");
    break;
  case 19:
      ReasonAnalysis.addToChoice("RateInquiry");
      logInfo ("  RateInquiry was added to the model");
    break;
  default:
      ReasonAnalysis.addToChoice("Other");
      logInfo ("  Other was added to the model");
    break;
}
```

3. Save the configuration by choosing **File > Save All**.

Figure 2–9 shows how the Reason Analysis model is updated when the Service Complete Informant is called.

Figure 2–9 Tutorial Inline Service Objects



2.9.5 Testing It All Together

To test all the parts of the configuration:

1. Deploy the configuration to the server. Make sure there are no errors in deployment or compilation.
2. Use the Test view to test the Integration Point. Select **Service Complete** and set values for the different arguments: `customerId = 7`, `agent = John`, `length = 21`, and `reasonCode = 18`.
3. Click **Send**. You should see results similar to the following:

```
13:57:29,794 Integration Point - Service Complete.
13:57:29,794   Reason Code: 18
13:57:29,794   Agent: John
13:57:29,794   Length: 21
13:57:29,794   MakePayment was added to the model
```

   When this Informant with the shown input values is called, the Call entity, which is an attribute of the Session, is populated with information about the agent, length of call, and reason code. The Informant logic then determines that since the reason code was 18, then the Make Payment choice will be added to the Reason Analysis model. That is, the count for the Make Payment choice will have been increased by one. Along with the count, the model also tracks all of the session attributes and correlation with the choices.
4. Change the values and test a few times to see that the correct Object ID is being added to the model for other reason codes.

3

Simulating Load for Inline Services

This chapter of the tutorial contains step-by-step instructions for utilizing Load Generator to simulate the run-time operation of the system. In general, Load Generator is used in three situations:

- **Performance:** To characterize the performance of the system under load, measuring response time and making sure the whole system, including back-end data feeds, can cope with the expected load.
- **Initialize:** To initialize the learnings and statistics with some significant data for demonstration purposes.
- **Process:** To process off-line batch sources - for example, when exploring previously collected data at the initial stages of a project.

This chapter contains the following topics:

3.1 Performance Under Load

To evaluate performance under load, we will create a load-simulator script that calls the integration points defined in the Inline Service: Call Begin, Service Complete, and Call End. The script will call this series of three integration points multiple times, each time with different customer id's, call reason codes, agent names, and call lengths. The Reason Analysis model will learn on each of these iterations and we will be able to see the analysis results in Decision Center reports.

For this tutorial, we can think of Load Generator as simulating the CRM application making multiple iterations of integration point calls to Real-Time Decision Server. The Load Generator script (saved as an `xml` file) we will create will contain the definition of this simulation.

Note: When defining the Load Generator script, all references to Inline Service objects must be in the form of object IDs, not labels. To view the object IDs in Studio, use the object ID **Toggle** icon on the Inline Service Explorer taskbar:

For example, the ID for the **Service Complete** Informant is `ServiceComplete`, the ID for the **reason code** Informant parameter is `reasonCode`, and so forth.

This section contains the following topics:

3.1.1 Creating the Load Generator Script

To create the Load Generator script:

1. Open Load Generator by running *RTD_HOME*\scripts\loadgen.cmd. Then, click **Create a new Load Generator script**.

   You can press **F1** to read the online help for Load Generator. It explains the parameters that are not explained in this tutorial.

2. Select the **General** tab and enter values for the parameters as shown in Table 3–1.

Table 3–1 Parameters for General Tab

Parameter Name	Explanation	Value
Client Configuration File	A properties file that indicates the protocol to be used to communicate with the server, what server to talk to and through what port. The default is to communicate using HTTP to the local server using port 8080. The default file is suitable for our needs.	*RTD_HOME*\client\clientHttpEndPoints.properties
Graph Refresh Interval in Seconds	This parameter only affects the user interface. It determines the refresh rate for the UI graph and counters. The default is to refresh every 2 seconds.	2
Inline Service	This is the name we gave the Inline Service we created in the previous section.	Tutorial
Random Number Generator Seed	The seed used to generate random numbers. Default is -1.	-1
Think Time	Think Time is the time in between transactions. In a session oriented load simulation you would give different numbers here. For this tutorial we will explore the maximum throughput, sending as many sessions as possible. Values for Think Time can be fixed or a range of values.	Fixed Global Think Time
Constant	A fixed constant for think time in between transactions.	0
Number of concurrent scripts to run	This is the number of sessions active at any given point, running in parallel. In this case we will just run one session at a time.	1

Table 3–1 (Cont.) Parameters for General Tab

Parameter Name	Explanation	Value
Maximum number of scripts to run	The total number of session that will be created. Load Generator will stop sending events after this number has been reached.	2000
Enable Logging	Option to enable/disable loadgen counters log. This log maintains a history of loadgen performance data, including the number of requests sent by Load Generator, number of errors, the average and peak response times of a request, etc. If deselected, the remaining three logging parameters (Append to Existing File, Log File, Logging Interval in Seconds) are ignored.	Deselected
Append to Existing File	Option to indicate whether to overwrite or append to an existing log file each time a loadgen script is run.	Deselected
Log File	File path to an `ascii` file. This is the location where the Load Generator log will be written, in tab-delimited format.	*RTD_HOME*`\log\loadgen.csv`
Logging Interval in Seconds	This parameter only affects the Load Generator log. It determines the interval for writing to the log. The default is 10 seconds.	10

In the path names, make sure to replace *RTD_HOME* with the path where you installed Real-Time Decision Server (for example, `C:\OracleBI\RTD`).

Note that parameters related to sessions cannot be changed in the middle of execution. More precisely, they can be changed, but their changes will not affect execution until the Load Generator script is stopped and restarted.

3. Save the configuration. It is customary to save Load Generator scripts in a folder named `etc` within the inline service project folder. If you had created the **Tutorial** Inline Service in the default workspace, the path would be similar to: `C:\Documents and Settings\`*Win_User*`\Oracle RTD Studio\Tutorial\etc`. Name the script (an `xml` file) anything you like (for example, `TutorialLoadgen.xml`).

4. To define the values for the parameters to the Integration Point, click the **Variables** tab. Variables allow an Integration Point's parameter values to be drawn from different sources.

> **Note:** It is possible that not all the tree is visible on the left. To make it all visible, you can drag the bar dividing the two areas.

5. Right-click the **Script** folder and select **Add Variable**. Name it `var_customerId`. In **Contents**, select **Integer Range** from 1 to 2000, sequential. This definition will create a variable that is computed once per session and goes from 1 to 2000 sequentially, that is, the first session will have `var_customerId` = 1 and the last

one will be 2000. Right-click **Script** and select **Add Variable** three more times for a total of four variables, as shown in Table 3–2.

Table 3–2 Variable Names and Values for Load Generator Script

Parameter Name	Content Type	Value
var_customerId	Integer Range	Minimum = 1, Maximum = 2000, Access type = Sequential
var_reasonCode	Integer Range	Minimum = 17, Maximum = 20, Access type = Random
var_agent	String Array	To add a string to the array, right-click on the table area and select **Add Item**. Then select (double-click) the newly created row to get a cursor and type the name to be used. Press the **Enter** key to register the value for that row. Add a few sample values of agent names (for example, John, Peter, Mary, and Sara).
var_length	Integer Range	Minimum = 75, Maximum = 567, Access type = Sequential. This will be used as the length of the call in seconds.

6. Select the **Edit Script** tab, then right-click the left area and select **Add Action**. We will add three actions, each corresponding to an integration point. We need the actions to be in the right order - **CallBegin**, **ServiceComplete**, and finally **CallEnd**.
7. For the first action, set the type to **Message** and the Integration Point name to **CallBegin**. In **Input Fields**, right-click and choose **Add item** to add an input field. Double-click in the space under **Name** and enter the value **customerId**; press **Enter** to register the new value. In the **Variable** column for customerId, choose **var_customerId** from the drop-down list. Select **Session Key** to identify this field as the session key.
8. For the action **ServiceComplete**, add three additional fields, as shown in Table 3–3.

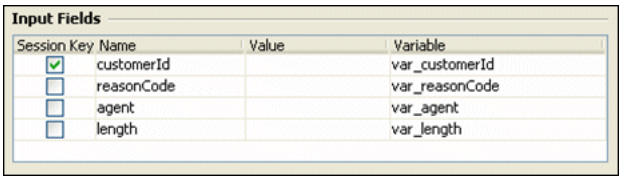
Table 3–3 Additional Input Fields for ServiceComplete

Name	Variable
reasonCode	var_reasonCode
agent	var_agent
length	var_length

Again, the names here must match exactly with the incoming parameter IDs for the ServiceComplete Informant as seen in Decision Studio. Use the **Toggle** icon on the Inline Service Explorer task bar in Decision Studio to toggle between the label of the object and its object ID:

Figure 3–1 shows what the completed **Input Fields** section for ServiceComplete should look like.

Figure 3–1 Input Fields for ServiceComplete



9. In the **Edit Script** tab, right-click the left area and select **Add Action**. Set the type to **Message** and the Integration Point name to **CallEnd**. In **Input Fields**, right-click and chose **Add item** to add an input field. Set the **Name** to `customerId`, the **Variable** to `var_customerId`, and select **Session Key**.
10. Once again, save the Load Generator configuration script. Our Load Generator script now contains calls to three integration points. Make sure the order of the actions in the Edit Script tab is correct: **CallBegin**, **ServiceComplete**, and **CallEnd**. If not in this order, right-click the actions to move items up or down. Then, save the script again.
11. Go to the **Run** tab and press the Play button. Allow Load Generator to complete.

Note that there is a **Pause** button and a **Stop** button. The difference between these two is that **Pause** remembers the sequences and will continue from the point it was paused, whereas **Stop** resets everything.

Tip: If you encounter problems, look at the **Total Errors** in the **Run** tab. If the number is above 0, look at the server output window. There may be an indication of the problem. Common mistakes are:

- The Inline Service has not been deployed.
- There is a spelling or case mistake in the name of the Inline Service or the Integration Point.
- The server is not running.

If the **Total Requests** stays at 1 and does not grow, there may be a mistake in the definition of the loadgen script. Some things to look for include:

- In **Integer Range** variables, make sure the Minimum is below the Maximum.
- Make sure that the mapping of values sent in messages to variables is correct. For example, if a variable name is changed, the mapping needs to be redone.
- Make sure the Client Configuration file is correct.

3.1.2 Viewing Analysis Results in Decision Center

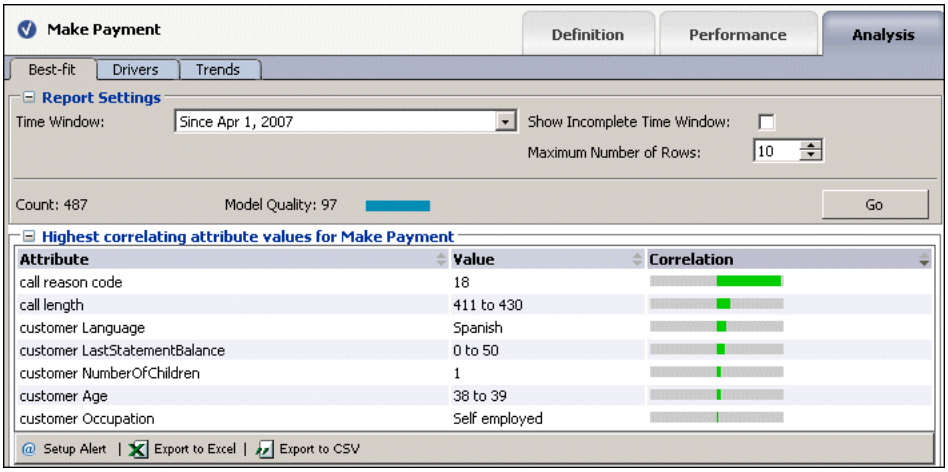
You can use the Decision Center to check what has been learned by the models after running Load Generator. To do this:

1. Open Decision Center by opening a Web browser and going to the URL `http://`*`server_name`*`:8080/ui`. Log in using the user name and password provided for you, as described in Section 2.2, "About Deployment and Decision Center Security."

2. Click **Open Inline Services**. The Select Inline Service window appears. Select **Tutorial**, then expand **Call Reason** and select one of the Choices, such as **Make Payment**. In the right pane, navigate to the **Analysis** tab and then the **Best-fit** subtab. This report summarizes the number of times this call reason was seen, and correlations between this call reason and attribute values.

   You will see something interesting. The **call reason code** has an unexpectedly strong correlation to **Make Payment**, as shown in Figure 3–2.

Figure 3–2 Correlating Attribute Values for Make Payment



Since we generated the call data randomly with Load Generator variables, we would not expect to have any significant correlations. In this case, however, the call reason code (sent by the ServiceComplete Informant) absolutely determines the call reason (see Section 2.9.5, "Testing It All Together" for a discussion of this logic).

To remove this induced correlation, we should exclude this attribute from being used as an input to the model. Another type of attribute we might exclude from models is a customer's full telephone number. After all, it is unlikely that correlations can be found between unique telephone numbers and the reason he/she called. On the other hand, there may be correlations between the area codes of customers and the call reasons, so this would be an attribute that we would not exclude from the model. In the next section, you will exclude the 'reason code' attribute from the model and re-run the Load Generator script.

3.1.3 Excluding the Attribute

To exclude the reason code attribute:

1. In Decision Studio, open the **Tutorial** project.
2. Expand **Service Metadata > Models**, then double-click **Reason Analysis** from the Inline Service Explorer.
3. Go to the **Attributes** tab. In the lower table, titled Excluded Attributes, click **Select** to choose an attribute to exclude. Expand the **Session** node, then expand the **call** entity and select **reason code**. Click **OK**.
4. Save all and redeploy to the `localhost` server.
5. You can now re-run the Load Generator script.

If you use Decision Center now to look at the counts, you will notice that they include the events from both runs of Load Generator. This happens because we did not reset the models between the two times we ran the Load Generator script.

3.2 Resetting the Model Learnings

Use the JConsole administration tool to reset the Model learnings, as follows:

1. If you are using OC4J or WebLogic, open JConsole by running *JAVA_HOME*\bin\jconsole.exe. If you are using WebSphere, run the batch script you created during JConsole configuration. See *Oracle Real-Time Decisions Installation and Administration Guide* for more information about accessing JConsole.
2. Click the **Remote** tab. Then, enter the appropriate port number (typically 12345) and the administrator credentials you created during installation and click **Connect**.
3. Click the **MBean** tab, then go to the **OracleRTD > InlineServiceManager > Tutorial > Development > Loadable** MBean.
4. Click the **Operations** tab, then use the **deleteAllOperationalData()** operation to remove all operational data, including the study, for this Inline Service.
5. To see the new results in Decision Center, run the Load Generator script again.

3.2.1 Summary of the Inline Service

We have so far created a simple but fully functional Inline Service. We did so by starting with the definition of the data environment, the data source and entity for the customer, and then the entity for the current call data. After testing the basic functionality, we created several Integration Points and a model to perform the analysis. Logic was added to determine the reasons of the customer calls and to record occurrence of the different reasons in a model for analysis purposes. We then used Load Generator to simulate requests against Real-Time Decision Server and the Tutorial inline service. The results are then viewed in Decision Center.

4

Enhancing the Call Center Inline Service

In Chapter 2, we created an Inline Service that tracks and analyzes incoming data related to the reason for calls to a credit card call center. In Chapter 3, we used Load Generator to simulate client requests (through Informant calls) to our Inline Service.

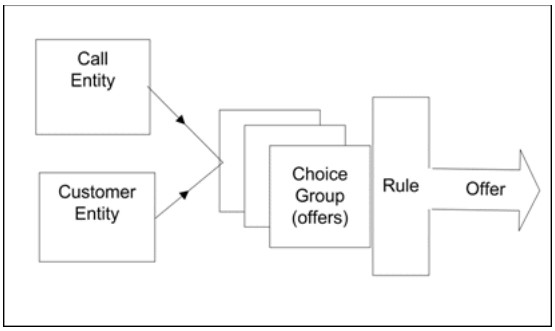
In this chapter, we will enhance the Tutorial Inline Service to provide cross-selling advice to the CRM application. The process enhancement is this: after the agent has finished processing the customer's call in the normal way (Service Complete Informant called), the agent then takes the opportunity to present an offer to the customer. In Chapter 5, we will track the customer's response to the offer, and then use what was learned from the responses in presenting offers to other customers.

This chapter contains the following topics:

4.1 About Using Choice Groups and Scoring to Cross Sell

We will create a choice group of offers that can be extended to customers calling the service center. Choice scores are based on cost in order to support our Performance Metric of minimizing cost. Next, an Advisor is created to pass that cross sell recommendation to the CRM application, so that the call center agent can extend the offer.

Figure 4–1 Cross-Selling Offer



4.2 Creating an Offer Inventory Using Choice Groups

To create an offer inventory using Choice Groups:

1. In the Inline Service Explorer, right-click the **Choices** folder and select **New Choice Group**. Name the group `Cross Selling Offer` and click **OK**.
2. Expand **Choices** and select/open the newly created group. Add a description.
3. In the **Choice Attributes** tab, click **Add** next to the **Attributes** table. Add the attributes shown in Table 4–1, making sure to select **Send to client** and **Overridable**.

Table 4–1 Attributes for Choice Attributes Tab

Attribute Name	Data Type	Send to client	Overridable
Offer Description	String	Selected	Selected
URL	String	Selected	Selected
Agent Script	String	Selected	Selected

Note: These attributes are sent to the client because they are needed by the client (call center agent) to present the offer.

The attributes should be overridable because their values will be different for each actual offer. Cross Selling offers will be represented by choices in this choice group.

In a real Inline Service, we are likely to see several levels of choice groups before we get to actual offers. Each choice group provides a logical group for offers, and may have attributes or business rules that apply uniformly to a group of offers.

---

4. In the Inline Service Explorer, under **Choices**, select the **Cross Selling Offer** choice group and add five choices with their attributes, as shown in Table 4–2. To add each of the choices, follow these steps:
   a. Right-click **Cross Selling Offer** in the Inline Service Explorer and select **New Choice**. Add the following choices: **Credit Card**, **Savings Account**, **Life Insurance**, **Roth IRA**, and **Brokerage Account**.
   b. In the Inline Service Explorer, under **Choices**, expand the **Cross Selling Offer** Group to show the choices. For each choice, follow these steps to add attributes:

- Select the Choice in the Inline Service Explorer. In the Editor for that choice, add a description.
- On the **Attribute Values** tab, you will see three attributes: **Agent Script**, **Offer Description**, and **URL**. Using **Attribute Value**, add the attribute values shown in Table 4–2.

Table 4–2 Attribute Values for Choices

Choice Name	Agent Script	Offer Description	URL
Brokerage Account	Would you like to try our new brokerage account?	Brokerage Account offer	`http://www.offer.com/offer1.html`
Credit Card	Would you like to try our new credit card?	Credit Card offer	`http://www.offer.com/offer2.html`
Life Insurance	Would you like to try our new life insurance?	Life Insurance offer	`http://www.offer.com/offer3.html`
Roth IRA	Would you like to try our new Roth IRA?	Roth IRA offer	`http://www.offer.com/offer4.html`
Savings Account	Would you like to try our new savings account?	Savings Account offer	`http://www.offer.com/offer5.html`

5. Save the configuration by choosing **File > Save All**.

4.3 Configuring Performance Goals

To configure performance goals:

1. In the Inline Service Explorer, double-click the **Performance Goals** element to open the editor. Click **Add** to add a Performance Metric. Name the metric `Cost`. Click **OK**.
2. In **Optimization**, choose **Minimize** and make the metric **Required**.

> **Note:** If you have more than one performance metric, you must use the Normalization Factor to normalize the values. For instance, if you had another metric called "Minimize hold time" measured in seconds, the normalization factor would be how many minimized seconds are worth a dollar (revenue) to your organization.

3. Save the configuration by choosing **File > Save All**.

4.4 Scoring the Choices

Each product costs the company an average amount to maintain on a yearly basis. The cost in dollars is the score for that product.

To score the choices:

1. In the Inline Service Explorer, under **Choices**, select and open the **Cross Selling Offer** choice group. In the **Scores** tab, click **Select Metrics** and choose the

performance metric **Cost**, then click **OK**. This sets up the choice group with a Cost score. The actual score values will be set on a per-choice basis.

Score values do not have to be constants. In many cases, score for one type of customer can differ by a significant amount from another customer type. We can express such differences through the use of formulas or **scoring rules**. For example, the Cost to maintain a credit card account may be less for customers who are age 40 or under. We will define this logic in a **Scoring Rule** and then assign this rule to the Cost score for the Credit Card offer.

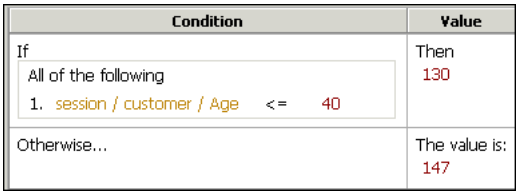
2. In the Inline Service Explorer, right-click the folder **Scoring Rules** and select **New Scoring Rule**. Name the scoring rule `Credit Card Score`. The editor for this new rule opens.
3. Click the **Add conditional value** icon to set up a rule condition in addition to the default:

A new row will appear, where the left cell is a two-sided rule and the right cell is the score value. The logic is as follows: "If the left cell evaluates to true, then the value in the right cell will be returned, otherwise use the value in the second row of the rule."

Click the left side of the rule and then on the ellipsis. An Edit Value dialog appears. Select **Attribute**, expand **session attributes > customer** and select **Age**, then click **OK**. Click the condition operator icon, then click the lower-right corner triangle:

Select the less than or equal to symbol (**<=**). Click in the right half of the rule and type the number `40`. In the **Then** cell, type the number `130`. In the second row, select and type in the number `147` for the value. The full rule should look like the one shown in Figure 4–2.

Figure 4–2 Completed Rule Condition



Save the Credit Card Score scoring rule. For the other offers, we will set constant values for the Cost score.

4. For each of the choices under the choice group **Cross Selling Offer**, open the **Scores** tab. In the Score column for the Cost metric, enter the values shown in Table 4–3. To set the Cost score for the Credit Card choice, click the ellipsis in the **Score** column, then select **Function or rule call** as the **Value Source**. In the **Function to Call** drop-down list, select **Credit Card Score**.

Table 4–3 Cost Scores for Choices

Choice	Cost Score
Brokerage Account	150

Table 4–3 (Cont.) Cost Scores for Choices

Choice	Cost Score
Credit Card	**Credit Card Score** Scoring Rule: 130 if Age <=40, Otherwise 147
Life Insurance	140
Roth IRA	145
Savings Account	135

Since our Performance Goal is to minimize costs, it is clear that the Savings Account offer (score =135) will be chosen unless the customer's age is 40 or below (score = 130), in which case the Credit Card offer will be chosen. In later sections of the tutorial, we will add another Performance Goal, Maximize Revenue, to see how these two competing performance metrics are optimized by the platform.

5. Save the configuration by choosing **File > Save All**.

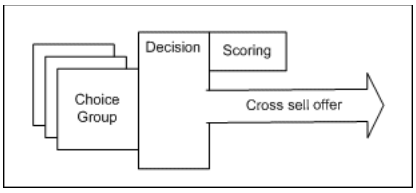
4.5 About Advisors

When an external system needs a decision to be made on its behalf, it calls an Advisor. Here, we create the Advisor that will send back to the CRM application an offer selected for a specific customer.

The Advisor's internal structure includes a Decision which associates it with one or more Choice Groups. These Choice Groups contain the offers that are to be made. The result of the decision is the result sent to the Advisor.

An Advisor has two decisions, one for normal processing and the other for the control group. The control group serves as a baseline to show performance gains achieved by Oracle RTD.

Figure 4–3 Structure of Get Cross Sell Offer Advisor



4.6 Creating the Decisions

To create the decisions:

1. In the Inline Service Explorer, right-click the **Decisions** folder and select **New Decision**. Name the Decision `Select Offer` and click **OK**.
2. Add a description for **Select Offer**. On the **Selection Criteria** tab in the Decision editor, locate **Select Choices from**. Click **Select**, then select the **Cross Selling Offer** from the list and click **OK**.
3. For our control group, we will have a decision that chooses an offer randomly. Create a new Decision and name it `Random Choice`.
4. Add a description for **Random Choice**. On the **Selection Criteria** tab in the Decision editor, locate **Select Choices from**. Click **Select**, then select the **Cross Selling Offer** from the list and click **OK**.

5. Check the **Select at random box**.

> **Note:** The Control Group acts as a baseline so that the business user can compare the results of the predictive model against the pre-existing business process. It is important to correctly define the Control Group decision to truly reflect the decision as it would have been made if Oracle RTD was not installed. For example, in a cross-selling application for a call center, if agents randomly selected an offer before Oracle RTD was introduced, then the Control Group Decision should return a random selection.

6. Save the configuration by choosing **File > Save All**.

4.7 Creating the Advisor

To create the Advisor:

1. In the Inline Service Explorer, under **Integration Points**, right-click the **Advisors** folder and select **New Advisor**. Name the element `Get Cross Sell Offer` and click **OK**.
2. To add a session key to the Get Cross Sell Offer Advisor, click **Select** under **Session Keys** in the Editor and select **customerId**. Click **OK**.
3. Under **External System**, select **CRM**. For **Order**, enter `3`.

   Recall that we had set the **Order** for Informant 'Service Complete' to `2`. We are preparing to call the Get Cross Sell Offer Advisor after this Informant, and thus the order number 3. Note that the **Order** is only used in Decision Center's integration map to help graphically describe the application process; **Order** does not force integration points to execute in any particular sequence.
4. On the **Response** tab, select a **Decision** for both the normal processing and the control group. Select **Select Offer** for the Decision and **Random Choice** for the Control Group Decision.
5. In the **Default Choices** section, click **Select**, then choose **Life Insurance** from the list and click **OK**. This will make the selected Offer the default response for this Advisor. This default will be used when there is any problem in the computation (for instance, if there is a timeout).
6. In the **Asynchronous Logic** tab, enter the following code:

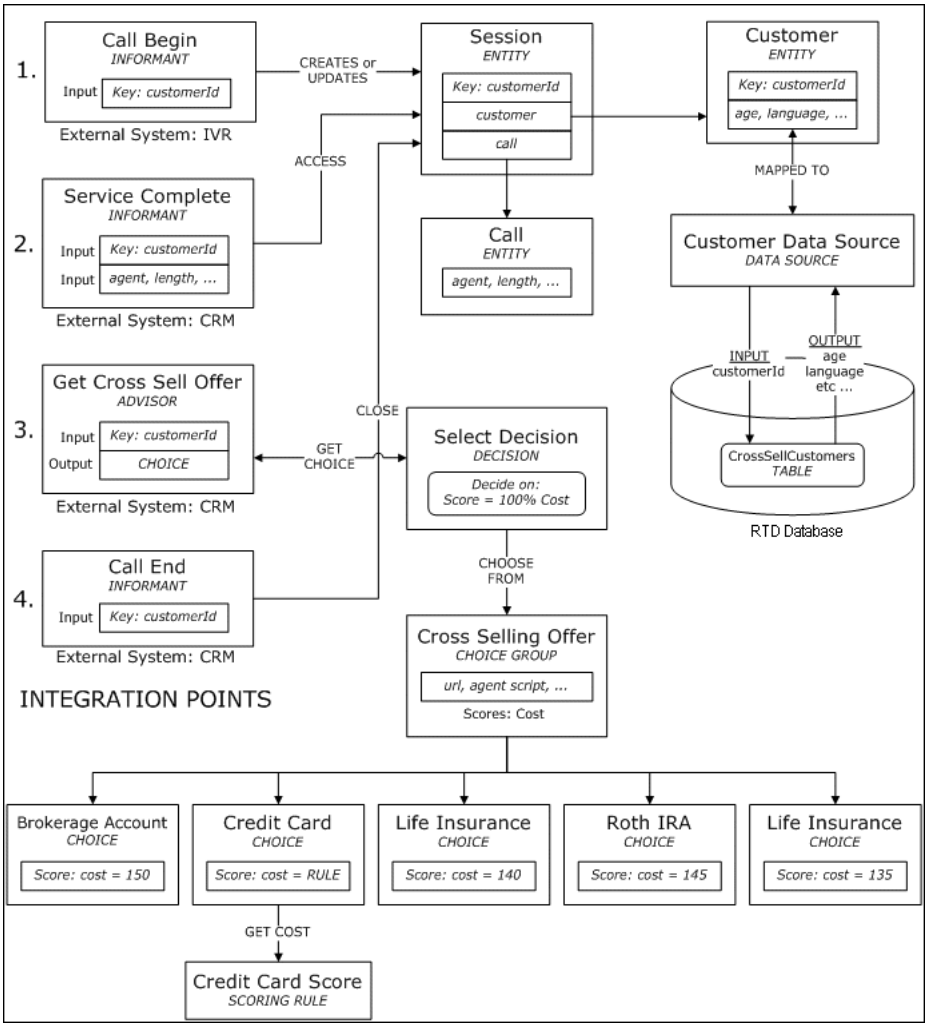
```
logInfo("Integration Point - Get Cross Sell Offer");
logInfo("  Customer age = " + session().getCustomer().getAge() );
// 'choices' is array returned by the 'Select Offer' decision
if (choices.size() > 0) {
  //Get the first offer from array
  Choice offer = choices.get(0);
  logInfo("  Offer presented: '" + offer.getSDOLabel() + "'");
}
```

   If we had entered the code in the **Logic** tab, it would have been executed before the decision was made on which offer to return, and we would not be able to print the name of the offer returned. In the preceding code, we print the customer's age and the presented offer name. Recall that because we are minimizing on Cost, only the offers Savings Account and Credit Card will be presented, depending on the age of the customer.

7. Save the Inline Service. Click the **Deploy** button. Select **Terminate Active Sessions (used for testing)** to remove any session that is still currently active. Deploy.

   Figure 4–4 shows how the Get Cross Sell Offer Advisor retrieves an offer from the Cross Selling Offer choice group, based on the performance goal Cost.

Figure 4–4 Get Cross Sell Offer Advisor Retrieving an Offer from Cross Selling Offer Choice Group



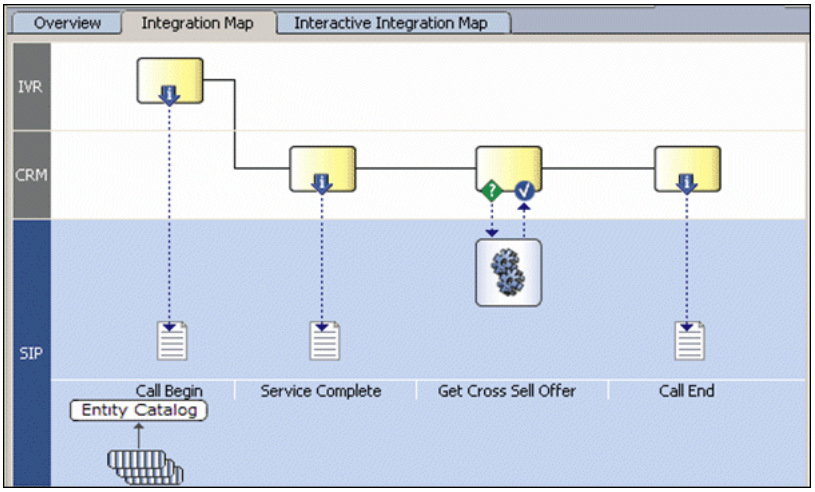
4.8 Viewing the Integration Map

To view the Integration Map in Decision Center:

1. Open Decision Center by opening a Web browser and going to the URL `http://`server_name`:8080/ui`. Log in using the default administrator credentials you created during installation. Real-Time Decision Server must be started for Decision Center to run.
2. Click **Open an Inline Service**.
3. Select the **Tutorial** Inline Service.

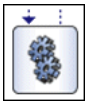
4. On the left-hand tree, click the root node **Tutorial**. In the right pane, on the **Definition** tab, click to view the **Integration Map** subtab. You should see something similar to the map shown in Figure 4–5.

Figure 4–5 Tutorial Integration Map



The symbols shown in Table 4–4 are used on the Integration Map to indicate integration points, processing, entities, and information flow.

Table 4–4 Integration Map Symbols

Symbol	Significance
	Processing on the Real-Time Decision Server
	Advisor call
	Information provided to Real-Time Decision Server
	Informant Call

4.9 Testing the Advisor

To test the Advisor:

1. In Decision Studio, use the Test View to send a request integration point. Select the **Service Complete** Informant and fill in some values for the parameters. For example: **customerId** = `7`, **agent** = `John`, **length** = `21`, **reason code** = `18` (others: `17`, `19`, or `20`).
2. Click the **Send** icon:

Then, confirm in the **Log** subtab that the message was sent. This Informant call creates a new session based on the customer ID and registers the customer's call reason, agent's name, and call length.

3. Now select the **Get Cross Sell Offer** Advisor, leaving the **customerId** as it is, as we want to continue with the same session. Click **Send**.

   The selected offer and its attributes are returned and displayed in the **Response** pane in the Test View.

   In the **Log** subtab in the Test View, for **customerId = 7**, you should see something similar to the following:

   ```
   00:24:40,764 Integration Point - Get Cross Sell Offer
   00:24:40,764   Customer age = 38
   00:24:40,764   Offer presented: 'Credit Card'
   ```

4. Repeat steps 1 to 3 with different values for **customerId** and other parameters. Notice that the Credit Card offer is returned if the customer's age is 40 or below, and the Savings Account offer is returned for all other ages. This is expected because so far, we have only optimized on the Cost performance metric, thus the lowest cost offer is either Savings Account or Credit Card, depending on the customer's age (see the Credit Card Score scoring rule in "Scoring the Choices" on page 4-3).

5. In the **Trace** subtab of the Test View, you will find a description of the sequence taken to arrive at the offer, from determining which offers are eligible to computing scores for each offer, and finally choosing the offer that met the performance goal (minimize Cost).

5

Closing the Feedback Loop

In the previous chapter, we added an Advisor that returns an offer to the CRM application so the call center agent can present it to the customer. Once presented to the customer, we want to track whether the customer has accepted the offer and thus close the loop on the offer presentation/acceptance process. The feedback loop can be closed in different ways and at different times. It is not unusual to know the results only days or weeks after a decision or offer is made. Even then, in many cases, only the positive result is seen, but not the negative. Feedback can come directly from customers, from the agents handling the call, from operational systems that handle service, fulfillment or billing, or even from batch processes.

The way the feedback loop is closed with an Inline Service is by notifying the Real-Time Decision Server through the use of Informants.

This chapter contains the following topics:

5.1 Using Events to Track Success

In most cases, there are different *events* in the lifetime of an offer that are interesting from the point of view of tracking success. For example, the events in the life of a credit card offer may be:

- Offer presented
- Customer showed interest
- Applied for the card
- Received the card
- Used the card

An argument could be made that only when the customer uses the credit card is there any real success. The goal is to bring more customers that not only show interest, apply and get the card, but for them to also use it, as card usage is what brings revenue to the company.

Usually, it is easier to track events that are closer to the presentation of the offer. For example, if an offer is presented in the call center by an agent, the agent can gauge the degree of interest shown by the customer. For an offer presented in a Web site, a click-through may be the indicator of interest.

Events further down the life of an offer may be much more difficult to track and decide on the right offer. Therefore, it is not unusual to begin a project having only the

immediate feedback loop closed, and adding events further down the road as the system matures. Nevertheless, even with only immediate feedback, Oracle RTD can provide significant lift in marketing decisions.

This section contains the following topics:

5.1.1 Defining Events in Choice Groups

Events are defined at the **Choice Group** level. While they can be defined at any level in the hierarchy, they are usually found at the highest level, close to the root.

We will define two events, one to represent the fact that an offer was **presented** to the customer, and the other to represent the fact that the offer was **accepted**. For the tutorial, we will assume that every offer selected as a result of the Advisor will be presented, and that the acceptance of offers is known immediately.

To define events in a choice group:

1. In the Inline Service Explorer, under **Choices**, double-click the Choice Group **Cross Selling Offer**.
2. Select the **Choice Events** tab. Click **Add** to add two events, one named `presented` and the second named `accepted`. Note that these event names are simply labels and do not correspond to any internal state of the offer. These events will be used in a Choice Event Model (described in the next section), where these event names will take on meaning.
3. For each event, set the **Statistic Collector** to **Choice Event Statistic Collector** using the drop-down list. This is the default statistics collector. This will provide for statistics gathering regarding each of the events.
4. Make sure that **Event History (days)** is set to **Session Duration**.
5. Leave the **Value Attribute** empty.

   This is used for the automatic computation of the event. In this tutorial, we will be causing the events to be recorded from the logic of the feedback Informant.
6. Choose **File > Save All**.

5.1.2 Defining a Choice Event Model

Events are defined and are ready to have statistics tracked. In addition to tracking statistics, we are interested in having a self-learning-model learn about the correlations between the characteristics of the customers, calls and agents, and the success or failure of offers. This knowledge is useful in two ways:

- It is useful for providing insight and understanding to the marketing and operations people.

- It is useful to provide automatic predictions of the best offer to present in each situation.

In this tutorial, we will show both usages.

To define a choice event model:

1. In the Inline Service Explorer, right-click the **Models** folder and select **New Choice Event Model**. Call the new model `Offer Acceptance Predictor` and click **OK**.
2. In the Editor, deselect **Default time window** and set **Time Window** to a week.
3. Under **Choice Group**, select **Cross Selling Offer**.

   This is the group at the top of the choice hierarchy for which we will track offer acceptance using this model.
4. Under **Base Event**, select **presented**. Recall that you had defined these event names in the choice group in the previous section.

   This is the event from which we want to measure the success. We want to track whether an offer was accepted after it was presented.
5. In **Positive Outcome Events**, click **Select**, choose **accepted**, and click **OK**. For the tutorial, this is the only positive outcome. If more events were being tracked, we would add them here also.
6. Optionally, you can change the labels to be more offer-centric.

5.1.3 Additional Model Settings

There are other settings that are useful for Choice Event Models. Using the Attributes tab, you see there are two main settings: partitioning attributes and excluded attributes. The following sections describe these and other settings.

This section contains the following topics:

5.1.3.1 Partitioning Attributes

Partitioning attributes are used to divide the model along strong lines that make a big difference. For example, the same offer is likely to have quite different acceptance profiles when presented in the Web or the call center, thus the presentation channel can be set as a partitioning attribute.

You can have more than one partitioning attribute, but you should be aware that there may be memory usage implications. Each partitioning attribute multiplies the number of models by the number of values it has. For example, a model having one partitioning attribute with three possible values and another with four possible values will use twelve times the memory used by a non-partitioned model. Nevertheless, do use partitioning attributes when it makes sense to do so, as it can significantly improve the predictive and descriptive capabilities of the model.

5.1.3.2 Excluded Attributes

Sometimes, it does not make sense to have an attribute be an input to a model. For example, we saw in the Reason Analysis model (as described in Section 3.1.2, "Viewing Analysis Results in Decision Center") that having the reason code as an input created a

correlation between reason code and the call reason choices. This relationship was entirely expected due to the logic we had written in Section 2.9.4, "Adding Logic for Selecting Choices." Since this correlation was artificial and did not offer insight, we excluded reason code from the model.

It should be noted that the reason code could be an important factor for other models and should not be excluded. For example, in the Offer Acceptance Predictor model, we would be very interested to see if offer acceptance was correlated with the reason code.

5.1.3.3 Learn Location

The **Learn Location** tab has the settings for the location in the process where model learning happens. The default, **On session close**, is a good one for most cases. Learning on specific Integration Points may be useful when it is desired to learn from more than one state in a session.

5.1.4 Remembering the Extended Offer

The choice event model is complete and it is ready to be used. In order to feed it with the right information, we need to complete the logic for closing the loop.

In order to have available which offer was extended, we will remember the offer ID in the session. This is not absolutely necessary, as the front-end client could remember that, but here we do not want to make any assumptions about the capabilities of the front end. We will just use a simple String attribute to remember the offer; in more complex cases we would use an array to remember many choices.

To remember the extended offer:

1. In the Inline Service Explorer, double-click **Session** under **Entities**.
2. Click **Add Attribute**, then add an attribute named `Offer Extended`.
3. Enter a description. Deselect **Show in Decision Center** and **Use for Analysis**. Click **OK**.

   We do so because for now, we will treat this as an internal variable, not to be seen by the business users.
4. In the Inline Service Explorer, double-click **Get Cross Sell Offer** under **Integration Points > Advisors**.
5. In the **Asynchronous Logic** tab, update the existing code by adding several lines to record the **presented** event and to set the OfferExtended session attribute with the value of the choice id. The completed code should be as follows:

```
logInfo("Integration Point - Get Cross Sell Offer");
logInfo("  Customer age = " + session().getCustomer().getAge() );
// 'choices' is array returned by the 'Select Offer' decision
if (choices.size() > 0) {
  //Get the first offer from array
  Choice offer = choices.get(0);
  //For the selected offer, record that it has been 'presented'
  offer.recordEvent("presented");
  //Set the session attribute 'OfferExtended' with the offer's ID.
  session().setOfferExtended(offer.getSDOId());
  logInfo("  Offer presented: '" + offer.getSDOLabel() + "'");
}
```

   This will assign the SDOId of the selected choice to the OfferExtended attribute of the session entity. The SDOId is a unique identifier. Every object in an Oracle RTD

configuration has a unique SDOId. It will also record the Presented event for the selected offer. Note the event name is in lowercase and corresponds to the choice event id for Presented. To see the id, go to Inline Service Explorer, expand **Choices**, double-click on **Cross Selling Offer**, click on the **Choice Events** tab, and click the label/id Toggle icon:

At this point of the decision, the session knows which offer has been chosen to be presented to the customer by the call center agent (through the Get Cross Sell Offer Advisor). We do not yet know the response from the customer. The response will be sent through a feedback Informant described in the next section.

5.1.5 Creating the Feedback Informant

This Informant provides Oracle RTD with the information needed to determine the result of the offer selection decision.

To create the feedback Informant:

1. In the Inline Service Explorer, expand **Integration Points**, right-click the **Informants** folder, and select **New Informant**. Call the Informant `Offer Feedback`.
2. In the Editor, type a description. Under **External System**, select **CRM**. Under **Order**, enter `4`.
3. To add a session key to the Offer Feedback Informant, click **Select** next to the **Session Keys** list. Select **customerId** and click **OK**.
4. Click **Add** to add an incoming parameter. Call it `Positive`.
5. Select the data type **String** if is not already selected and click OK.

   Leave it unmapped. We do not need to map it to any session attribute because we will use this argument immediately to determine whether the offer was accepted or not. A yes value will be used to indicate offer acceptance.
6. Using the **Logic** tab, enter the following under **Logic** to record the acceptance event when appropriate.

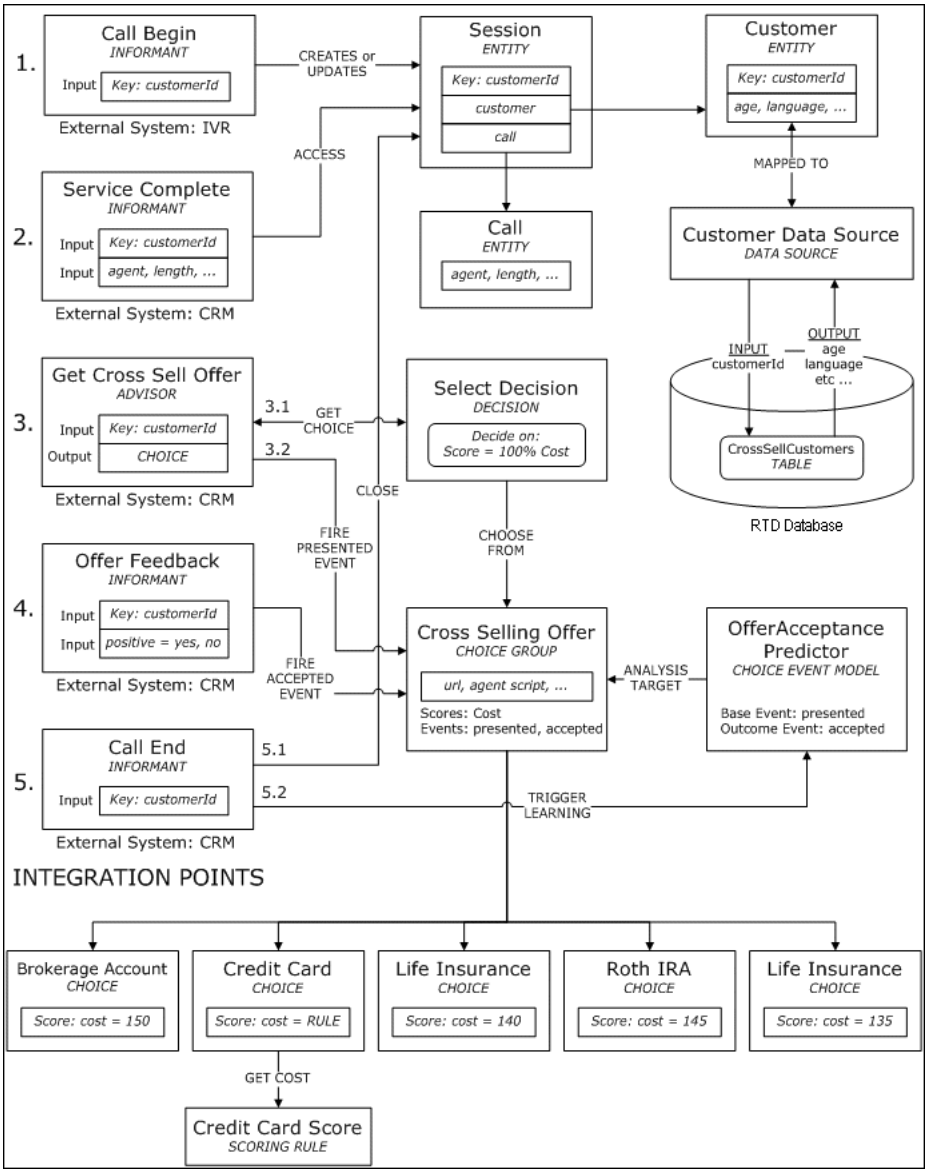
```
logInfo("Integration Point - Offer Feedback");
//"yes" or "no" to accept offer.
String     positive = request.getPositive();
positive = positive.toLowerCase();

//Get the offer id from session attribute 'OfferExtended'
String extendedOfferID = session().getOfferExtended();
if (extendedOfferID != null) {
   //Get the offer from choice group 'Cross Selling Offer'
   Choice offer = CrossSellingOffer.getChoice(extendedOfferID);
   if (offer != null){
     String offerId = offer.getSDOId();
     //If response is "yes", then record the offer as accepted.
    if (positive.equals("yes")) {
      offer.recordEvent ("accepted");
      logInfo("  Offer '" + offer.getSDOLabel() + "' accepted");
    }
  }
}
```

7. Save all and redeploy the Inline Service. On the **Deploy** dialog, check **Terminate Active Sessions (used for testing)**.

   The following diagram shows how the Get Cross Sell Offer Advisor retrieves and presents an offer, and then the Offer Feedback Informant accepts or rejects the offer. When the Call End Informant closes the session, the Offer Acceptance Predictor model is updated with the offer Presented/Accepted events.

Figure 5–1 Tutorial Inline Service Objects: Advisor/Informant Flow



5.1.6 Testing the Feedback Informant

In order to test the Offer Feedback Informant, we need to first call the Get Cross Sell Offer to retrieve and present an offer.

To test the feedback Informant:

1. In Test View, select the Integration Point **Get Cross Sell Offer**. Enter a value for the **customerId**, such as `10`.

2. Click the **Send** icon:

Then, confirm in the **Response** subtab that an offer was retrieved. In the **Log** subtab, you should see something similar to the following:

```
00:45:28,466 Integration Point - Get Cross Sell Offer
00:45:28,466   Customer age = 38
00:45:28,466   Offer presented: 'Credit Card'
```

Note that even if you tried different values for **customerId**, the offer presented is always Savings Account or Credit Card. This is because we have only one performance goal at this point - to minimize cost, and Savings Account or Credit Card is the lowest cost, depending on the age of the customer.

3. Now select the **Offer Feedback** Informant from the **Integration Point** drop-down list. Leave the **customerId** as it is, as we want to continue with the same session. Enter a value for input **Positive**, such as `yes`.

4. Click **Send** and confirm in the **Log** subtab that the offer retrieved by the Get Cross Sell Offer Advisor is accepted. You should see something similar to the following:

```
00:46:01,418 Integration Point - Offer Feedback
00:46:01,418   Offer 'Credit Card' accepted
```

5. Change the input **Positive** value to `no` and re-Send the **Offer Feedback** Informant. The **Log** subtab will look something similar to the following:

```
00:47:31,494 Integration Point - Offer Feedback
```

5.1.7 Updating the Load Generator Script

We will now update the Load Generator script to include calls to the GetCrossSellOffer Advisor and the OfferFeedback Informant. Note that these integration point calls should take place after the ServiceComplete Informant but before the CallEnd Informant, which closes the session. The logic is: 1) call begins, 2) regular service is complete - we record and analyze call reasons using the ReasonAnalysis model, 3) agent presents a cross sell offer to customer, based on lowest Cost goal, 4) we record if customer has accepted offer, 5) call/session ends, OfferAcceptancePredictor model learns on offer presented/accepted.

To add the GetCrossSellOffer Advisor to the Load Generator script:

1. Open Load Generator by running *`RTD_HOME`*`\scripts\loadgen.cmd`. Then, open the previous script.

2. Select the **Edit Script** tab, then right-click the left tree view and select **Add Action**. The action is of type **Message** and the Integration Point name should be **GetCrossSellOffer**.

3. In **Input Fields**, right-click and chose **Add item** to add an input field. Click in the space under **Name** and type **customerId**, then press **Enter**.

4. Click **Variable** for the input field and use the drop-down list to choose the matching variable, **var_customerId** (see Section 3.1.1, "Creating the Load Generator Script" for more information). Mark **customerId** as a session key by selecting **Session Key**.

5. After we add this action to the script, it is placed at the bottom of the actions list. We need to adjust the order so that GetCrossSellOffer is called after ServiceComplete. In the left side of the **Edit Script** tab, right-click

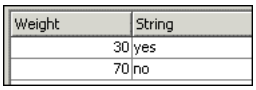
GetCrossSellOffer and select **Move Up** or **Move Down** so that the order is CallBegin, ServiceComplete, GetCrossSellOffer, and CallEnd.

6. Save the Load Generator script.

To add the OfferFeedback Informant to the Load Generator script:

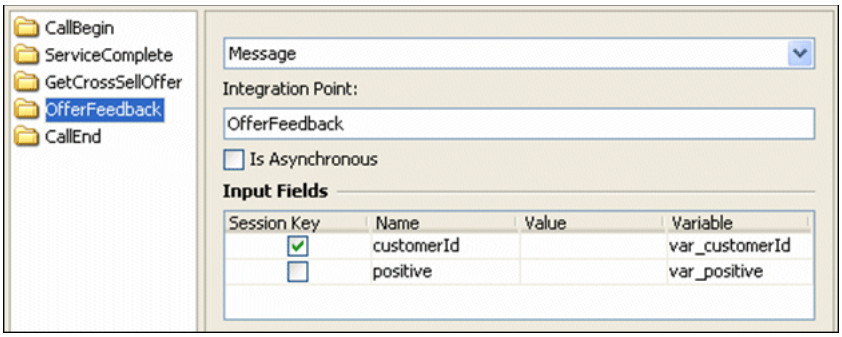
1. Before we add the call to OfferFeedback in the **Edit Script** tab, we need to create a new variable in the **Variables** tab. Recall in the definition of the OfferFeedback Informant, the parameter **positive** is used to indicate offer acceptance. In Load Generator, we will set the value of this parameter to randomly be **yes** 30% of the time and **no** 70% of the time. We do this by using a weighted string array.
2. In the Variables tab, in the left side, right-click on the folder **Script** and select **Add Variable**. Enter `var_positive` for **Variable name**, then set the **Contents** type to **Weighted String Array**. Add two items to the array (right-click in the space below the content type and select **Add Item**). For the first item, double-click in the **Weight** cell to make it editable and type the value `30`, and in the corresponding String cell, type the value `yes`. The second item should have the weight value of `70` and string value of `no`. Note that the weights do not have to add up to 100, because they are normalized automatically. Weight values of 6 and 14 would have the same desired effect.

Figure 5–2 Weighted String Array Variable



3. Select the **Edit Script** tab, then right-click the left tree view and select **Add Action**. The action is of type **Message** and the Integration Point name should be `OfferFeedback`.
4. In **Input Fields**, right-click and chose **Add item** to add an input field. Click in the space under **Name** and add **customerId**. In the **Variable** column, select the matching variable, **var_customerId** (see Section 3.1.1, "Creating the Load Generator Script" for more information). Mark **customerId** as a session key by selecting **Session Key**.
5. Again in **Input Fields**, right-click and chose **Add item** to add an input field. Click in the space under **Name** and add `positive`. In the **Variable** column, select the matching variable, **var_positive**.
6. After we add this action to the script, it is placed at the bottom of the actions list. We need to adjust the order so that OfferFeedback is called after GetCrossSellOffer. In the left side of the **Edit Script** tab, right-click **OfferFeedback** and select **Move Up** or **Move Down** so that the order is CallBegin, ServiceComplete, GetCrossSellOffer, OfferFeedback, and CallEnd.

Figure 5–3 Adding the OfferFeedback Informant to the Load Generator Script

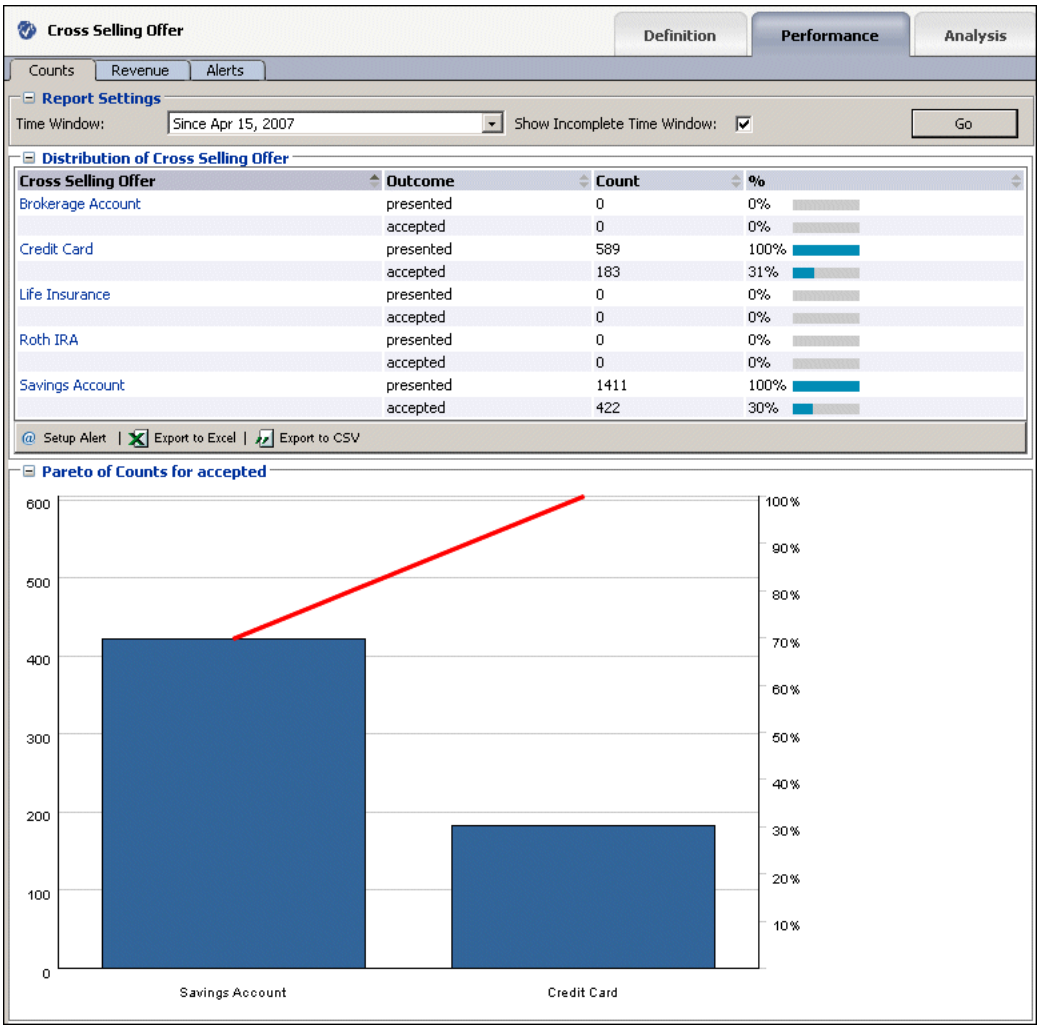


7. Save the Load Generator script.

You can run the Load Generator script at this point. Again, it is recommended that you remove existing data before running the script so the results are not mixed with older data - see Section 3.2, "Resetting the Model Learnings" for information about how to do this.

If you do run the Load Generator script, you can view the results in Decision Center. Log in to Decision Center and click the **Cross Selling Offer** choice group to show the results of the Offer Acceptance Predictor model. Click the **Performance** tab and then the **Counts** subtab. The distribution of offers and the Pareto graph should look like the one shown in Figure 5–4.

Figure 5–4 Decision Center Performance Counts for Cross Selling Offer



Notice that only two offers were presented - Credit Card and Savings Account, and each one had an acceptance rate of about 30%. This is entirely expected due to the logic we have set up so far: 1) Only one performance goal - minimizing Cost - was to be met, and the Cost is lowest for Savings Account or Credit Card, depending on the age of the customer (see Section 4.4, "Scoring the Choices"). In the Load Generator script, we specified that 30% of the time, a positive response to an offer is registered through the OfferFeedback Informant. If we drill down into the analysis reports of individual offers, we will not see much correlation between the acceptance of an offer and session attributes. This is because we are using random customer profile data and forcing the acceptance rate to be 30%, regardless of customer or other attributes (such as call length, call agent name, call reason, and so on).

We have now demonstrated how to use performance goal to decide which offer to present and how to use a choice event model to record how often presented offers are accepted. We have only used the model for analysis so far. In the next section, we will add a second performance goal (Maximize Revenue) and use what the model has learned in order to influence which offer is to be presented. We will also introduce an artificial bias that increases the likelihood of customers who have two or more children to accept the Life Insurance offer if it is presented. We will then be able to see how the bias affects the model results.

5.2 Using the Predictive Power of Models

The model we have created learns the correlations between the characteristics of the customers, the call characteristics, and the cross selling results. This model can be used in a predictive fashion, to predict the likelihood an offer will be accepted. We can use the likelihood information to adjust the values of offers when deciding which offer to present. For example, if offer A is twice as likely to be accepted as offer B, it is reasonable to favor offer A when an offer is picked to be presented. In this section, we will introduce a second performance goal - Maximize Revenue - whose value/score is calculated as the product of the **likelihood of acceptance** and the base **Revenue**.

For example, if the base Revenue for the Brokerage Account offer is $300, and the likelihood of acceptance is 30% (0.3), then the Maximize Revenue score is $300 x 0.3 = $90. If the base Revenue for the offer Life Insurance is $185, but the likelihood of acceptance is 60% (0.6), then the Maximize Revenue score is $185 x 0.6 = $111. Even though Brokerage Account had a higher base Revenue value, the Life Insurance offer would be favored because its Maximize Revenue score is higher.

Note that we will be choosing the offer to present based on both the Cost and Maximize Revenue performance goals, so in the previous example, Brokerage Account may still win if the weighted total of its Cost and Maximize Revenue is higher than the total for Life Insurance.

We will begin this section by adding a base Revenue, then adding the second performance goal Maximize Revenue. Then we will set the score for the Maximize Revenue goal to Revenue multiplied by the likelihood of acceptance. Afterwards, we will update the Select Offer decision so that both Cost and Maximize Revenue goals are considered when choosing an offer to present. Finally, in the Offer Feedback, we will add logic to introduce offer acceptance bias for customers with a certain profile who are presented the Life Insurance offer.

This section contains the following topics:

5.2.1 Adding a Base Revenue Choice Attribute

To add a base Revenue choice attribute:

1. In the Inline Service Explorer, under **Choices**, double-click the **Cross Selling Offer** Choice Group. In the **Choice Attributes** tab, click **Add**.
2. Set the name of this attribute to **Revenue** of data type **Integer**. Make sure the **Overridable** option is selected, as we will assign a different value for each of the offers, then click **OK**.

3. For each choice under the **Cross Selling Offer** Choice Group, set the value of the **Revenue** attribute as shown in Table 5–1.

Table 5–1 Revenue Value for Choices

Choice Name	Revenue Value
Brokerage Account	300
Credit Card	205
Life Insurance	185
Roth IRA	190
Savings Account	175

5.2.2 Adding a Second Performance Goal (Maximize Revenue)

Earlier in this tutorial, we defined a Cost performance goal. Now we will add a second performance goal called Maximize Revenue. We will use the likelihood of acceptance and the base Revenue of the choice in calculating the score for this new performance metric. The formula for this is: (Revenue) * (likelihood of acceptance) = potential revenue score.

To add a second performance goal:

1. In the Inline Service Explorer, double-click **Performance Goals** to open the editor. Click **Add** to add a Performance Metric. Name the metric **Maximize Revenue**, then click **OK**.
2. In **Optimization**, choose **Maximize** and make the metric **Required**. Since $1 of cost equals $1 of revenue, the Normalization Factor does not need to be adjusted.
3. Next, we need to add this metric to the **Cross Selling Offer** Choice Group. In the Inline Service Explorer, double-click **Cross Selling Offer**. In the **Scores** tab, click **Select Metrics**. In the **Select** dialog, select **Maximize Revenue** and click **OK**.

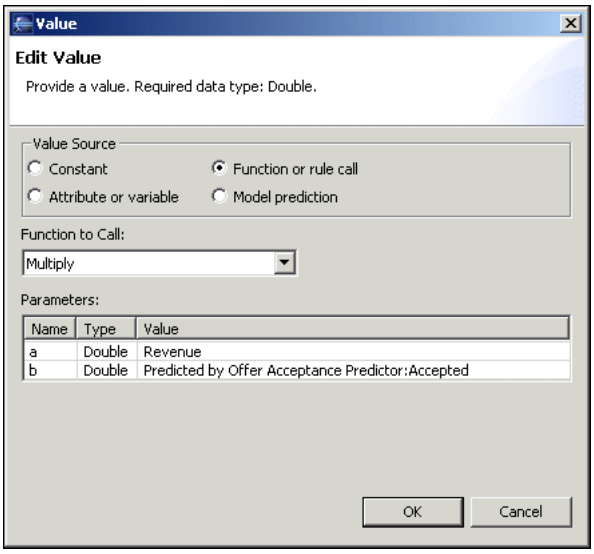
5.2.3 Calculating Score Value for the Maximize Revenue Performance Goal

To calculate the score value for the Maximize Revenue goal, we need the base Revenue and the likelihood of acceptance value as determined by the Offer Acceptance Predictor choice event model. This can be retrieved using the edit value dialog by changing the value source to Model Prediction.

To calculate the score value for the Maximize Revenue goal:

1. In the Inline Service Explorer, under **Choices**, double-click the **Cross Selling Offer** choice group. In the **Scores** tab, click in the **Score** column for the Maximize Revenue metric, then click the ellipsis to bring up the Edit Value dialog.
2. For the **Value Source**, select **Function or rule call**. Under **Function to Call**, choose the function **Multiply**. In the Parameters table, click in the **Value** cell for parameter **a**. Click the ellipsis and choose **Attribute or variable**, then expand the **Choice** folder, select **Revenue**, and click **OK**. In the Parameters table, click in the **Value** cell for parameter **b**. Click the ellipsis and choose **Model Prediction**. Choose the likelihood predicted by the Offer Acceptance Predictor model and the Accepted event, then click **OK**. Click **OK** again in the Edit Value dialog.

Figure 5–5 Edit Value Dialog for Maximize Revenue Score



The actual value of the likelihood is from 0 to 1, 1 being 100% likely to accept. It is also possible for the value to be NaN (Not a number), which means the model did not have enough data to compute a likelihood value. In such situations, the Maximize Revenue score cannot be computed and the offer selection by the Select Offer decision will be based on built-in score comparison logic, which depends on whether the score is or is not required.

3. By defining the score for Maximize Revenue on the choice group level, all of the choices within this group will inherit the definition and apply choice-specific values for Revenue and likelihood of acceptance during run time.

5.2.4 Updating the Select Offer Decision to Include the Second Performance Goal

We have so far defined a new performance metric and how to calculate its value. We will now update the Select Offer decision to consider both performance metrics when choosing an offer to present.

To update the Select Offer Decision:

1. In the Inline Service Explorer, expand the **Decisions** folder and double-click **Select Offer**.
2. In the **Selection Criteria** tab, you should see only one Performance Goal in the Priorities for the "Default" Segment table, **Cost**, with a **Weight** value of 100%. Click **Goals**, then select the goal **Maximize Revenue** and click **OK**.

   The priorities table now shows two performance goals, each with a Weight of 50%. The default is to evenly split weighting between all selected metrics. If you wanted the Maximize Revenue performance goal to take precedence over Cost, you could adjust the percentages so that it had more weight. We will use the default Weight of 50% in this tutorial.

   Table 5–2 shows an example of how the Select Offer decision calculates a total score for a particular offer, assuming the offer's Cost score is 150 and its Maximize Revenue score is 215.

Table 5–2 Calculating a Total Score for an Offer

Performance Goal	Score	Weight	Max/Min	Norm.	Weighted Score
Cost	150	50%	Min	1	-75
Maximize Revenue	215	50%	Max	1	107.5

The **Total Score** based on the values in Table 5–2 is **32.5**. The weighted Cost score is negative because the optimization is Minimize. The total score of the offer is the sum of the two weighted scores. The total score is calculated for each offer, and the offer with the highest value will be selected.

5.2.5 Adding a Choice Attribute to View Likelihood of Acceptance

To view the value of the likelihood of acceptance, we can add a choice attribute and display it through logInfo or in the Response tab of Test view.

To add a choice attribute:

1. In the Inline Service Explorer, under Choices, double-click the **Cross Selling Offers** choice group. In the **Choice Attributes** tab, click **Add** to add an attribute. In the properties dialog, set the **Display Label** to **Likelihood Of Acceptance**. Set the **Data Type** to **Double**.
2. Deselect the option **Overridable**, because all choices in this choice group will use the same definition for this attribute. Then, select the option **Send to client** and click **OK**.
3. In the **Value** column for the **Likelihood Of Acceptance** attribute, click the ellipsis to set its value. In the Edit Value dialog, set the **Value Source** to **Model prediction**. Choose the **Offer Acceptance Predictor** model and the **Accepted** event, then click **OK**.
4. Save all changes to the Inline Service.

5.2.6 Checking the Likelihood Value

To view values of the likelihood, add a logInfo statement in the Get Cross Sell Offer Advisor, as follows:

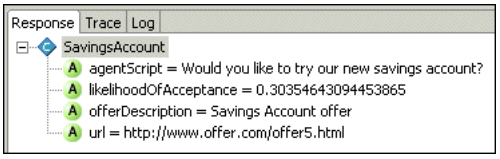
1. In the Inline Service Explorer, double-click the **Get Cross Sell Offer** folder under **Integration Points > Advisors**.
2. In the **Asynchronous Logic** tab, update the existing code by adding several lines to print the value of the Likelihood Of Acceptance. The completed code should appear as follows:

```
logInfo("Integration Point - Get Cross Sell Offer");
logInfo("  Customer age = " + session().getCustomer().getAge() );
// 'choices' is array returned by the 'Select Offer' decision.  The
// name 'choices' was set (and can be changed) in the 'Choice Array'
// text box in the 'Select Offer' decision's 'Pre/Post Selection
// Logic' tab.
if (choices.size() > 0) {
  //Get the first offer from array
  Choice offer = choices.get(0);
  //For the selected offer, record that it has been 'presented'
  offer.recordEvent("presented");
  //Set the session attribute 'OfferExtended' with the offer's ID.
  session().setOfferExtended(offer.getSDOId());
```

```
  logInfo("  Offer presented: '" + offer.getSDOLabel() + "'");
  //Cast selected offer to type CrossSellingOfficeChoice -
  //the base Choice type of choice group 'Cross Selling Offer'
  CrossSellingOfferChoice cso = (CrossSellingOfferChoice) offer;
  logInfo("  Likelihood of Acceptance = " + cso.getLikelihoodOfAcceptance());
}
```

3. To see the effect of the changes to the Advisor, save all and deploy the Inline Service.
4. In Test view, select the **Get Cross Sell Offer** Integration Point and input a value for **customerId**, such as `8`. Click **Send**. In the **Response** subtab in Test View, you should see something similar to the image shown in Figure 5–6.

Figure 5–6 Response Subtab in Test View



In the Log subtab, you should see something similar to the following:

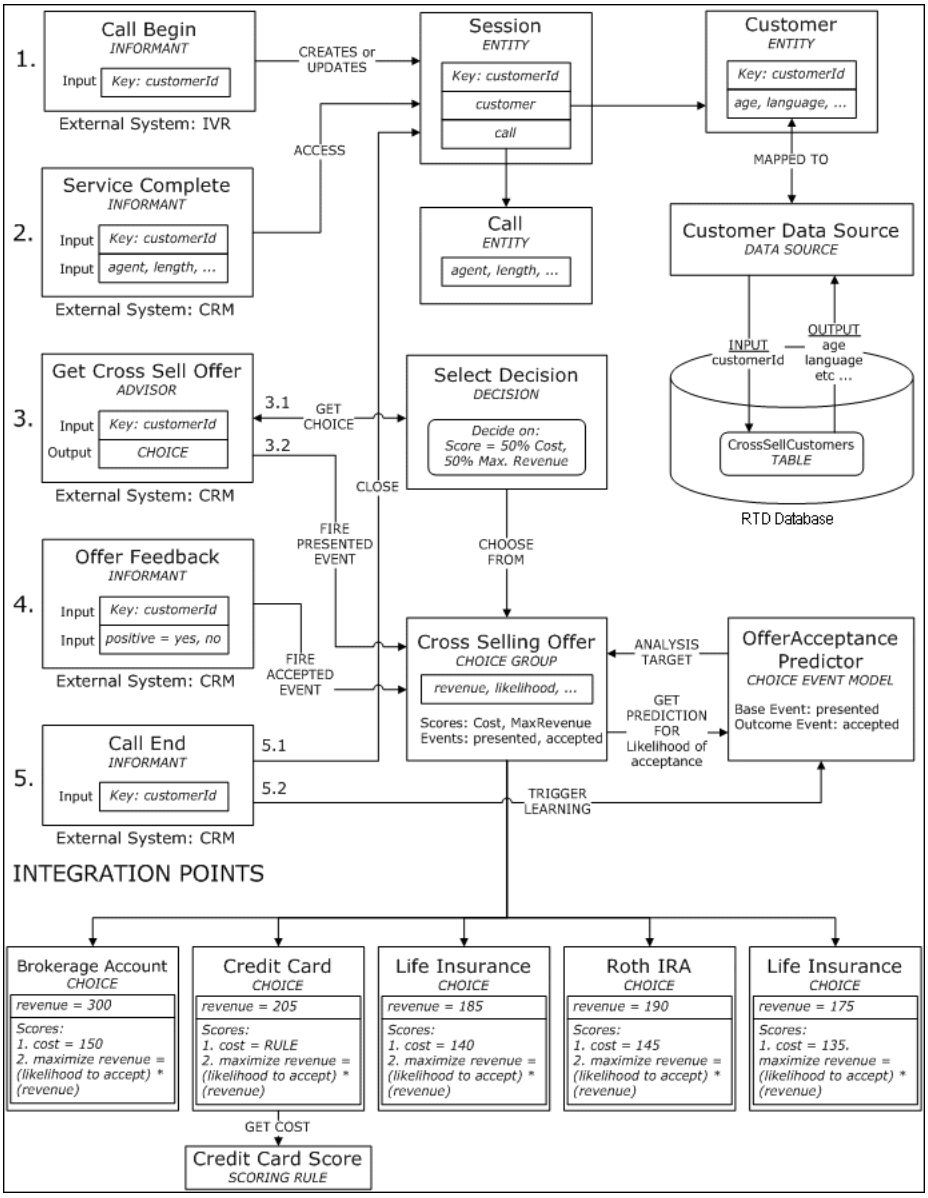
```
14:07:37,908 Integration Point - Get Cross Sell Offer
14:07:37,908   Customer age = 57
14:07:37,908   Offer presented: 'Savings Account'
14:07:37,908   Likelihood of Acceptance = 0.30354643094453865
```

If you are getting a value of NaN (Not A Number) for Likelihood Of Acceptance, this means the model did not have enough data to compute the likelihood value for this offer. The number of iterations necessary to reach model convergence (likelihood numbers no longer NaN) depends on the application and quality of the data.

In our case, we had imposed a definite offer acceptance rate of about 30% (see Section 5.1.7, "Updating the Load Generator Script"), and since we are using random customer profile data, the Offer Acceptance Predictor model should converge quickly and be able to compute likelihood of acceptance values within just a few hundred iterations. Before the model has reached convergence, the offer selection process is based on built-in score comparison logic, which depends on whether the score is required.

The following diagram shows the Get Cross Sell Offer Advisor retrieving an offer from the Cross Selling Offer choice group, where the total score of each offer is a weighted sum of two scores - Cost and Maximize Revenue.

Figure 5–7 Tutorial Inline Service Objects: Weighted Sum



5.2.7 Introducing Offer Acceptance Bias for Selected Customers

Earlier in the Offer Feedback Informant, we specified whether to accept a presented offer through the Positive Informant parameter. We then updated the Load Generator script so that when this Informant is called, we pass the value yes to the parameter Positive 30% of the time (see Section 5.1.7, "Updating the Load Generator Script"). This percentage did not depend on any customer profile data - any presented offer had a 30% chance of being accepted by any customer.

If we run the Load Generator script at this point, the models would not show any strong correlation between customer attribute to the acceptance of the offer. We will introduce an artificial bias in the Offer Feedback Informant logic which will always record positive offer acceptances for customers who have two or more children and who were presented the Life Insurance offer. This logic is in addition to the default acceptance rate (as defined in the Load Generator script) and will skew the acceptance rate for the Life Insurance offer to more than 30%. In Decision Center, we will be able

to see clear correlations between the number of children and the acceptance rate of this offer.

To introduce the Offer Acceptance bias:

1. In the Inline Service Explorer, double-click **Offer Feedback** under **Integration Points > Informants**.
2. In the **Logic** tab, update the existing code by adding several lines to add offer acceptance bias for customers who have two or more children and who were presented the Life Insurance offer. The completed code should appear as follows:

```
logInfo("Integration Point - Offer Feedback");
//"yes" or "no" to accept offer.
String    positive = request.getPositive();
positive = positive.toLowerCase();

//Get the offer id from session attribute 'OfferExtended'
String extendedOfferID = session().getOfferExtended();
if (extendedOfferID != null) {
  //Get the offer from choice group 'Cross Selling Offer'
  Choice offer = CrossSellingOffer.getChoice(extendedOfferID);
  if (offer != null){
    String offerId = offer.getSDOId();
    //Introduce artificial bias for customers with 2 or more
    //children to always accept "LifeInsurance" if it was
    //selected after scoring.
    //If data source is Oracle, change the following method from
    //getNumberOfChildren() to getNumberofchildren()
    int numOfChildren = session().getCustomer().getNumberOfChildren();
    if ( numOfChildren >= 2 && offerId.equals("LifeInsurance")) {
       positive="yes";
    }
    //If response is "yes", then record the offer as accepted.
    if (positive.equals("yes")) {
      offer.recordEvent ("accepted");
      logInfo("  Offer '" + offer.getSDOLabel() + "' accepted");
    }
  }
}
```

3. Save all changes and deploy the inline service.

5.2.8 Running the Load Generator Script

In Section 5.1.7, "Updating the Load Generator Script," we updated the Load Generator Script to include the GetCrossSellOffer Advisor and the OfferFeedback Informant. At that point, the offer selection process was based on only one performance goal - to minimize Cost. We then added a second performance goal, Maximize Revenue, which uses predicted values of acceptance likelihoods as computed by the Offer Acceptance Predictor model. The offer selection process now depends on both performance goals. We have also introduced an artificial acceptance bias for customers who fit a certain profile, and who were presented the Life Insurance offer. We will now run the Load Generator script again to see the results.

To run the Load Generator script:

1. If you are using OC4J or WebLogic, open JConsole by running *JAVA_HOME*\bin\jconsole.exe. If you are using WebSphere, run the batch script you created during JConsole configuration. See *Oracle Real-Time Decisions Installation and Administration Guide* for more information about using JConsole.

2. Click the **Remote** tab. Then, enter the appropriate port number (typically `12345`) and the administrator credentials you created during installation and click **Connect**.
3. Click the **MBean** tab, then go to the **OracleRTD > InlineServiceManager > Tutorial > Development > Loadable** MBean.
4. Click the **Operations** tab, then use the **deleteAllOperationalData()** operation to remove all operational data, including the study, for this Inline Service.
5. Start Load Generator and open the Load Generator script previously defined. There should be no changes necessary.
6. Start the Load Generator script. After about 200 total finished scripts, click the **Pause** icon to temporarily stop sending requests to the server:

Then, view the server's output in the `server.log` file, which is in the *RTD_RUNTIME_HOME*\log directory.

You will see that the printed Likelihood Of Acceptance values are NaN for all sessions. This is an indication that the model has not yet learned enough data to be able to compute the likelihood of acceptance. Note that offers are still being presented despite the lack of likelihood values. Offers are being selected using built-in scores comparison logic.

7. Un-pause the Load Generator script and let it finish running for 2000 total finished scripts. In the server output, you should now see actual values for Likelihood Of Acceptance, varying around 0.3 for all offers except Life Insurance, which has higher values because of the bias introduced.
8. It is important to note that the model-predicted Likelihood Of Acceptance values for a given offer will differ for different customer profiles. For example, suppose we have two customers John and Tom, who only differ in the number of children they have. If we printed the Likelihood Of Acceptance values for the Life Insurance offer for these two customers (at a snapshot in time), we will see a higher value for Tom, as shown in Table 5–3. This is because Tom has three children, and is therefore more likely to accept the Life Insurance offer, if it is presented to him.

Table 5–3 Likelihood of Acceptance for Life insurance Offer

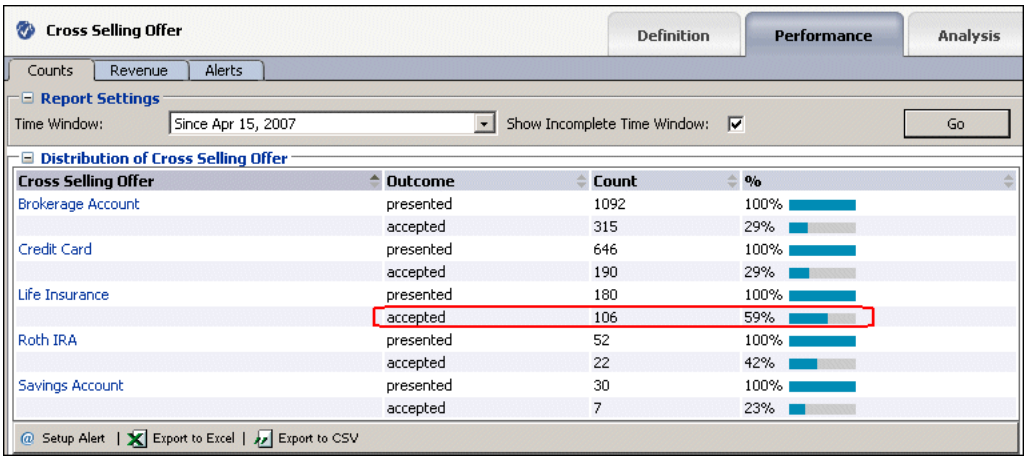
Customer	Number of Children	Likelihood of Acceptance for Life Insurance Offer
John Doe	0	.32
Tom Smith	3	.89

Since we determine which offer to present to the customer based on the combination of Cost and Maximize Revenue scores, and because Maximize Revenue depends on the model's predicted Likelihood Of Acceptance value for each offer, the Life Insurance offer will have a high Maximize Revenue value for customers with two or more children, and therefore for such customers, Life Insurance will be presented (and then accepted) far more frequently than other offers!

5.2.9 Studying the Results

To view the results of the Load Generator run, log in to Decision Center. Click the **Cross Selling Offer** Choice Group in the left navigation box. This will show the results of the Offer Acceptance Predictor model. Click the **Performance** tab and then the **Counts** subtab. You should see a table similar to the one shown in Figure 5–8.

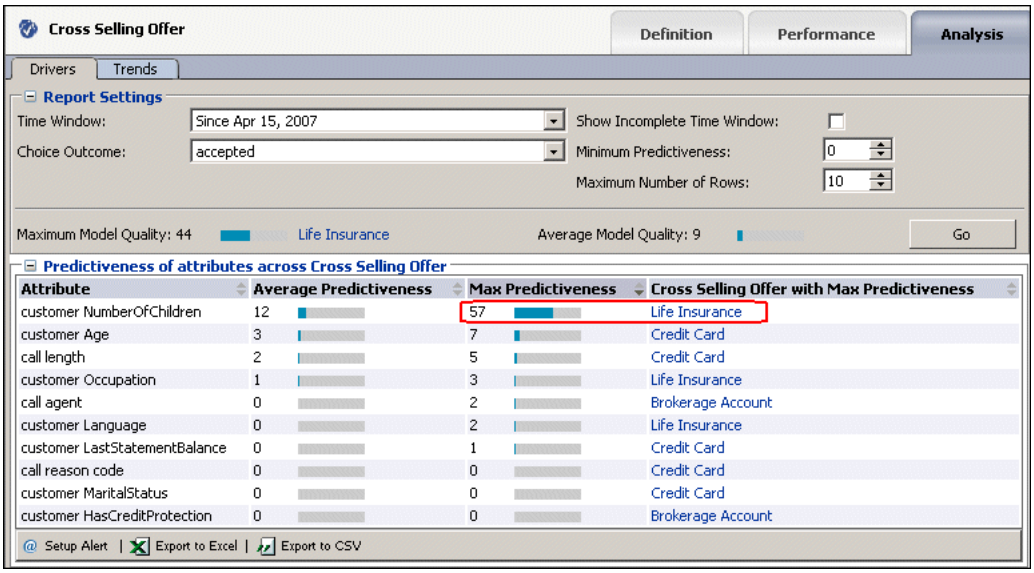
Figure 5–8 Performance Counts for Cross Selling Offer Choice Group



The Decision Center table shows the distribution of the offers - how many were presented and how many were accepted for each offer. Except for Life Insurance, all of the other offers had acceptance rate of about 30%, as shown in Figure 5–8. This is expected because of how we set up the Load Generator script (see Section 5.1.7, "Updating the Load Generator Script"). The acceptance rate for Life Insurance is higher than 30% because of the artificial bias we introduced in Section 5.2.7, "Introducing Offer Acceptance Bias for Selected Customers." The bias dictated that in addition to 30% of the customers accepting any offer, customers who had two or more children and were offered Life Insurance will always accept the offer.

Given the artificial bias, the model results should show that for the Life Insurance offer, the NumberOfChildren attribute will be an excellent predictor for whether or not the offer will be accepted. This is exactly what we see in the Decision Center reports: click the **Cross Selling Offer** Choice Group and click on the **Analysis** tab, then the **Drivers** subtab. In the **Report Settings** section, change the **Minimum Predictiveness** value to 0 and then click **Go**. You will see a list of attributes, ordered by the maximum predictiveness value. The highest value for Max Predictiveness should be for the NumberOfChildren attribute, since it is the only artificial bias we added. The corresponding offer should be Life Insurance, similar to the image shown in Figure 5–9.

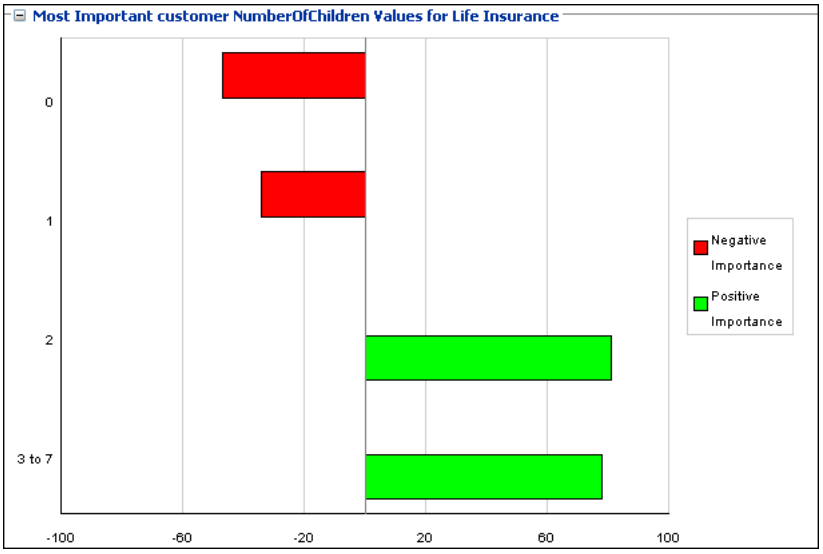
Figure 5–9 Cross Selling Offer Analysis Drivers



We can further analyze the importance of the NumberOfChildren attribute for the Life Insurance offer by viewing reports specific to this offer. In the navigation box in Decision Center, expand the **Cross Selling Offer** Choice Group and click the choice **Life Insurance**, then click the **Analysis** tab and finally the **Drivers** tab. This report shows the important drivers for acceptance of this particular offer (Life Insurance).

In the **Report Settings** section, change the **Minimum Predictiveness** value to `0` and then click **Go**. You will see a list of attributes, ordered by the **Predictiveness** value. The NumberOfChildren attribute should have the highest predictiveness value. Click the attribute name to display more detailed reports, one of which should look similar to Figure 5–10.

Figure 5–10 Life Insurance Offer Analysis Drivers



The graph shown in Figure 5–10 shows that for NumberOfChildren values of 2 and above, there is a strong positive correlation for offer acceptance. This means that the number of acceptances of this offer for these attribute values (2 or more) is much

higher than expected. Similarly, for values of 0 or 1, the correlation is also very strong, but is negative, meaning that customers with 0 children or 1 child did not accept Life Insurance as much as expected.

Part II

Integration with Oracle RTD

The chapters in Part II explain how to use the Java Smart Client, Java Smart Client JSP tags, and the .NET Smart Client to integrate with Oracle RTD. They also explain how to use the Oracle RTD Web services.

Part II contains the following chapters:

6

About Integrating with Oracle RTD

Oracle RTD features several robust and easy-to-use ways to integrate with enterprise operational systems:

- Smart Clients: For Java and .NET environments, these components manage communication to Integration Points on Real-Time Decision Server.
- Zero Clients: Access to Integration Points is available through Web services as a zero client approach.

This chapter, and the following chapters in Part II, outline how to use these ways to integrate with deployed Inline Services running on Oracle RTD.

See Part I, "Getting Started" for information about using Decision Studio to deploy Inline Services. For information about the integration APIs, see the Decision Studio online help.

Note: The following terms are referenced throughout the Oracle RTD documentation:

- *RTD_HOME*: This is the directory into which Oracle RTD is installed. For example, `C:\OracleBI\RTD`.
- *RTD_RUNTIME_HOME:* This is the application server specific directory in which the application server runs Oracle RTD.

For more information, see the section "About the Oracle RTD Run-Time Environment" in *Oracle Real-Time Decisions Installation and Administration Guide*.

This chapter contains the following topics:

6.1 Choosing the Best Means of Integration

Oracle Real-Time Decisions offers multiple means of integration. To choose the best means for your environment you should consider the platform you are working on, performance needs and the additional functionality offered by RTD Smart Client over other methods of integration.

This section contains the following topics:

6.1.1 About the Java Smart Client

The Oracle RTD Smart Client for Java is a component that allows easy, managed integration to deployed Inline Services for operational systems. If you are working in a Java environment, the Java Smart Client is the preferred means of integration. The Java Smart Client offers two important features above and beyond the other methods of integration: session key mapping (to facilitate HTTP session affinity management by an external load balancer) and default response handling.

The factory methods of the Java Smart Client interface take parameters representing the minimal information required to establish contact with a cluster of servers. After connecting, the component's full configuration is downloaded from the server. This way only a small set of parameters must be managed in the client application, while most of the component's configuration is centrally managed by the server's administration console.

The configuration information returned by the server to the client is shared by all the instances of the Smart Client created in the same Java virtual machine. There is a client-side class called a client-side dispatcher that manages this shared configuration and also manages session-affinity information used to dispatch requests to the correct server, based on session keys in the request.

The Java Smart Client is thread-safe, but for optimal performance a separate Java Smart Client should be created for each thread. Separate instances of the Java Smart Client share information and connections, so there is practically no penalty to having multiple instances.

Several factory methods are available to create a Java Smart Client. Most either directly or indirectly reference a properties file in the file system or on a Web server. The properties file supplies addresses for connecting to one or more servers in a single cluster as well as other properties that configure the connection to the server. Factory methods are also available to directly supply an HTTP URL and port or use a default address.

After the client's constructor communicates with one server and receives more complete configuration information, the detailed configuration is saved in a local file called the client configuration cache, where it can be accessed should the client restart when the server is unavailable. The configuration cache contains information such as the client's set of default responses for all integration points in all Inline Services. The client's configuration cache is updated automatically by the client whenever it changes in the server.

Part of the configuration information downloaded to a client from the server includes a set of default responses to use if the client loses contact with the server or the server fails to respond to an integration point request in a timely fashion. This maintains the Service Level Agreement (SLA) between Real-Time Decision Server and client application regardless of individual transactional availability.

These default responses are configured at the granularity of the individual integration points; each integration point relies on its own specialized default response. When any default responses are reconfigured on the server, the changes are propagated automatically to the client's out-of-band data, bundled together with normal integration point responses.

The Java Smart Client automatically keeps track of any HTTP cookies that are returned by the Web Container of Real-Time Decision Server. The next time the same Inline Service key is used in a request, its cookies are included in the HTTP request so that the external load balancer can route the request to the server instance that is already handling that Inline Service key.

To achieve clustering using other methods of integration, the application must track the Inline Service keys itself.

6.1.2 About the .NET Smart Client

For the .NET environment, a .NET Smart Client component is available. This component offers a way to call the same interfaces provided by the Java Smart Client. However, it does not offer the added functionality of maintained session affinity or default values.

6.1.3 About the JSP Smart Client

JSP client integration tags are provided for developers to integrate Web applications with deployed Inline Services. These JSP tags deliver client interfaces that are equivalent to the APIs provided by the Java Smart Client.

6.1.4 About Web Services

Any client can access Real-Time Decision Server through Web services. The benefit to this means of integration is the lack of code needed on the client. Web service operations are defined in a WSDL file and definitions are contained in a schema file.

6.2 About the CrossSell Inline Service

Example Inline Services are included with Decision Studio. One of these is a cross selling example.

The CrossSell Inline Service simulates a simple implementation for a credit card contact center. As calls come into the center, information about the customer and the channel of the contact is captured.

Based on what we know of this customer, a cross selling offer is selected that is extended to the customer. The success or failure of that offer is tracked and sent back to the server so that the underlying decision model has the feedback that helps to refine its ability to make a better cross selling recommendation.

The CrossSell Inline Service is used to demonstrate the various means of integration in this guide.

Several Integration Points are included in the CrossSell example. Use the following instructions to familiarize yourself with these Integration Points.

Informants execute on the server when supplied with the proper parameters. Advisors execute and also return data. In order to supply the correct parameters for calls to Integration Points, we must first identify the Object IDs.

This section contains the following topics:

- Section 6.2.1, "Using Decision Studio to Identify Object IDs"
- Section 6.2.2, "Determining the Response of an Advisor"
- Section 6.2.3, "Knowing How to Respond to the Server"

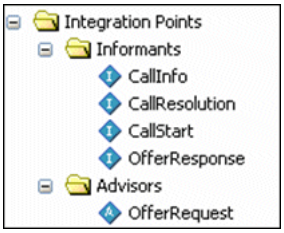
- Section 6.2.4, "Identifying Session Keys and Arguments"

6.2.1 Using Decision Studio to Identify Object IDs

To identify Object IDs:

1. Open Decision Studio by running *RTD_HOME*\eclipse\eclipse.exe.
2. Select **File > Import** to open the CrossSell Inline Service. Import appears.
3. Select **Existing Project into Workspace** and click **Next**. Browse for the CrossSell project at the location *RTD_HOME*\examples\CrossSell. Select **OK** and click **Finish**, opening the project.
4. Using the Inline Service Explorer, expand **Integration Points**. Then, expand the **Informants** and **Advisors** folders to view the Integration Points, as shown in Figure 6–1.

Figure 6–1 Informants and Advisors in the Inline Service Explorer



5. Use the Object ID **Toggle** icon to show the Object ID in the Inline Service Explorer:

When the Toggle icon is highlighted, the Object IDs show in the Inline Service Explorer; when the Toggle icon is not highlighted, the display label is shown.

Note: The Object ID of the Integration Point may or may not be the same as the label. Object IDs are used to reference Integration Points in method calls.

6.2.2 Determining the Response of an Advisor

Integration Points that deliver responses are called Advisors. An Advisor's **Response** tab in Decision Studio determines the response, by identifying a parameterized Decision object that gets implicitly invoked by the Advisor. The Decision object's responsibility is to select the best Choices from its assigned Choice Group. The choice attributes that are returned are determined by the configuration set on the definition of the Choice Group.

In our example, the OfferRequest integration point is an Advisor. It returns a single cross sell offer when it is invoked.

To determine the response of an advisor:

1. In Decision Studio, select the **OfferRequest** Integration Point to view the editor.
2. On the **Response** tab, under **Decision**, look up the Decision that OfferRequest uses to return a response. It should be **OfferDecision**.
3. Double-click **OfferDecision** under **Decisions** to view its detail pane.

4. On the **Selection Criteria** tab, under **Number of Choices to Select**, find the number of responses that OfferRequest provides.
5. On the **Selection Criteria** tab, under **Choice Group**, find the Choice Group that OfferRequest uses. It should be **Offers**.
6. Under **Choices**, double-click **Offers** to see the choice attributes associated with this Choice Group. These attributes will be returned when a call to the Advisor is made.

   Tip: In Decision Studio, use the Test view to call the Advisor and see what is returned. That way, you can see the offer returned and the attributes that come with it. To access the Test view, click the **Test** tab next to the **Problems** tab. Click the **Send** icon to send the request to the server:

6.2.3 Knowing How to Respond to the Server

Inline Services are most powerful when the success or failure of a Choice is tracked and the model is self learning based on that information. To know what feedback the server needs to be self learning, you must examine the Choice Event Model, as follows:

1. In Decision Studio, double-click the **Offer Acceptance** Choice Event Model. The editor appears on the right.
2. On the **Choice** tab, under **Positive Outcome Events**, you can see the Events that the server is interested in for learning. These are:
   - Interested
   - Purchased

   These outcomes are to be reported to the server from your Inline Service to give the proper feedback to the model.
3. The **OfferResponse** Integration Point is responsible for reporting this information.

6.2.4 Identifying Session Keys and Arguments

To invoke an Integration Point, we must supply values for the session keys and arguments expected by the Integration Point. In the request, we must use the Object IDs defined by Decision Studio for the Integration Point's session keys and arguments. The key name must match one of the session key names defined in Decision Studio for the Integration Point.

To identify session keys and arguments.

1. Select the **CallStart** Integration Point. On the **Request** tab of the editor of the Integration Point, under the **Session Keys** list, a path to the session key is shown starting with `session`; the last name in the path is the Object ID of the session key.

Note: If the session key is not displayed in object format, use the Object ID **Toggle** icon to change the display settings:

Only the final object ID is necessary for the session key. For example, in the case shown above, only the final string, `customerId`, is used.

2. To identify the arguments of the Integration Point, use the detail pane of to view the **Incoming Attribute** column of the **Request** tab. The **CallStart** incoming argument is **channel**.

7

Using the Java Smart Client

This chapter explains how to use the Java Smart Client for integration. An example is included with Oracle RTD installation.

For full information about the Java Smart Client API, see the Decision Studio online help.

This chapter contains the following topics:

7.1 Before you Begin

You must perform the following tasks first before you can work with the Java Smart Client example:

1. Install a Java Development Kit (JDK), with the `JAVA_HOME` environment variable set to its location. To obtain a JDK, go to the Sun Microsystems Web site at:

   `http://java.sun.com/products`

2. Install the Oracle RTD files and deploy Oracle RTD to an application server. See *Oracle Real-Time Decisions Installation and Administration Guide* for full information.

3. The Java Smart Client example works with the sample CrossSell Inline Service. Because of this, you must first populate the Oracle RTD Database with the CrossSell example data, then deploy the CrossSell Inline Service using Decision Studio.

   See *Oracle Real-Time Decisions Installation and Administration Guide* for information about populating the Oracle RTD Database with the CrossSell example data. See Part III, "Decision Studio Reference" for information about deploying Inline Services.

4. Start Real-Time Decision Server. For more information, see *Oracle Real-Time Decisions Installation and Administration Guide*.

7.2 Integrating with an Inline Service Using the Java Smart Client

In general, integration using the Java Smart Client includes the following steps:

1. Prepare a properties file.
2. Create a connection to the Inline Service.

3. Create a request that identifies the Integration Point to connect to and the parameters to identify the session and any other information the Integration Point needs to determine an outcome.
4. Invoke the request.
5. Gather and parse any response information from Advisors.
6. Close the connection.

This section contains the following topics:

7.2.1 Preparing the Java Smart Client Example

For this example, the CrossSell Inline Service has been integrated to a simple command-line application to demonstrate how to use the Java Smart Client for integration.

To prepare the Smart Client example:

1. Locate the file *RTD_HOME*`\client\Client Examples\Java Client Example\lib\sdbootstrap.properties` and open it for editing. Comment out all properties except for `client=true`, as follows:

   ```
   client=true
   #StudioStaticFilesLocation=shared_ui/studio
   #WebServerLocation=http://localhost:8080
   #WorkbenchServlet=/ui/workbench
   ```

   Then, save and close the file.
2. Open Decision Studio and choose **File > Import** , then select **Existing Projects into Workspace** and click **Next**.
3. For **Select root directory**, browse to *RTD_HOME*`\client\Client Examples\Java Client Example` and click **OK**. Then, click **Finish**.
4. From the menu bar, select **Window > Open Perspective > Java**. If the Console view is not visible, select **Window > Show View > Console**.
5. From the menu bar, select **Run > Run**.
6. In the Create, manage, and run configurations screen, select **Java Application** and click **New**.
7. Click **Browse** next to the **Project** field, then select **JavaSmartClientExample** and click **OK**.
8. Click **Search** next to the **Main class** field, then select **Example** and click **OK**.
9. Click **Apply**, then click **Run**. In the Console view, the following text appears:

   ```
   Ring! Ring! New telephone call!
   ```

```
Enter a customer ID between 1 and 1000:
```

10. Place the cursor after the colon, then enter a customer ID (such as 5) and press **Enter**. The response appears similar to the following:

```
Here are the deals we've got for you:
 1: ElectronicPayments
  Electronic payments eliminate the complications of handling checks.
Enter the line number of the offer that catches your interest, or zero if none
do:
```

11. Place the cursor after the final colon, then enter `1` to select the offer. The server responds with a final message.
12. The process repeats. Enter a customer ID greater than 1000 to stop the program.

You can find the source code for this example in the following file:

RTD_HOME\client\Client Examples\Java Client Example\src\com\sigmadynamics\client\example\Example.java

The example is explained in the following sections.

7.2.2 Creating the Java Smart Client Properties File

When a client application creates a Java Smart Client, it passes a set of properties to a Java Smart Client factory that represents the component's endpoint configuration. This file contains just enough information to allow the client to connect to a server endpoint. There are additional factory methods that use default configuration values; however it is best to explicitly specify the properties. The default properties file is shown in the following procedure.

The factory method uses the properties to connect to the server. When the factory connects to the server, it downloads the more complete configuration information to the client, such as the set of default responses that the client should use if it ever needs to run when the server is unavailable. The detailed client configuration is saved in a local file, the Java Smart Client configuration cache, and is updated automatically whenever the server's configuration changes.

To create the properties file:

1. Locate the file *RTD_HOME*\client\Client Examples\Java Client Example\lib\sdclient.properties and open it for editing. The file should appear as follows:

```
UseEndpointsInOrder = HTTP1
appsCacheDirectory = ${rootDir}/etc
timeout = 0
HTTP1.type = http
HTTP1.url = http://localhost:8080/
```

2. Modify the contents to match your server configuration. Explanations of the elements of this file are listed in Table 7–1. In particular, make sure that you have a valid cache directory and the endpoint URL is the URL and port of your local Real-Time Decision Server. By default, this is `http://localhost:8080`.

Table 7–1 Elements of sdclient.properties File

Element	Description
`UseEndpointsInOrder`	A comma-separated list of endpoint names, indicating the order in which the endpoints should be tried when establishing an initial connection to the server cluster during the Smart Client's initialization. After initialization, this list of endpoints is irrelevant because the server will supply an updated list of endpoints. The endpoint names in this list refer to definitions within this properties file; the names are not used elsewhere.
`appsCacheDirectory`	A file URL identifying a writable directory into which the client component may save the configuration information that it gets from the server. The cache provides insurance against the possibility that Real-Time Decision Server might be unavailable to the client application when the application initializes its client components. If `sdclient.properties` specifies a cache directory, it must already exist, otherwise, the client will use the Java virtual machine's temp directory
`timeout`	The timeout, in milliseconds, used by the original attempt to contact the server during the client component's initialization. After connecting to the server, the client uses the server's timeout, configured through the JMX MBean property `IntegrationPointGuaranteedRequestTimeout`.
`endpoint_name``.type`	The named endpoint type. Only HTTP is supported at this time.
`endpointName``.url`	A URL specifying the HTTP host and port of the server's HTTP endpoint. The default endpoint is `http://localhost:8080`.

7.2.3 Creating the Java Smart Client

To create the Java Smart Client, open the source file for the Example application at the following location:

```
RTD_HOME\client\Client Examples\Java Client Example\src\com\sigmadynamics\
client\example\Example.java
```

> **Tip:** This example source code can be used as a template for your Java Smart Client implementation.

The following imports are used to support Oracle RTD integration:

```
import com.sigmadynamics.client.IntegrationPointRequestInterface;
import com.sigmadynamics.client.IntegrationPointResponseInterface;
import com.sigmadynamics.client.SDClientException;
import com.sigmadynamics.client.SDClientFactory;
import com.sigmadynamics.client.SDClientInterface;
```

In the main method, the Example application demonstrates several techniques for using `SDClientFactory` to create an implementation of `SDClientInterface`, based on the arguments supplied to the Example application.

These arguments are passed to `getClient`, where the proper factory method is identified.

```
SDClientInterface client = getClient(args);
```

There are several factory methods used to create a Java Smart Client. By examining `getClient`, we see the various methods:

```
private static SDClientInterface getClient(String[] args ){
```

```
  try{
    if ( args.length == 0 )
      return getClientWithDefaultPropertiesFile();
```

This creates a Java Smart Client with the default properties file using `create(java.lang.String)`. The default properties file is referenced.

```
    if ( "-h".equals(args[0])){
      if ( args.length < 2 )
        return getClientWithDefaultHttpAddress();
```

This creates a Java Smart Client with the default HTTP address of `http://localhost:8080`. This is the default installation URL and port of Real-Time Decision Server. Uses `createHttp(java.lang.String, int, boolean)`.

```
      return getClientWithHttpAddress( args[1]);
    }
```

This creates a Java Smart Client with a supplied HTTP address. This is the address and port of your Real-Time Decision Server, if it is not at the default address. Uses `createHttp(String)`.

```
    if ( "-u".equals(args[0])){
      if ( args.length < 2 )
      {
        System.out.println("Missing properties file URL argument" );
        System.exit(-1);
      }
      return getClientWithPropertiesFileURL( args[1] );
    }
```

This creates a Java Smart Client with the information supplied in the properties file at the address specified. Uses `createFromProperties`.

```
    if ( "-f".equals(args[0])){
      if ( args.length < 2 )
      {
        System.out.println("Missing properties filename argument" );
        System.exit(-1);
      }
      return getClientWithPropertiesFileName( args[1] );
    }
```

This creates a Java Smart Client with the information supplied in the properties file. Uses `createFromPropertiesURL`.

```
    System.out.println("Unrecognized argument");
  }catch (SDClientException e ){
    e.printStackTrace();
  }
  System.exit(-1);
  return null;
}
```

These methods are summarized in the Java Smart Client API section of the Decision Studio online help.

7.2.4 Creating the Request

After populating the request, the client application calls the `invoke` method of `SDClientInterface` to send the request to the server and receives an `IntegrationPointResponseInterface` representing an array of choices calculated by the server.

```
IntegrationPointResponseInterface invoke(IntegrationPointRequestInterface
request);
```

In the example application, this call is made:

```
client.invoke(request);
```

> **Note:** If the client application wants to send a request for which it does not expect a response, and for which message delivery sequence is not critical, it can use the `invokeAsync` method instead of `invoke`.
>
> Requests sent through `invokeAsync` are not guaranteed to arrive at the server before requests sent through subsequent `invokeAsync` or `invoke` calls. When message delivery sequence is important, the `invoke` method should be used instead of `invokeAsync`, even when no response is expected.

After the request to the CallStart Integration Point is invoked, a new request is prepared and invoked for CallInfo.

```
// Supply some additional information about the telephone call.
// Apparently the CrossSell service expects very little here --
// just the channel again, which it already knows. Hence this message
// could be left out with no consequences.
request = client.createRequest(INLINE_SERVICE_NAME, "CallInfo");
request.setSessionKey( SESSION_KEY, sCustID );
request.setArg( "channel", "Call");
client.invoke(request);
```

7.2.5 Examining the Response

When an Advisor is invoked, a number response items, also known as Choices, will be returned. Your application must be prepared to handle this number of response items. See Section 6.2.2, "Determining the Response of an Advisor" for more information.

In the client application, the selected Choices are accessible through the `IntegrationPointResponseInterface` returned by the invoke method. The `IntegrationPointResponseInterface` provides access to an array of response item objects, `ResponseItemInterface`, where each response item corresponds to a Choice object selected by the Advisor's Decision.

The package `com.sigmadynamics.client` surfaces a Choice as a collection of value strings, keyed by name string.

In our example, when invoking a request on an Advisor Integration Point, be prepared to receive a response.

```
// Based on what the server knows about this customer, ask for some
// product recommendations.
request = client.createRequest(INLINE_SERVICE_NAME, "OfferRequest");
IntegrationPointResponseInterface response = client.invoke(request);
request.setSessionKey( SESSION_KEY, sCustID );
```

Knowing the number of responses expected allows you handle them accurately. The responses are read from the array and displayed to the customer.

```
if ( response.size() > 0 ){
// Since I know that CrossSell's OfferDecision returns only
// one Choice, I could get that choice from the response with
// response.get(0); Instead, I'll pretend that
// multiple offers could be returned instead of just one.
    System.out.println();
    System.out.println("Here are the deals we've got for you:");
    ResponseItemInterface[] items = response.getResponseItems();
    for ( int i = 0; i < items.length; i++ ){
      System.out.println(" " + (i+1) + ": " + items[i].getId());
      String message = items[i].getValue("message");
      if ( message != null )
        System.out.println("   " + message );
    }
    System.out.println();
    System.out.println("Enter the line number of the offer that catches your
    interest, or zero if none do: " );
```

7.2.6 Closing the Loop

Many Inline Services are designed to be self learning. In the CrossSell Inline Service, the OfferResponse Informant reports interest in a cross sell offer back to a Choice Event model.

```
// Tell the server the good news.
request = client.createRequest(INLINE_SERVICE_NAME, "OfferResponse");
request.setSessionKey( SESSION_KEY, sCustID );
request.setArg( "choiceName", prodName );

// "Interested" is one of the Choice Events defined for the choice group, Offers.
```

To identify the Choice Event model and Choices, see Section 6.2.3, "Knowing How to Respond to the Server."

```
request.setArg( "choiceOutcome", "Interested" );
client.invoke(request);
```

Finally, the session is closed by invoking the CallResolution Informant in the server, which in the CrossSell example has been designed to terminate the session.

```
// Close the server's session.
request = client.createRequest(INLINE_SERVICE_NAME, "CallResolution");
request.setSessionKey( SESSION_KEY, sCustID );
client.invoke(request);
```

7.2.7 Closing the Client

When the client application is finished using its `SDClientInterface`, and doesn't intend to use it again, it calls the component's close method, to release any instance-specific information.

```
client.close();
```

8

Using Java Smart Client JSP Tags

A convenient way to integrate a Web application with a deployed Inline Service is to use the JSP client integration tags. JSP allows you to generate interactive Web pages that use embedded Java. The JSP tags provided are based on the Java Smart Client discussed in the previous chapter.

There is negligible overhead when using the JSP tags. In addition, the tags incorporate automatic reuse of Smart Clients for same session to enhance performance. When a Java Smart Client is created using the JSP tag, a check is performed to see if a client already exists with the same name and properties and has not been closed. If it does, it automatically reuses that client; if not it will create a new one.

For full information about the JSP Smart Client tags, see the Decision Studio online help.

This chapter contains the following topics:

8.1 Before You Begin

You must perform the following tasks first before you can work with the JSP client integration tags:

1. Install a Java Development Kit (JDK), with the `JAVA_HOME` environment variable set to its location. To obtain a JDK, go to the Sun Microsystems Web site at `http://java.sun.com/products`.
2. Install the Oracle RTD files and deploy Oracle RTD to an application server. See *Oracle Real-Time Decisions Installation and Administration Guide* for full information.
3. The Java Smart Client example works with the sample CrossSell Inline Service. Because of this, you must first populate the Oracle RTD Database with the CrossSell example data, then deploy the CrossSell Inline Service using Decision Studio.

   See *Oracle Real-Time Decisions Installation and Administration Guide* for information about populating the Oracle RTD Database with the CrossSell example data. See Part III, "Decision Studio Reference" for information about deploying Inline Services.
4. Start Real-Time Decision Server. For more information, see *Oracle Real-Time Decisions Installation and Administration Guide*.

8.2 Integrating with an Inline Service Using Java Smart Client JSP Tags

In general, integration using the Java Smart Client includes the following steps:

1. Prepare a properties file.
2. Use an Invoke or AsyncInvoke tag to create a request to the server.
3. Gather and parse any response information from Advisors.
4. Close the connection.

A working example of using the Smart Client JSP tags for integration can be found at *RTD_HOME*\client\Client Examples\JSP Client Example\example.jsp.

8.3 Deploying the JSP Smart Client Example

Note: If your Real-Time Decision Server port is not 8080, you must edit the client properties information before deploying the JSP Smart Client example, as follows:

1. Open *RTD_HOME*\client\Client Examples\JSP Client Example\sdclient-test.war with WinZip or WinRAR.
2. In sdclient-test.war, open client\sdclient.properties.
3. Search for the entry:

   ```
   HTTP1.url = http://localhost:8080
   ```

4. Change the URL localhost:8080 to match the host and port of the Real-Time Decision Server that you are using.
5. Save the file back into sdclient-test.war.

For this example, the CrossSell Inline Service has been integrated to a simple command-line application to demonstrate how to use the Smart Client for integration. You need to deploy the JSP Smart Client example to your application server, as described in the following sections.

This section contains the following topics:

- Section 8.3.1, "Deploying the JSP Smart Client Example to OC4J"
- Section 8.3.2, "Deploying the JSP Smart Client Example to WebSphere"
- Section 8.3.3, "Deploying the JSP Smart Client Example to WebLogic"

8.3.1 Deploying the JSP Smart Client Example to OC4J

To deploy the JSP Smart Client example to OC4J:

1. Log in to the Application Server Control as the oc4jadmin user. You can access the Application Server Control at http://*oc4j_host*:*port*/em. For the standalone version of OC4J, the port number is typically 8888.
2. On the OC4J home page, click the **Applications** tab.

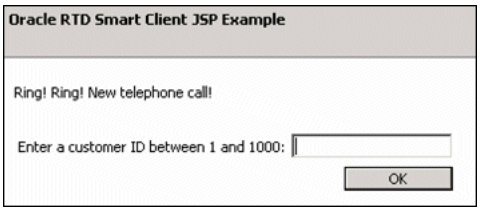
   If you are using OC4J as part of Oracle Application Server, first click **home** under the **Groups** heading, then proceed to the **Applications** tab.
3. Click **Deploy**. On the Deploy: Select Archive page, under the Archive heading, browse to specify the archive location *RTD_HOME*/client/Client Examples/JSP Client Example/sdclient-test.war. Then, click **Next**.

4. On the Deploy: Application Attributes page, enter `JSPClientExample` for **Application Name**, then choose **rtd-web-site** for **Bind Web Module to Site**. Then, click **Next**.
5. On the Deploy: Deployment Settings page, click Deploy.
6. To access the application, open a Web browser and go to:

   `http://ocj4_host:port/sdclient-test/example.jsp`

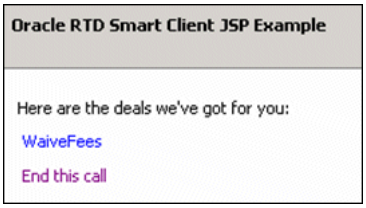
   A Web page appears that simulates a service call, as shown in Figure 8–1.

Figure 8–1 JSP Smart Client Example: Enter Customer ID



7. Enter a customer ID (such as 5) and click **OK**. A response page appears, displaying an offer and an option to end the call, as shown in Figure 8–2.

Figure 8–2 JSP Smart Client Example: Displayed Offer



8. Click the offer link, or click **End this call**.

8.3.2 Deploying the JSP Smart Client Example to WebSphere

To deploy the JSP Smart Client example to WebSphere:

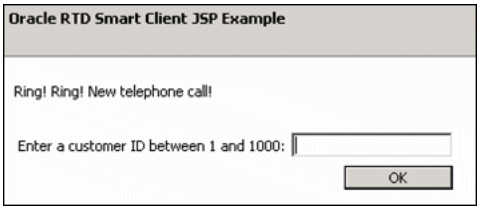
1. Access the Integrated Solutions Console at the URL `http://websphere_host:port/ibm/console`. At the login prompt, enter the administrator user name and password. On Windows, you can also access the Integrated Solutions Console through **Start > Programs**.
2. In the tree on the left, expand **Applications**, then choose **Enterprise Applications**.
3. Click **Install**.
4. In the Path to the new application section, enter or browse to the path `RTD_HOME/client/Client Examples/JSP Client Example/sdclient-test.war`.
5. For **Context root**, enter `sdclient-test`.
6. Click **Next**, then click **Next** again, then click **Next** again.
7. Click **Finish**, then click **Save**.
8. On the Enterprise Applications page, select the **sdclient-test** application and click **Start**.

9. To access the application, open a Web browser and go to:

   `http://websphere_host:port/sdclient-test/example.jsp`

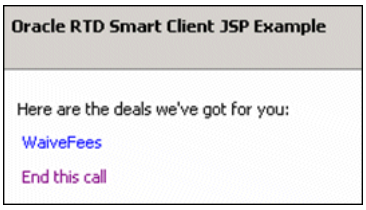
   A Web page appears that simulates a service call, as shown in Figure 8–3.

Figure 8–3 JSP Smart Client Example: Enter Customer ID



10. Enter a customer ID (such as `5`) and click **OK**. A response page appears, displaying an offer and an option to end the call, as shown in Figure 8–4

Figure 8–4 JSP Smart Client Example: Displayed Offer



11. Click the offer link, or click **End this call**.

8.3.3 Deploying the JSP Smart Client Example to WebLogic

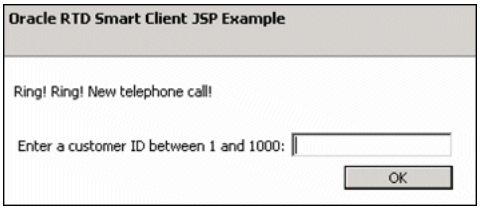
To deploy the JSP Smart Client example to WebLogic:

1. Access the WebLogic Server Administration Console for your Oracle RTD domain at the URL `http://weblogic_host:port/console`. At the login prompt, enter the administrator user name and password. On Windows, you can also access the WebLogic Server Administration Console through **Start > Programs > BEA Products > User Projects > domain_name > Admin Server Console**.
2. In the tree on the left, click **Deployments**.
3. Click **Install**. You may need to click **Lock & Edit** first to enable the **Install** button.
4. Go to *RTD_HOME*`/client/Client Examples` and select **JSP Client Example**, then click **Next**.
5. Select **Install this deployment as an application**, then click **Next**.
6. On the Optional Settings page, enter `JSPClientExample` for **Name**. Then, click **Next**.
7. Review your settings and click **Finish**.
8. Click **Save**, then click **Activate Changes**.
9. Start the application by selecting **JSPClientExample** application in the **Deployments** table, then clicking **Start > Servicing all Requests**. When prompted, click **Yes**. The application is now running.
10. To access the application, open a Web browser and go to:

`http://weblogic_host:port/sdclient-test/example.jsp`

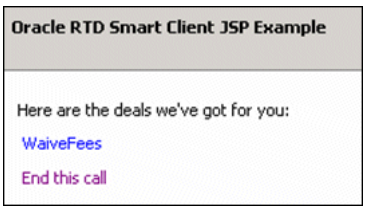
A Web page appears that simulates a service call, as shown in Figure 8–5.

Figure 8–5 JSP Smart Client Example: Enter Customer ID



11. Enter a customer ID (such as `5`) and click **OK**. A response page appears, displaying an offer and an option to end the call, as shown in Figure 8–6.

Figure 8–6 JSP Smart Client Example: Displayed Offer



12. Click the offer link, or click **End this call**.

9

Using the .NET Smart Client

The .NET Smart Client provides a very similar client to the Java API to make calls from your application. With the current implementation, the .NET Smart Client does not have some of the advanced features of the Java Smart Client, including session affinity management and default response handling.

For full information about the .NET Smart Client API, see the Decision Studio online help.

This chapter contains the following topics:

9.1 Before You Begin

You must perform the following tasks first before you can work with the .Net Smart Client:

1. Install a Java Development Kit (JDK), with the `JAVA_HOME` environment variable set to its location. To obtain a JDK, go to the Sun Microsystems Web site at `http://java.sun.com/products`.
2. Install the Oracle RTD files and deploy Oracle RTD to an application server. See *Oracle Real-Time Decisions Installation and Administration Guide* for full information.
3. The .NET Smart Client example works with the sample CrossSell Inline Service. Because of this, you must first populate the Oracle RTD Database with the CrossSell example data, then deploy the CrossSell Inline Service using Decision Studio.

   See *Oracle Real-Time Decisions Installation and Administration Guide* for information about populating the Oracle RTD Database with the CrossSell example data. See Part III, "Decision Studio Reference" for information about deploying Inline Services.
4. Start Real-Time Decision Server. For more information, see *Oracle Real-Time Decisions Installation and Administration Guide*.

9.2 Integrating with an Inline Service Using the .NET Smart Client

In general, the following are the steps for integration:

1. Create the Oracle RTD Smart Client within your application code.

2. Create a request directed at an Inline Service and an Integration Point.
3. Populate the request with arguments and session keys.
4. Invoke the request using the Smart Client.
5. If the request is invoked on an Advisor, examine the response.
6. Close the Smart Client when finished.

The .NET Smart Client is located at *RTD_HOME*\client\Client Examples\Dot Net Client Example\sdclient.dll. This file should be co-located with your application in order to be accessible.

9.3 .NET Integration Example

You can find an example of a .NET integration client in *RTD_HOME*\client\Client Examples\Dot Net Client Example\DotNetSmartClientExample.sln. You can open this example in Microsoft Visual C# 2008 Express Edition, then run or debug the example.

In this example, Informant and Advisor Integration Points are invoked on the CrossSell Inline Service. See Section 6.2, "About the CrossSell Inline Service" to familiarize yourself with this Inline Service. In the example, the Integration Points are invoked and the return values from the Advisor are written to the console.

Follow these steps to run the example in Microsoft Visual Studio:

1. Start **Microsoft Visual C# 2008 Express Edition**.
2. From the menu bar, select **File > Open > Project**.
3. For **File Name**, select *RTD_HOME*\client\Client Examples\Dot NET Client Example\DotNetSmartClientExample.sln and click **Open**.
4. If either the host or the port that Real-Time Decision Server is running on is different from the default localhost:8080, perform the following:
   - In the right-hand Solution Explorer window, double-click **DotNetSmartClientExample.cs**.
   - Locate the following line:

     ```
     SDClient client = new SDClient("http://localhost:8080");
     ```

   - Change localhost:8080 to match the host and port where Real-Time Decision Server is running.
   - Save and close the file.
5. From the menu bar, select **Debug > Start**. In the Console window, the following text appears:

   ```
   Ring! Ring! New telephone call!
   Enter a customer ID between 1 and 1000:
   ```

6. Place the cursor after the colon, then enter a customer ID (such as 5) and press **Enter**. The response appears similar to the following:

   ```
   Here are the deals we've got for you:
    1: ElectronicPayments
     Electronic payments eliminate the complications of handling checks.
   Enter the line number of the offer that catches your interest, or zero if none
   do:
   ```

7. Place the cursor after the final colon, then enter `1` to select the offer. The server responds with a final message.
8. The process repeats. Press **Enter** at the Customer ID prompt, without entering a number, to stop the program.

10

Web Service Client Example

Real-Time Decision Server Integration Points are available through a Zero Client approach. Integration Points on a deployed Inline Service are exposed through a Web services definition.

The ability to invoke and asynchronously invoke a deployed Integration Point is exposed as a Web service by Real-Time Decision Server. The definition of these operations are available in a WSDL file, located at:

RTD_HOME\deploy\DecisionService\DecisionService.wsdl

The WSDL file defines all complex types and operations available.

Some slight structural changes were introduced in Oracle RTD Version 2.2 to bring the Decision Service up to the WS-I Basic level of compliance. The previous version of the WSDL file is named:

RTD_HOME\deploy\DecisionService\DecisionServiceLegacy.wsdl

Although implementors should develop new clients using the new WSDL, the server still understands the protocol defined by `DecisionServiceLegacy.wsdl`, and existing clients should experience no loss of functionality.

The instructions in this chapter detail how to use Sun's NetBeans IDE to create a Java WSDL Web service client using the RTD DecisionService WSDL file. The code content for the Web service project main class is also provided.

This chapter contains the following topics:

10.1 Before You Begin

You must perform the following tasks first before you can work with the Web Service Client:

1. Download the NetBeans IDE (Java Bundle) from `http://www.netbeans.org/downloads/index.html`.
2. Install and start the NetBeans IDE.

3. Locate the Oracle RTD `DecisionService.wsdl` file and Java project main content file. These files can be located in an Oracle RTD installation at the following location:
   - *RTD_HOME*`\OracleBI\RTD\client\Client Examples\Web Service Client Example\DecisionService.wsdl`
   - *RTD_HOME*`\OracleBI\RTD\client\Client Examples\Web Service Client Example\main-content.txt`

RTD_HOME`\deploy\DecisionService\DecisionService.wsdl`

The WSDL file defines all complex types and operations available.

Some slight structural changes were introduced in Oracle RTD Version 2.2 to bring the Decision Service up to the WS-I Basic level of compliance. The previous version of the WSDL file is named:

RTD_HOME`\deploy\DecisionService\DecisionServiceLegacy.wsdl`

Although implementors should develop new clients using the new WSDL, the server still understands the protocol defined by `DecisionServiceLegacy.wsdl`, and existing clients should experience no loss of functionality.

10.2 Creating a New NetBeans Java Application Project

To create a new NetBeans Java application project, perform the following steps:

1. In the top menu, select **File > New Project**.
2. In the **New Project** dialog under Categories, select **Java**.
3. Under the **Projects** section of the dialog, select **Java Application**.
4. Click **Next**.
5. Name the **Project**, and click **Finish**.

10.3 Installing the JAX-RPC Web Services Plug-in

To install the JAX-RPC Web services plug-in, perform the following steps:

1. In the top menu, select **Tools > Plugins > Available Plugins**.
2. In the **Plugins** dialog, click the **Available Plugins** tab.
3. Check the **JAX-RPC Web Services plug-in** check-box and click **Install**. Install the plug-in.

10.4 Creating an Oracle RTD Web Service Client

To create an Oracle RTD Web Service Client, perform the following steps:

1. In the **Projects** explorer, right-click your project and select **New > Web Services Client...**
2. In the **New Web Service Client** dialog, select the **Local File** radio button.
3. Click the **Browse...** button next to **Local File:**
4. Locate the file `DecisionService.wsdl` in the RTD installation.

   Example: `C:\OracleBI\RTD\client\Client Examples\Web Service Client Example\DecisionService.wsdl`

5. For **Client Style**, select **JAX-RPC Style**.
6. For **Package**, select your project package.
7. Click **Finish**.

10.5 Adding the Provided Java Code and Testing the Client

To add the provided Java code and to test the client, perform the following steps:

1. In the **Projects** explorer, locate the generated `Main.java` file and open it.

   It should be under *`PROJECT_NAME`* `> Source Packages >` *`PROJECT_NAME`* `> Main.java`.

2. In the **main()** method, add the code content in the file `main-content.txt` and save `Main.java`.

   Note: Additional exceptions handling code may be required for the project to compile and run properly.

3. Select **Run > Run Main Project** from the top menu.
4. View the output in the **Output** tab at the bottom of the IDE.

11

Using the Oracle RTD PHP Client

The Oracle RTD PHP Client is a new feature introduced in Oracle RTD Version 3.0.0.1.

A convenient way to integrate a Web application with a deployed Inline Service is to use the PHP client integration classes. PHP allows you to generate interactive Web pages. The PHP classes provided offer functionality similar to the Java Smart Client discussed in *Oracle Real-Time Decisions Platform Developer's Guide*.

This chapter contains the following topics:

11.1 Before You Begin

You must perform the following tasks first before you can work with the PHP Client example:

1. Install a Java Development Kit (JDK), with the `JAVA_HOME` environment variable set to its location. To obtain a JDK, go to the site:

   `http://developers.sun.com/downloads`

2. Install the Oracle RTD files and deploy Oracle RTD to an application server. See *Oracle Real-Time Decisions Installation and Administration Guide* for full information.

3. The PHP Client example works with the sample CrossSell Inline Service. Because of this, you must first populate the Oracle RTD Database with the CrossSell example data, then deploy the CrossSell Inline Service using Decision Studio.

   See *Oracle Real-Time Decisions Installation and Administration Guide* for information about populating the Oracle RTD Database with the CrossSell example data. See Part III, "Decision Studio Reference" in *Oracle Real-Time Decisions Platform Developer's Guide* for information about deploying Inline Services.

4. Start Real-Time Decision Server. For more information, see *Oracle Real-Time Decisions Installation and Administration Guide*.

5. Install and configure an environment suitable for evaluating PHP 5.2 scripts, such as Apache with mod_php.

6. Install either the PHP Soap library from the PHP Installer (version 5.2 or later), or the NuSoap library (release 0.7.3 or later) from SourceForge.

11.2 Integrating with an Inline Service Using the Oracle RTD PHP Client

In general, integration using the Oracle RTD PHP Client includes the following steps:

1. Prepare an Oracle RTD PHP Client .ini file.
2. Prepare Client objects.
3. Create a request that identifies the following:
   - The Integration Point to connect to
   - The parameters to identify the session
   - Any other information the Integration Point needs to determine an outcome
4. Use Oracle RTD PHP method syntax to create requests for Informants.
5. Use Oracle RTD PHP method syntax to parse responses from Advisors.
6. Close the connection.

> **Note:** The Oracle RTD PHP Client API reference may be found in *RTD_HOME*\client\Client Examples\PHP Client Example\docs.

Two working examples of using the PHP Client may be found at *RTD_HOME*\client\Client Examples\PHP Client Example:

- example_nusoap.php demonstrates the use of the Oracle RTD PHP Client with NuSoap
- example.php implements the same functionality but uses PHP Soap

11.3 Deploying the PHP Client Examples

This section consists of the following topics:

11.3.1 Installing PHP Client Library and Example Files

For this example, the CrossSell Inline Service is exercised by a simple PHP page to demonstrate how to use the PHP Client.

You need to deploy the PHP Client example to your Web server, as described in the following section.

To deploy the PHP Client example to Apache:

1. Place the Oracle RTD PHP Client library:

- *RTD_HOME*\client\Client Examples\PHP Client Example\rtd

into a location on your PHP include path.

This rtd folder should contain the following files:

- DecisionService.wsdl
- rtd.client.base.php
- rtd.client.nusoap.php
- rtd.client.phpsoap.php
- rtd_client_conf.ini
- rtd_client_nusoap_conf.ini

2. Place the appropriate Oracle RTD PHP Client example - **example.php** for PHP Soap or **example_nusoap.php** for NuSoap - into a path from which your Apache server is configured to serve PHP scripts, for example, /home/www/example.php.

11.3.2 Editing the NuSoap Path Library Location

The Oracle RTD client library assumes that nusoap.php is located in a directory structure of the form <*php_includes*>/nusoap/nusoap.php, where <*php_includes*> is among the PHP installation's include directories.

However, NuSoap files download into the directories **lib** and **sample**.

Users must do one of the following:

- Copy the contents of the **lib** directory to <*php_includes*>/nusoap.
- In rtd.client.nusoap.php, edit the following entry:

  include_once "nusoap/nusoap.php"

  to reflect the actual NuSoap library location.

11.3.3 Preparing the Oracle RTD PHP Client .ini File

The .ini files provided by Oracle RTD are the following:

- rtd_client_conf.ini (for PHP Soap)
- rtd_client_nusoap_conf.ini (for NuSoap)

The PHP client properties in these files are as follows:

- **wsdl** - wsdl file location. Use with PHP Soap.
- **clientClass** - Client class name
- **appsCacheClass** - Cache class name
- **appsCacheFile** - temp file for default response
- **clientTimeout** - Oracle RTD integration point invoke timeout in seconds. This is optional.
- **endpointUrl** - url for the decision service

Example of rtd_client_conf.ini

The following is an example of rtd_client_conf.ini:

```
# RTD integration service wsdl file path
```

```
wsdl=rtd/DecisionService.wsdl

clientClass=Oracle_Rtd_Client_Impl
appsCacheClass=Oracle_Rtd_Client_File_Cache

# temp file for default response
appsCacheFile=c:/temp/rtd_default.dat

# RTD integration point invoke timeout in seconds
clientTimeout=2

# RTD service url
endpointUrl=http://localhost:8080/
```

Example of rtd_client_nusoap_conf.ini

The following is an example of `rtd_client_nusoap_conf.ini`:

```
# client class name
clientClass=Oracle_Rtd_Client_Nu

# cache class name
appsCacheClass=Oracle_Rtd_Client_File_Cache

# temp file for default response
appsCacheFile=c:/temp/rtd_default_nusoap.dat

# RTD integration point invoke timeout in seconds
clientTimeout=2

# RTD service url
endpointUrl=http://localhost:8080/
```

Editing the .ini files

Oracle RTD provides Client example initialization files as a basis for your configuration. You must ensure that the settings in the files match your system. If necessary, edit the files so that they are correct for your configuration setup.

For example, if your Real-Time Decision Server server is not running on localhost or its listening port is not 8080, you must edit the appropriate .ini file before deploying the Oracle PHP Client example, as follows:

1. In *RTD_HOME*`\client\Client Examples\PHP Client Example\rtd`, open the appropriate file, `rtd_client_conf.ini` or `rtd_client_nusoap_conf.ini`, with a text editor.
2. Search for the entry:

   ```
   endPointUrl = http://localhost:8080
   ```

3. Change the URL `localhost:8080` to match the host and port of the Real-Time Decision Server that you are using.
4. Save the file.

How the Client Properties are Used

When a client application creates an Oracle RTD PHP Client, it passes a set of properties to an Oracle RTD PHP Client factory that represents the component's endpoint configuration.

If no argument is given, the Client factory derives its settings by applying `parse_ini_file` to `rtd_client_conf.ini`.

The factory method uses the settings to connect to the server. When the Oracle RTD PHP Client has connected to the server, it downloads a more complete set of configuration information, such as the request timeout duration. It will also maintain locally the set of default responses that the client should use if it ever needs to run when the server is unavailable.

The detailed client configuration is saved in a local file, the Oracle RTD PHP Client configuration cache, and is updated automatically whenever the server's configuration changes.

11.3.4 Creating the Oracle RTD PHP Client

To create the Oracle RTD PHP Client, open the source file for the Example PHP script appropriate for your environment (**`example.php`** for users of PHP Soap and **`example_nusoap.php`** for NuSoap users).

The following `include` is used to support Oracle RTD integration with NuSoap:

```
include_once "rtd/rtd.client.nusoap.php";
```

The following `include` is used to support Oracle RTD integration with PHP Soap:

```
include_once "rtd/rtd.client.phpsoap.php";
```

`example.php` and **`example_nusoap.php`** demonstrate different ways of obtaining an instance of the Oracle RTD PHP Client.

A client may be obtained with settings given explicitly inline, as is shown by **`example.php`**:

```
$client = Oracle_Rtd_Client_Factory::createClient(array(
      "wsdl"=>"g:/php/includes/rtd/DecisionService.wsdl",
      "clientClass"=>"Oracle_Rtd_Client_Impl",
      "appsCacheClass"=>"Oracle_Rtd_Client_File_Cache",
      "appsCacheFile"=>"c:/temp/rtd_default.dat",
      "clientTimeout"=>2,
      "endpointUrl"=>"http://192.168.0.196:8081/"
   ));
```

In **`example_nusoap.php`**, the settings are parsed from a `.ini` file that has been placed in the same directory as the example PHP script, and may be found in *`RTD_HOME`*`\client\Client Examples\PHP Client Example\rtd\rtd_client_nusoap_conf.ini`:

```
$config = parse_ini_file("rtd_client_nusoap_conf.ini");
$client = Oracle_Rtd_Client_Factory::createClient($config);
```

11.3.5 Creating the Request

This line of code creates a Request object:

```
$request = $client->createRequest();
```

A request must be configured with the destination Inline Service:

```
$request->setServiceName("CrossSell");
```

The selection of an Inline Service may be further specialized with a Deployment State. If omitted then the Inline Service deployed in the highest State receives the Request (Production is higher than QA, which is higher than Development):

```
$request->setDeploymentState("Development");
```

The details of a Request are specific to each Inline Service. In this example, the CallStart Informant requires a Session Key named `customerId` and an additional parameter named `channel`:

```
$request->setIntegrationPointName("CallStart");
$request->setSessionKey("customerId", 3);
$request->setArg("channel", "Call");
```

After populating the request, the client application calls the invoke method of the Client, sending the Request to the RTD Server:

```
$client->invoke($request);
```

11.3.6 Examining the Response

When an Advisor is invoked, a number of response items, also known as Choices, will be returned. Your application must be prepared to handle these items. See Section 6.2.2, "Determining the Response of an Advisor" in *Oracle Real-Time Decisions Platform Developer's Guide* for more information.

In the client PHP script, the selected Choices are accessible through the Response interface returned by the Client's invoke method. This object provides access to an array of ResponseItem objects, each one corresponding to a Choice object selected by the Advisor's Decision.

The Choice's name may be accessed with the `getId()` method, and the Choice's attributes are available through `getAttributes()`:

```
$request->setIntegrationPointName("OfferRequest");
$response = $client->invoke($request);
```

In the PHP example scripts, the response items are each printed to the Web page in the order given by the RTD server:

```
$items = $response->getResponseItems();
foreach ($items as $item) {
   echo "<h1>" . $item->getId() . "</h1>";
   foreach ($item->getAttributes() as $key => $value) {
      echo $key . ': '.$value."<br>";
   }
}
```

For an example of the Web page output, see Section 11.3.8, "Testing the PHP Client Example."

11.3.7 Closing the Loop

Many Inline Services are designed to be self learning. In the CrossSell Inline Service, the OfferResponse Informant reports interest in a cross sell offer back to a Choice Event Model. For simplicity, this example registers an "Interested" Choice Event for the highest ranked Choice among the response items.

```
if ($response->size() > 0) {
   $request->setIntegrationPointName("OfferResponse");
   $request->setArg("choiceName", $response->get(0)->getId());
```

```
    $request->setArg("choiceOutcome", "Interested");
    $client->invoke($request);
}
```

For more information about Choice Event models and Choices, see Section 6.2.3, "Knowing How to Respond to the Server" in *Oracle Real-Time Decisions Platform Developer's Guide*.

Finally, the session is closed by invoking the CallResolution Informant, which in the CrossSell example has been designed to terminate the session, freeing resources and triggering tasks to run that wait for the end of a session.

```
$request->setIntegrationPointName("CallResolution");
$client->invoke($request);
```

11.3.8 Testing the PHP Client Example

To access the application, open a web browser and enter one of the following URLs (or as specified in Section 11.3.1, "Installing PHP Client Library and Example Files," step 2):

`http://`*`apache_host:port`*`/example.php` (for PHP)

or

`http://`*`apache_host:port`*`/example_nusoap.php` (for NuSoap)

This displays the choice response from the Advisor call in the Inline Service, together with the choice attributes, such as shown in the following sample output:

Part III

Decision Studio Reference

The chapters in Part III provide an in-depth look at the concepts, components, and APIs needed to use Decision Studio to develop Inline Services.

Part III contains the following chapters:

12

About Decision Studio

Decision Studio is a tool used to define and manage Inline Services. All aspects of Inline Services are exposed in Decision Studio. The target user of Decision Studio is an IT professional with a basic knowledge of Java and a general understanding of application development and lifecycle issues.

Decision Studio is a rich-client application that follows an integrated development environment (IDE) paradigm. Decision Studio makes use of an Inline Service Explorer view on the left, and an editor view on the right. The navigator view displays a predefined Inline Service folder structure. Items within each folder are Inline Service metadata elements. Using Decision Studio, metadata elements may be added, edited, and deleted. When a metadata element is double-clicked, the details of the element are shown in the object editor. Each metadata element type has its own editor. The elements are first represented as XML metadata, and then later, Java classes are generated from which the running Inline Service is compiled.

Decision Studio is based on the Eclipse IDE. It combines features that are specific to managing Inline Services with the features of the Eclipse IDE, which include general purpose Java development tools, integration with Software Configuration Management (SCM) systems, and so on.

Note: The following terms are referenced throughout the Oracle RTD documentation:

- *RTD_HOME*: This is the directory into which Oracle RTD is installed. For example, `C:\OracleBI\RTD`.
- *RTD_RUNTIME_HOME*: This is the application server specific directory in which the application server runs Oracle RTD.

For more information, see the section "About the Oracle RTD Run-Time Environment" in *Oracle Real-Time Decisions Installation and Administration Guide*.

This chapter contains the following topics:

12.1 About Inline Services

An Inline Service is a deployable application that monitors and advises business processes at key points across the enterprise on a real-time and continuous basis. Inline Services do not follow business processes from end-to-end, but rather focus on specific and identified points within the process. Inline Services are configured and deployed using Decision Studio and analyzed and tuned using Decision Center. Inline Services run on Real-Time Decision Server. Together these components comprise Oracle RTD.

12.2 Decision Studio and Eclipse

Decision Studio is based on Eclipse, an open source Java IDE produced by the Eclipse Foundation. Decision Studio exists as a standard plug-in to the Eclipse environment. If you are using Eclipse, you have the advantage of using the environment for additional development and advanced features. If you are not familiar with Eclipse, it is completely transparent to using Decision Studio.

This section contains the following topics:

12.2.1 Selecting the Decision Studio Workspace

When you log into Decision Studio, you can select the workspace for the session. The default workspace is `C:\Documents and Settings\`*`username`*`\Oracle RTD Studio`. You can optionally set the workspace location to be the default from then on

You can also change the workspace during any session through the menu path File > Switch Workspace.

12.2.2 Setting the Java Compiler Compliance Level

Inline Services must be compiled with Java compiler compliance level 5.

To set this as a preference for all Inline Services, perform the following steps:

1. Log in to Decision Studio.
2. Navigate the menu path Windows > Preferences.
3. In the Preferences window, select **Java**, then **Compiler**, in the left panel.
4. In the JDK Compliance area, select 5.0 for **Compiler compliance level**.
5. Click OK, and confirm that you want the full rebuild to proceed.

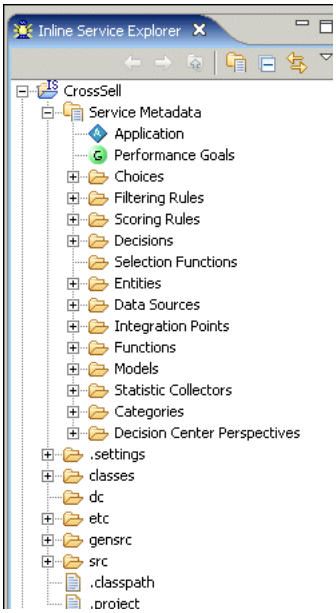
Inline Services created in previous versions of Oracle RTD may need to have their Java compiler compliance level manually changed to 5.0.

To change the Java compiler compliance level at the individual Inline Service level, right-click the Inline Service name, select Properties > Java Compiler, then select 5.0 for the Compiler compliance level.

12.2.3 About the Inline Service Explorer

The Inline Service Explorer gives you access to all aspects of your Inline Service projects. A typical Inline Service project is shown in Figure 12–1.

Figure 12–1 Inline Service Explorer



The contents of the Inline Service folder are described in Table 12–1.

Table 12–1 Contents of Inline Service Folder

Folder Name	Description
Service Metadata	The metadata that forms the Inline Service. The default editor for this type of file is the editor specific to each element. Although you can also edit metadata in a text editor, this is not recommended.
.settings	Contains settings specific to the Eclipse editor.
classes	The classes generated by the compile process.
etc	This directory contains various scripts and files which are used for system administration. If a Load Generator script is built, it is kept in this folder by convention.
gensrc	The generated source code files for the Inline Service.
src	Custom Java code, which may include arbitrary user-provided Java classes. Some of these classes can be used to override the default behavior of the generated Inline Service Java classes.

Table 12–1 (Cont.) Contents of Inline Service Folder

Folder Name	Description
lib	Optionally, a `lib` folder is created by the user when using outside classes. For instance, assume you want to access a class called `THashMap` in some function, logic, or initialization block. This class exists in the `tcollections.jar` file. To use the class, create the `lib` folder under the project directory in the project workspace, and then put the `tcollections.jar` file in the folder. To use a class from this jar, import using the **Advanced** button next to description and then use the class in your code.
.classpath	The file containing the Java classpath for the project. There is no need to edit this file.
.project	The Eclipse project file.

12.2.4 Code Generation

In general, as elements are configured for an Inline Service, four files are produced:

- An `.sda` file that stores the configuration as metadata.
- A `.java` file that is generated from the metadata and is compiled into a class file.
- A `.java` file that extends the original generated file and can be used in unusual circumstances to override the actions of the generated file.
- The class file that is first compiled from the generated file and subsequently compiled from any overrides.

The files are named in the following manner:

- **Metadata:** `Object_ID.sda`
- **Generated:** `GENObject_ID.java`
- **Override:** `Object_ID.java`
- **Class:** `Object_ID.class`
- **Generated Class:** `GENObject_ID.class`

> **Tip:** The Object ID is created as you name the object. Object IDs are shown in the text box below the name of the object for which they are created. The Object ID may be different from the label to conform to Java standards. To look up an Object ID, toggle between showing labels and showing Object IDs using the Toggle icon:
>
> 

For instance, consider an element named **Customer account**. An Object ID is formed, **CustomerAccount**, that conforms to Java naming standards.

The files created are:

- **Metadata:** `CustomerAccount.sda`
- **Generated:** `GENCustomerAccount.java`
- **Override:** `CustomerAccount.java`
- **Class:** `CustomerAccount.class`
- **Generated Class:** `GENCustomerAccount.class`

12.2.5 About Decision Studio Perspectives and Views

Decision Studio lets you work with an Inline Service from several *perspectives*. A perspective defines the initial set and layout of views and editors for the perspective. Each perspective provides a set of functionality aimed at accomplishing a specific type of task, or works with specific types of resources. Perspectives control what appears in certain menus and toolbars.

The default Inline Service perspective contains four views:

- **Inline Service Explorer view:** Shows the project and elements in tree form; by default, it is located on the left-hand side of the screen.
- **Problems view:** Shows errors and exceptions with your project; by default, it is located at the bottom of the screen as a tabbed selection, along with Test view.

  The Problems view identifies compilation errors and validation errors as the Inline service is built. Double-click a compilation error to display the Java perspective with the error highlighted.

  Double-click a validation error to display the Inline Service perspective with the element editor for the element that has validation errors.
- **Test view:** Provides an area for testing your Inline Service; by default, it is located at the bottom of the screen as a tabbed selection, along with Problems view.
- **Cheat Sheets view:** Provides step-by-step instructions for common tasks; by default, it is located on the right-hand side of the screen.

The center area of the Inline Service perspective is the editor area, and shows an editor that is specific to the node on the project tree you have selected. To change to a new editor, double-click the element you want to edit.

To edit a Java file, change to the Java perspective and double-click the Java file you want to edit.

The Inline Service perspective is the default perspective, and is the main work area for configuring and deploying Inline Services. Oracle RTD has a number of features for working with Inline Service metadata. These are documented in the following sections. If there is a feature you do not see here, it is part of the core Eclipse platform. For information about these features, see the Eclipse online help document *Workbench User Guide*.

Table 12–2 describes the menu and toolbar items for the Inline Service perspective.

Table 12–2 Menu and Toolbar Items for Inline Service Perspective

Menu or Toolbar Item Name	Description
File > New Inline Service Project	Creates a new Inline Service project in the workspace you choose.
Project > Download	Downloads an already deployed Inline Service from Real-Time Decision Server to make changes.
Project > Deploy	Deploys an Inline Service to Real-Time Decision Server.
Window > Open Perspective > Inline Service Perspective	Opens an Inline Service perspective.
Window > Show View > Inline Service Explorer View	Shows the current Inline Service View.
Window > Display Object IDs	Toggles between showing labels and Object IDs.
Help > About	Displays version information about Decision Studio.

Table 12–2 (Cont.) Menu and Toolbar Items for Inline Service Perspective

Menu or Toolbar Item Name	Description
Deploy icon: 	Deploys an Inline Service to Real-Time Decision Server.
Download icon: 	Downloads an already deployed Inline Service from Real-Time Decision Server to make changes.
Toggle icon: 	Toggles between showing labels and Object IDs.

The Inline Service Explorer View also has toolbar items. These items are described in Table 12–3.

Table 12–3 Toolbar Items for Inline Service Explorer View

Toolbar Item Name	Description
Metadata icon: 	Toggles between showing the entire project tree, or just Inline Service metadata.
Collapse All icon: 	Collapses the project tree.
Link with Editor icon: 	Finds the proper editor for the selected element type and links so that when you select an element, the editor adjusts accordingly.
Menu icon: 	Provides access to **Link with Editor**, **View Metadata Only**, and **Always Show Object IDs**. This last option shows both the Object ID and label of elements in the Inline Service Explorer and the editors.

The Java perspective combines views that you would commonly use while editing Java source files, while the Debug perspective contains the views that you would use while debugging Java programs.

To work directly with the generated Java code, use the Java perspective. To debug an Inline Service at the Java code level, use the Debug perspective.

12.2.6 Arranging Views and Resizing Editors

Tabs in the editor area indicate the names of resources that are currently open for editing. An asterisk (*) indicates that an editor has unsaved changes. Tabs on views indicate the name of the view, and have a toolbar that provides functionality specific to that view.

You may drag and drop the views and editors of a perspective to any space on the screen. Views and editors will resize themselves to fit the area in which they are placed. Occasionally portions of an editor (where you do your main work) or view will become covered by other views, or resized to an area that is not convenient to use. To resize the editor or view, either close some other open views and the remaining will automatically resize, or maximize the editor or view.

Both editors and views can be toggled between Maximize and Minimize by double-clicking the tab, or by using the right-click menu item. For more information

on perspectives, editors, and views, see the online documentation provided in the *Workbench User Guide*, contained in the Eclipse online help.

12.2.7 About Element Logic

Java code is added to the logic panels of elements within Decision Studio. This code is then inserted into the proper methods of the `GENObject_ID.java` file. To add logic to an element, or to update it, select the element, and use the editor to change the code in the logic panel.

Sometimes it is more convenient to insert larger code fragments directly within the generated code. You may edit these files directly through the Java perspective of Decision Studio. It is very important to note that the generated code can only be manually edited in specific places. Also, note that when you choose **Project > Clean**, Decision Studio regenerates the generated code, overwriting any code changes made directly to the generated Java code.

Any method that can be edited through the Java perspective in Decision Studio is clearly marked with a Start and End marker. For instance, the Application object has a method to initialize the Inline Service, `init()`.

Code for this method can be added through the Decision Studio interface, using the **Initialization Logic** panel on the **Logic** tab of the Application element.

If you choose, instead, to add your initialization code directly into the Application class using Eclipse, add it only to the method marked as such:

```
public void init() {
    // SDCUSTOMCODESTART.Application.InitBody.java.0
    // SDCUSTOMCODEEND.Application.InitBody.java.0
  }
```

Your code must fall between the start and end comments of the method. Any code that falls outside of the commented areas risks being overwritten. The code added directly to a generated Java file will be lost when the file is regenerated. To preserve the code, it has to be copied back to the corresponding metadata element.

12.2.8 Overriding Generated Code

The generated class `Object_ID.java` extends the class `GENObject_ID.java`. If for any reason you need to override the code contained in `GENObject_ID.java`, add your overriding code to the file `Object_ID.java`. This file should be moved from the `gensrc` directory to the `src` directory.

12.2.9 Performing Inline Service Searches

You can perform searches for objects and strings in an Inline Service., as follows:

1. Start the procedure through the menu path Search > Search.
2. Click the Inline Service Search tab.
3. Enter your Search string, then refine your search criteria as required:
   - You can search for a combination of Object IDs and all other strings.
   - By default Inline Service searches are case insensitive. You can check one or more of the following Search Options as required:
     – Case sensitive - the entered expression is searched with the capitalization as entered

- Match whole words only - the expression is searched where it appears on its own, and not as part of a longer word
- Regular expression - use a Java regular expression for the search string

4. Click **Search**.

   The results appear in the Search view.

12.3 About Decision Studio Projects

Inline Services are built as *projects* within Decision Studio.

This section contains the following topics:

12.3.1 Starting a New Project

To start a new Inline Service, select **File > New Inline Service Project** to start your project. Choose a template from the list, name your project, and click **Finish** to create a project.

The list of templates contains templates supplied by the Oracle RTD installation, as well as any user-defined templates.

12.3.2 Importing a Project

If you are opening an existing project, select **File > Import** to import the project. If the metadata needs to be updated from a previous version, you will be prompted to upgrade.

12.3.3 Creating a User-Defined Template

To create a template from an Inline Service, select **File > Export** to export the project to a template. Choose the export type **Inline Service Template**. Templates are stored in the location defined by Inline Services Preferences. To access preferences, select **Window > Preferences** and choose **Inline Services**. The directory entered is where your templates are stored on the file system.

12.3.4 Downloading a Deployed Inline Service

To download a deployed Inline Service, select **Project > Download**. You can also download it from Real-Time Decision Server using the Download icon on the toolbar.

If you are going to make changes to a deployed Inline Service, it is important to follow these practices in order to preserve both your changes and the potential changes that have been made by business users. Use the following method:

1. Make sure that no business users are editing the deployed Inline Service.

2. You should always lock an Inline Service when you download, so that additional changes cannot be made by business users while you are enhancing it.
3. Make enhancements in Decision Studio.
4. Redeploy the Inline Service, releasing the locks.

During the period that you have the Inline Service locked, business users will be able to view, but not edit, the deployed Inline Service.

12.3.5 About Deployment States

When an Inline Service is deployed from Decision Studio, you chose a deployment state from the deploy dialog. Three deployment states are packaged with Decision Studio: Development, QA, and Deployment. Your system administrator may add additional deployment states through JConsole.

When you test your Inline Service through the Test View, the last deployment state is tested.

12.3.6 Example Projects

A sample project is available to import in the *RTD_HOME*\examples directory. This directory includes the Cross Sell Inline Service.

The Cross Sell Inline Service simulates a simple implementation for a credit card contact center. As calls come into the center, information about the customer and the channel of the contact is captured.

Based on what we know of this customer, a cross selling offer is presented to the customer. The success or failure of that offer is tracked and sent back to the server, so that the underlying decision model has the feedback that helps to refine its ability to make a better cross-selling recommendation.

The Cross Sell Inline Service highlights many features of Oracle RTD, including:

- Driving the decisioning process through Key Performance Indicators (KPIs)
- Optimizing competing KPIs, such as reducing cost and increasing revenue
- Using graphical rules-based scoring for making the right decision
- Using analytical self-learning models to predict the best decision

It should be noted that some features displayed in the Cross Sell Project are for simulation purposes only. These are clearly marked in the example and should not be used for production Inline Services.

The Cross Sell example can be viewed by importing the project. The project is located at *RTD_HOME*\examples\CrossSell. After importing the project, you can view the features described in Table 12–4 by double clicking each of the elements and viewing the editor for that element.

Table 12–4 Features of the Cross Sell Example Inline Service

Feature	Element Name	Description
Multiple KPIs	Performance Goals	The Cross Sell Inline Service is designed to optimize both the maximization of revenue and the reduction of costs of the organization.

Table 12–4 (Cont.) Features of the Cross Sell Example Inline Service

Feature	Element Name	Description
Dynamic customer data	Data Source/CustomerData SourceEntity/Customer	The combination of a Data Source and an Entity give access to customer data that will assist us in making a decision of the type of offer to present to the customer. An *Entity* is an object that provides a means to map one or more Data Sources together into an object that represents a significant unit in the Inline Service. The data accessed through the Entity is session specific.
Cross Selling and Customer Retention Offers	Choices	The offers that are available to be extended are organized under Choices. Some of these offers are designed as cross selling offers, while others are designed to boost customer retention rates. By viewing the Score tab of each offer, you can see that offers are assigned a score for evaluation. A Score is provided for each performance goal, Revenue and Retention. Some offers (for instance all Credit Cards) inherit their scoring from the parent Choice Group. This indicates that all offers in this group are scored in the same manner. In this case, the score is calculated by the formula 'Profit Margin multiplied by Likelihood of Acceptance.' Other offers (such as Reduced Interest Rate) calculate the score using a rule. Note that the Revenue Goal on Reduced Interest rate is actually scored negatively, as it represents a loss of revenue to the organization.
Scoring Rules	Scoring Rules/Reduced Interest Rate	Scoring Rules are a way to use session data, such as information about the customer, to dynamically score the offer.
Population segment	Filtering Rules/Segment to Retain	The population can be segmented by using Filtering Rules. The outcome of this rule is two groups: a group that is eligible for customer retention offers and the remaining group to which we will cross sell. If a customer has abandoned six or more calls and has been a customer for over two years, they are filtered into a group for retention offers.

Table 12–4 (Cont.) Features of the Cross Sell Example Inline Service

Feature	Element Name	Description
Weighting decisions by population segment	Decisions/OfferDecision	The Decision element allows you to weight the decision process across the competing performance metrics. In this case, we give priority to the offers that score high on Customer Retention a heavier weight for the population segment that fits the customer retention profile.
Integration to organizational processes	Integration Points	Integration Points are the sites that touch outside systems and processes, either by gathering information (such as CallStart, which gathers information from the IVR about the customer) or provides information to an outside system (such as OfferRequest, which provides the CRM system with the highest scored offer for the customer). It should be noted that the OfferResponse Integration Point has code in the `else` branch for simulation purposes. In a production situation, this would be feedback from the service center operator on whether the offer was accepted or not.

Caution: In order to simulate the passage of time when the Inline Service load generation script is run, the method `currentTimeMillis` has been overridden in `Application.java`. If you plan on using CrossSell as a basis for a production Inline Service, you need to remove the following override file:

RTD_HOME`\examples\CrossSell\src\com\sigmadynamics\sdo\Application.java`

See Section 12.2.8, "Overriding Generated Code" for more information.

The Cross Sell Inline Service is ready to be deployed and loaded with data. After you deploy the Inline Service, open Load Generator by running *RTD_HOME*`\scripts\loadgen.cmd`. Then, choose **Open an existing Load Generator script** and browse to *RTD_HOME*`\examples\CrossSell\etc\LoadGen.xml`. Finally, run the script.

This script takes simulated customer data and runs the Inline Service. The data and correlations found can then be viewed in Decision Center.

12.3.7 Opening Decision Studio Version 1.2 Files

If you are opening an Inline Service from a previous version of Decision Studio, it was not created as a project. To open it as a project, start a new Inline Service project and then select **Create project at external location** to locate the files on your file system.

This will convert the previous version Inline Service to a Decision Studio 2.2 project.

12.4 Inline Service Directory Structure

When you create your Inline Service, you can create your project anywhere on your file system. It is recommended that you keep all of your projects in one directory for ease of use. The default workspace is `C:\Documents and Settings\user_name\Oracle RTD Studio`.

When saving an Inline Service, the directory name is the same as your Inline Service name. The following directory structure is created for your Inline Service.

Table 12–5 Inline Service Directory Structure

Directory Name	Description
Inline_Service_name\classes	The compiled classes of your Inline Service.
Inline_Service_name\dc	A folder for custom JSPs for Decision Center. The custom JSPs can be accessed though a URL of the form *http://<host_name_or_ip>:<port>/ui/custom/<ILS-name>/<custom page file name>*. For example: *http://localhost:8081/ui/custom/CrossSell/mypage.jsp*.
Inline_Service_name\etc	Optional directory for miscellaneous files related to the Inline Service. For example: ■ Loadgen scripts ■ Readme files ■ Inline Service description files ■ Inline Service setup instruction files This directory is not pushed to the Real-Time Decision Server when the Inline Service is deployed.
Inline_Service_name\gensrc	Location of the generated source code for your Inline Service.
Inline_Service_name\meta	The metadata of your Inline Service.
Inline_Service_name\src	The source code for overriding the generated code of your Inline Service.

12.5 Configuring Inline Services

Inline Services are configured and deployed using Decision Studio. Elements are added to a project and Java scriptlets with functional logic are added to certain elements. When the Inline Service is saved, XML metadata is created and Java code is generated and deployed to Real-Time Decision Server, where the Inline Services runs.

This section contains the following topics:

12.5.1 Observer Inline Services

In monitoring, Inline Services focus on collection points: points where data about the business can be gathered. Insights and discoveries of trends and correlations in this data are made by a self-learning model that predicts future behavior and anticipates the consequences of change. This type of Inline Service is known as an Observer.

These discoveries are published to a thin client, Decision Center, where business users use these insights to make decisions. Business users also manage and optimize the Inline Service through Decision Center.

12.5.2 Advisor Inline Services

In advising a business process, Inline Services connect at key decision points as well as collection points. Decision points are places in the overall business process where key business decisions are made, such as product recommendations or retention offers. Data is first gathered at collection points, discoveries are made through the self-learning model, and then choices are scored according to performance metrics the organization wants to achieve. The highest scored choice is presented by the Inline Service at the decision point in the business process. As the success level of these choices is returned to the Inline Service through feedback, the model increases its capability of providing better and better choices. Inline Services of this type are called Advisors.

13

About Decision Studio Elements and APIs

Decision Studio elements are configured within Decision Studio, and the logic is added in the form of Java scriptlets. This chapter describes the properties of each element, and the Java scriptlets contained by the element (if any), with examples.

This chapter contains the following topics:

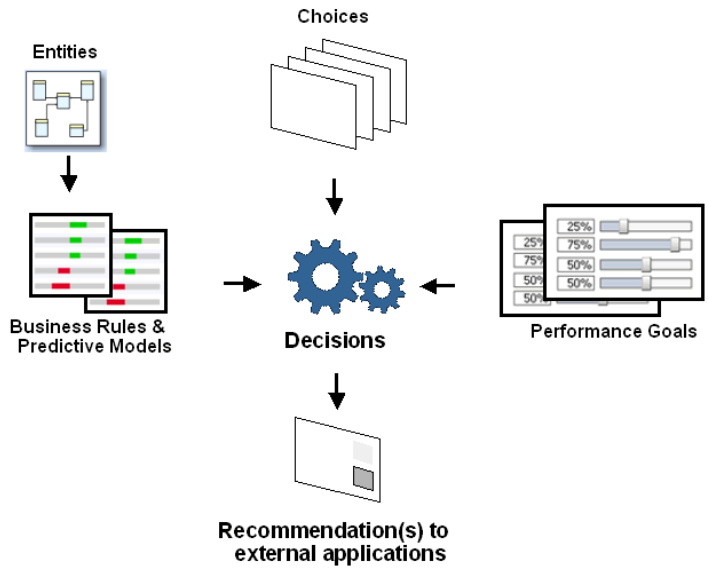
13.1 The Oracle RTD Decisioning Process

The Oracle RTD decisioning process is based on a framework that takes into account the overall performance goals with which an organization is concerned, the performance metrics that measure those goals, the action required to score each of the available choices, and a weighting of that score based on segments of the population.

The following elements are part of this framework:

- Performance Goals
- Decisions
- Choice Groups and Choices
- Filtering Rules
- Scoring Rules
- Predictive Models

The following shows an overview of how the elements feed into the general Oracle RTD decisioning process, and form the basis for an Inline Service:



To see how extensions of these inputs can enable external applications together with Oracle RTD to provide a composite decision service for their end users, see Chapter 17, "Externalized Objects Management."

13.2 About Element Display Labels and Object IDs

As you create elements, you enter a Display Label for the element. An Object ID is automatically generated as you type the Display Label.

Object IDs are automatically made to conform to Java naming conventions: variables are mixed case with a lowercase first letter; classes are mixed case with an uppercase first letter. If you have spaces in your label name, they will be removed when forming the Object ID. If you choose to manually enter an Object ID, you can overwrite the Object ID that was created for you.

You can use the Toggle icon on the Inline Service Explorer task bar to toggle between the Display Label of the object and its Object ID:

Note: When creating a new object, if the object name is already used by an existing one, Decision Studio will automatically append a number to the Object ID (such as 1, 2, and so on) to avoid conflicts.

13.3 About the Application Element

When a new project is started in Decision Studio, an Application object is created under the Service Metadata folder. Properties of the Application object are defined with the following characteristics:

- Application Parameters
- Control Group
- Model Defaults
- Logic
- Permissions

All of these values are defined through the Decision Studio interface.

This section contains the following topics:

13.3.1 Application Parameters

This section describes Application parameters. This section contains the following topics:

13.3.1.1 Using Debugging Options

If you are testing a deployed Inline Service against a production database, and you do not want to contaminate the model data, you can use the debugging options to keep data from being written. Debugging options are:

- **Disable learning:** This option maintains the model's current state so that testing does not introduce additional learnings.
- **Disable database writes:** This option keeps data from being written to the database.

Parameters have a name, data type, default value, and can be made an array.

13.3.1.2 Adding Application Parameters

Application parameters are global-level parameters that can be defined and stored across all sessions.

Click **Add** on the **Application Parameter**s tab to add a parameter. When adding parameters, you supply the Name, Type, Array, and Default Value.

If you want to use an application parameter in a rule, you can select a Type Restriction for the parameter. This is not a mandatory requirement, but it will help you in

formulating the rule. For more information about creating and using type restrictions, see Section 13.18, "About Type Restrictions."

Click **Remove** to remove parameters.

13.3.2 Application APIs

The following returns the Object label and Id, respectively:

```
public String getSDOLabel();
public String getSDOId();
```

When global parameters are set, getters and setters are generated in the code. To access the application parameter, for instance `myApplicationParameter` of type `string`, use the following:

```
String param =  Application.getApp().getMyApplicationParameter();
```

Conversely, to set an application parameter:

```
Application.getApp().setMyApplicationParameter("my parameter");
```

13.3.3 Configuring the Control Group

The *control group* acts as a baseline so that business users can compare the Oracle RTD decisioning process against either a preexisting ranking process or a random selection. It is important to define the control group decision correctly to truly reflect the decision as it would have been made if Oracle RTD was not installed.

For example, in a cross-selling Inline Service for a call center, if cross-selling was being done randomly before Oracle RTD was introduced, the Control Group Decision should also reflect a random decision.

Table 13–1 Control Group Options

Option Name	Description
No Control Group	If selected, control group will not be used, the **Selection Value** will be set to 0, and **Use value literally** will be selected and disabled. Deselect **No Control Group** to turn on control group and to change the settings.
Selection Value	A reference to an attribute value. This value is used to seed the Random selection of requests for the control group.
Use value literally	If **Use value literally** is deselected, then the selection value for the control group should refer to a session key or an attribute uniquely identifying a customer. Control group participation is determined using a pseudo-random hash of the selection value. The result of the calculation is deterministic, and depends only on the selection value and the specified size of the control group. The actual size of the control group may differ slightly from the specified size. If **Use value literally** is selected, then the selection value directly determines the control group participation. The selection value in this case can be either Boolean (participation in control group is indicated by the true value) or integer (participation in control group is indicated by a non-zero value). For example, select **Use value literally** when assignment of customers to control group is done outside of Oracle RTD. The attribute used as the control group selection value has to indicate this assignment.

Table 13–1 (Cont.) Control Group Options

Option Name	Description
Percent of Population	This option is only active if **Use value literally** is set to False. In this case, the user decides what percentage of the total number of sessions should be assigned to the control group.
Use for analysis	Controls whether the control group participation should be tracked by analytic models or not.
Name for Analysis	Name that the analytic models should use for tracking the control group participation.

13.3.4 Setting Model Defaults

Model defaults control which model is used and how the model is set up. Most model defaults should not be changed unless you are advised to do so by Oracle Support Services.

Table 13–2 Model Defaults

Option Name	Description
Study name	The name of the study used by the Inline Service. Typically, each Inline Service has its own separate study. This can be achieved by keeping the name of the study the same as the name of the Inline Service. However, an existing study may be used when testing an Inline Service. In that case, learning should be disabled to preserve production data.
Persistence Interval	The interval at which model data is saved to a database. This feature should not be adjusted without assistance from Oracle Support Services.
Time Window Duration	The default value is Quarter.
First Day of Week	The default value is locale dependent. In the United States, the default value is Sunday.
First Month of Year	The default value is January.
Build when data changes by	The percentage of new records to prompt the building of a new prediction model. The default value is 20%. For example, if set to 10% and the last time the prediction models were built was with 13400 records, then the next time they will be built will be after 1340 records.
Significance threshold	The default value is 25. This feature should not be adjusted without assistance from Oracle Support Services.
Correlation threshold	The default value is 0. This feature should not be adjusted without assistance from Oracle Support Services.
Max Input Cardinality	The maximum number of values that will be tracked in discrete attributes. The default value is 500. This feature should not be adjusted without assistance from Oracle Support Services.
Max Input Buckets	The maximum number of input buckets for numeric attributes. The default value is 200. A value of 100 may also be sufficient. This feature should not be adjusted without assistance from Oracle Support Services.

13.3.5 Writing Application Logic

Scriptlets to initialize and clean up an Inline Service are added through Decision Studio, using the **Initialization Logic** and **Cleanup Logic** panels on the **Logic** tab of the Application element.

These scriptlets are inserted into the `init` and `cleanUp` methods of the Application class. The `init` method is called when the Inline Service is being loaded. The method `cleanUp` is called when the Inline Service is unloaded. If the application is redeployed, `init` will be called again.

13.3.5.1 Adding Imported Java Classes

If the `init` or `cleanUp` method refers to user-provided Java classes, these classes may have to be imported. To add additional import statements, click **Advanced** next to the description.

13.3.6 Setting Inline Service Permissions

Inline Service permissions are set for roles that allow users to work with the Inline Service during its lifecycle. The permissions listed in Table 13–3 are available for each Inline Service, and are appropriate for Inline Service developers and Decision Center business users.

Table 13–3 Inline Service Permissions

Permission Name	Description
Open Service for Reading	Allows Decision Center to open the Inline Service in a read-only mode. This mode is appropriate for a business user who will use Decision Center to view reports.
Open Service	Allows a Decision Center user to edit and redeploy an Inline Service to Real-Time Decision Server.
Deploy Service from Studio	Allows a Decision Studio user to deploy the Inline Service. To redeploy an existing Inline Service, the user has to be assigned a role with Deploy permission for both the existing and the new service. This mode is appropriate for the Inline Service developer who will use Decision Studio to deploy Inline Services.
Download Service	Allows a Decision Studio user to download an existing, deployed Inline Service from a server. This mode is appropriate for the Inline Service developer who will use Decision Studio to deploy Inline Services.

Inline Service permissions work with server-side Cluster Permissions to secure the Inline Service from being changed or redeployed by an unauthorized user. See *Oracle Real-Time Decisions Installation and Administration Guide* for more information on setting Cluster Permissions. In addition, you can set permissions on Decision Center perspectives. See Section 13.20, "About Decision Center Perspectives" for more information.

Use the **Permissions** tab of the Application element to set Inline Service permissions. Click **Add** to add roles to the Inline Service. To retrieve roles from the server, click **Get Names**.

You can choose a role from the list, or you can enter a name in the **Role** field. After you have added roles, select the permissions you would like to apply under the **Granted** field in the Permissions list.

To remove roles from the Inline Service, select the role and click **Remove**.

Note: When Windows Authentication is enabled, you cannot retrieve a list of roles by clicking **Get Names**. Instead, you must enter the name of the role to which you want to assign permission in the **Role** field.

13.4 Accessing Data

To access data within your Inline Service, use the *entity*, *data source*, and *session* elements.

Entities provide an abstract way to bring together data from multiple sources to form an object that is of use to the overall Inline Service. Entities are comprised of a number of *attributes* that describe the contents of the Entity. Entities also provide methods to access their attributes.

An Entity, such as a Customer, may combine incoming data from different sources, such as an account number entered through an IVR and customer history retrieved from a corporate database. One of the Entity's attributes may be a *key*, or unique identifier. The incoming data would then be mapped to a desired attribute of the Entity. Alternatively, entity attributes can be populated through the life of the session via additional logic coded in the inline service

Data sources act as *suppliers* of data. They provide a way to manage the connection to relational database tables and stored procedures. Data sources identify columns in a database table, or result sets from stored procedures as attributes. These attributes are mapped to Entity attributes.

The session object is a specialized Entity that identifies which attributes are available in memory to the Inline Service. Those attributes can be comprised of a Entity, such as Customer described previously, as well as attributes that are set by some other source, such as calculation. A session object is used to store information for a user session. Attributes stored in session are available throughout the Inline Service, and are destroyed when the session is closed. Attributes associated to the session will by default be used as inputs for Oracle RTD learning models unless explicitly excluded from the specific model or excluded for analysis altogether.

To access data, you typically follow these steps:

1. Create a data source, based on a SQL table or stored procedure.
2. Create an Entity with attributes from one or more data sources.
3. Add a key value to the Entity.
4. Add the Entity to the session as an attribute, and assign a session key.
5. Map the Entity attributes to data source columns or output values.
6. Map the Entity key to a session key or function.

This section contains the following topics:

13.4.1 About Data Sources

Data is accessed within Inline Services using the elements data source and entity. A data source is an abstract provider of data. Data sources act as suppliers of data to the Inline Service.

Data sources are configured entirely within Decision Studio. There are two types of data sources: SQL data sources, and Stored Procedure data sources.

13.4.2 Creating SQL Data Sources

This section describes how to create a SQL data source. This section contains the following topics:

13.4.2.1 SQL Data Source Characteristics

Table 13–4 lists the properties of a SQL data source.

Table 13–4 Properties of a SQL Data Source

Data Source Property Name	Description
Description	Description of the data source.
JDBC Data Source	The JNDI name of a JDBC data source. See *Oracle Real-Time Decisions Installation and Administration Guide* for information about how to create a new data source.
Table Name	The name of the table. This name is always case insensitive, even when the database itself is case sensitive.
Output Column Name	The columns to select from the data source.
Output Type	Data type of the output column.
Input Column Name	The columns used in the WHERE clause of the query to the data source. This is the column or columns you will match on in order to select data from the data source.
Input Type	Data type of the input column.
Allow multiple rows	Allows multiple rows to be returned. If this option is not selected and multiple rows are returned, only the first one is used.
Advanced	The Advanced button lets you choose to show the element in Decision Center and change the label of the element. Changing the label of the element does *not* change the Object ID.

13.4.2.2 Adding Columns to the Data Source

Click **Add** or **Remove** to add or remove columns from the data source. If you expect more than one row, select **Allow Multiple Rows**. If you do not select this option and multiple rows are returned, only the first will be used.

13.4.2.3 Importing Database Column Names

Click **Import** to connect directly to the data source. All of the database tables for the specified data source will be shown. If no data source is specified, the default data source SDDS is used.

Select **Include objects from all schemas** to display tables not defined in the data source schema. Tables from all accessible schemas will be shown, with the table schema displayed in a separate column.

Choose the table you want to import, and the column names and data types of those columns are imported. If there are columns you do not need, click **Remove** to remove them.

13.4.2.4 Setting the Input Column

The **Input column** is the column you will match on the database table to retrieve the rows needed for the session. Most likely, this will be a value of the primary key or a unique index column to return a single record. Otherwise, if you need larger result sets, it may be a non-unique indexed column. Choose the attribute on which you want to match by clicking **Add**.

13.4.3 Creating Stored Procedure Data Sources

This section describes how to create a stored procedure data source. This section contains the following topics:

- Section 13.4.3.1, "Stored Procedure Data Source Characteristics"
- Section 13.4.3.2, "Importing Stored Procedure Parameters"
- Section 13.4.3.3, "Adding Attributes to the Data Source"
- Section 13.4.3.4, "Adding Result Sets to the Data Source"

13.4.3.1 Stored Procedure Data Source Characteristics

Table 13–5 lists the properties of a stored procedure data source.

Table 13–5 Properties of a Stored Procedure Data Source

Data Source Property Name	Description
Description	Description of the data source.
JDBC Data Source	The JNDI name of a JDBC data source
Procedure Name	The name of the stored procedure. This name is always case insensitive, even when the database itself is case sensitive.
Inputs and Outputs	Input and output parameters for the stored procedure. Each Input and Output has a Name, a Type, and a Direction.
Result Sets	The result sets from the stored procedure.
Allow multiple rows	Allows multiple rows to be returned. If this option is not selected, and multiple rows are returned, only the first is used.
Result Set Details	The column names and type of the results expected.
Advanced	The Advanced button lets you choose to show the element in Decision Center and to change the label of the element. Changing the label of the element does *not* change the Object ID.

13.4.3.2 Importing Stored Procedure Parameters

Click **Import** to connect directly to the data source. All of the stored procedures for the specified data source will be shown. If no data source is specified, the default data source SDDS is used.

Select **Include objects from all schemas** to display stored procedures in all the data source schemas.

Choose the stored procedure that you want to import, and the parameter names and data types of those parameters are imported, to become attributes of the data source, then click **Finish**. If there are parameters that you do not need, click **Remove** to remove them.

13.4.3.3 Adding Attributes to the Data Source

Click **Add** or **Remove** to add or remove attributes from the data source. Choose whether the attribute is an Input, Output, or Input/Output.

Attributes must be ordered. Use **Up** or **Down** to order the attributes.

13.4.3.4 Adding Result Sets to the Data Source

If the stored procedure has one or more result sets, perform the following steps for each result set:

1. Click the Result Sets **Add** button to add a result set to the data source.
2. Use the Result Set Details **Add** button to add the column names and types of the result set.

   You must manually enter the column names and data types to match the columns in the stored procedure result set, as follows:

   - The column names in the data source must be exactly the same as the column names in the result set
   - The data types in the data source must be valid for the corresponding data types in the result set

     For example, for VARCHAR or VARCHAR2 result set columns, enter String for the corresponding data source column data types. For FLOAT columns from SQL Server or Oracle stored procedures, and REAL columns from DB2 stored procedures, enter Double for the corresponding data source column data types.

13.4.3.5 Examples of Setting Up Data Sources from Stored Procedures

Appendix B, "Examples of Data Sources from Stored Procedures" shows examples of setting up data sources from Oracle, SQL Server, and DB2 databases, and creating entities that derive their attributes from these data sources.

13.4.4 Accessing Oracle's Siebel Analytics Data

Oracle's Siebel Analytics Server exposes an ODBC client interface for accessing data stored in OLTP and OLAP databases. The RTD Decision Service uses the JDBC-ODBC bridge included in Java Runtime Environment (JRE) to connect to the ODBC driver provided by Siebel Analytics Server.

From the RTD Decision Service point of view, Siebel Analytics Server is a SQL data source similar to a regular database. Subject areas in Siebel Analytics Server are

treated as database tables by the Inline Service. Column names are compound, combining two levels of the presentation object hierarchy.

See *Oracle Real-Time Decisions Installation and Administration Guide* for information about how to add a JDBC data source that can be accessed by a Decision Studio data source.

13.5 Forming Entities

An *Entity* is a set of named attributes and methods to access those attributes. One attribute is usually designated as the Entity's key. For example, a simple customer Entity might look as follows:

```
Customer

customerId: string, key
name: string
age: integer
accounts: collection of Account entities
```

In this Entity, the *customerId* is the key of the Entity, *name* and *age* are simple attributes, and *accounts* is a collection of Account entities that has been associated under the Customer header.

This section contains the following topics:

13.5.1 About the Session Entity

The Session entity is a specialized Entity that is automatically created for every Inline Service.

The Session entity identifies which attributes are available in memory to the Inline Service. Those attributes can be comprised of a Entity, such as Customer, as well as attributes that are set by some other source, such as calculation. A Session object is used to store information for a user session. Attributes stored in a session are available throughout the Inline Service, and are destroyed when the session is closed.

Sessions are closed either explicitly by an Integration Point, or when the session times out.

13.5.1.1 About Session Keys

A session key is a field passed in the request that identifies an instance of a Real-Time Decision Server server-resident Session object that will be available to the Integration Points.

As an example of using sessions, consider a client Web application, where each request supplies the CustomerId of the Web application as the session key. When the first request arrives with a new CustomerId, Real-Time Decision Server notices that this session key is new, and consequently creates a new Session object and makes it available to the Integration Point as it executes the request. Any subsequent requests using the original CustomerId will access the original Session object.

The processing of the Integration Points may implicitly or explicitly save information in the session, so that it will be available to subsequently invoked Integration Points.

13.5.2 Creating Entities

Entities are defined using Decision Studio. Entity names must begin with an uppercase letter. Entities are defined with the following characteristics:

- **Description:** Description of the entity as entered in Decision Studio.
- **Key:** A unique ID for the entity. Click **Add Key** to add a key to an entity.

There are also properties for entity attributes. These attribute properties are listed in Table 13–6.

Table 13–6 Entity Attribute Properties

Entity Attribute Property Name	Description
Description	Description of the attribute as entered in Decision Studio.
Type	Attribute types are either primitive types, Java classes, Entity types, Choices, or Choice Groups.
Array	Whether single-valued, or a collection.
Type Restriction	If you want to use an entity attribute in a rule, you can select a Type Restriction for the attribute. This is not a mandatory requirement, but it will help you in formulating the rule. For more information about creating and using type restrictions, see Section 13.18, "About Type Restrictions."
Default Value	The default value, which can be a constant, a function, or a reference to an attribute.

Table 13–6 (Cont.) Entity Attribute Properties

Entity Attribute Property Name	Description
Use for analysis	Select this option to use this attribute for analysis within the predictive model.
Category	The category of the attribute. Categories help organize the display of attributes in Decision Center.
Analysis options	Additional analysis options are available for Date attributes. To use Dates for analysis, specify the pattern you are interested in analyzing. The effect of month, day of month, day of week, and time of day can be analyzed separately, or in any combination.

13.5.3 Adding Attributes and Keys to the Entity

Click **Add Attribute** or **Add Key** to add attributes to the Entity. If the attribute is a collection, select the **Array** column.

Caution: When adding a Key attribute, the data type will automatically be String. If the data type of your data source column or output parameter is a type other than String, use a transformation function when you set the input on the data source.

13.5.4 Importing Attributes From a Data Source

To automatically add all of the output columns of a data source as Entity attributes, click **Import**, then choose a data source from which to import. If you would like to import from more than one data source, repeat the procedure. Click **Remove** to remove any unwanted attributes.

When using **Import**, select **Build data mappings for selected data source** to automatically map the attributes to the data source. If the entity is nested (for example, in a one-to-many relationship) and the attributes are mapped indirectly, deselect this option.

Note: While the Build data mappings for selected data source functionality maps the entity attributes to the data columns, users may still assign input values under the **Data Source input values** section of the Entity editor.

13.5.5 Using Attributes for Analysis

Select **Use for Analysis** to have the attribute added to the analytical model.

By default, the **Use for Analysis** option for each entity attribute will be set to true, and will be used as an input to Oracle RTD's models. If this is not desired, users may explicitly uncheck this option. Right click the desired attribute and select **Properties** to access this setting.

13.5.6 Decision Center Display

The option **Show in Decision Center** is selected by default. Deselect this option if you want the attribute to be hidden from Decision Center users. If desired, choose a **Category** to control the display of the attribute in Decision Center. Right click the desired attribute and select **Properties** to access these settings.

13.5.7 Adding a Session Key

If the session key value that you choose to use is an attribute of an entity, first add the entity to the session. To do this, click **Add attribute** in the session entity, and add the entity as a new attribute of that entity type to that session.

For instance, assume you want to make the session key the `customerId` attribute from the Customer entity. Click **Add Attribute**, then add an attribute to the session called `customer`. The type of this attribute is an entity type, namely `Customer`.

To access the entity types, use the dropdown list on the **Type** column and choose **Others**. The Type window appears. Choose the **Entity Type** for this attribute.

To add the session key, click **Select** from **Session Keys from Dependent Entities**. All key values from entities that are attributes of the session are available to be chosen as a key value for the session. Choose the key on which you want to base the session, in this instance `customerId`.

Note: The keys of cached entities cannot be used as session keys.

13.5.8 Adding Attributes to the Session

Click **Add Attribute** to add an attribute that you want to make available for the entire session. Session attributes have getters and setters generated for them, as do other entity attributes.

13.5.9 Mapping Attributes to Data Sources

After creating an entity using Decision Studio, map the attributes of the entity to values that are either constant, calculated, a reference to an attribute of a data source, or to the session key.

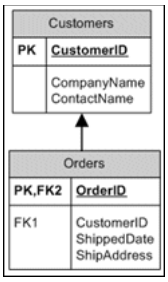
Mapping to configured data sources is done through the **Mapping** tab of the entity object. To map the attributes of an entity to a data source, use the **Source** column to choose the path to the data source column or output parameter that will supply the attribute with data.

To map the key value, click **Input Value** from **Data Source Input Values**. Your key value will appear here when you map the attributes to data source values. You can map the key to a session key attribute, to another entity key value, or to a function. The input type must be of type String. If it is not, use a function to transform the non-string value.

13.5.10 One-to-Many Relationships

To access data in an Entity in a one-to-many foreign key relationship, make the related Entity an attribute of the first Entity. For example, say that the Customers table has a key, CustomerID. Customers have many Orders, which are identified by OrderID and a foreign key CustomerID.

Figure 13–1 Entity Mapping Example



To access data in an Entity in a one-to-many foreign key relationship, using Customers and Orders as an example, perform the following steps:

> **Note:** In this example, it is assumed that there is a CustomerID foreign key on the order table. This CustomerID serves as the input column to the Orders data source. The Orders data sources would also have its **Allow Multiple Rows** setting set to true.

1. In Decision Studio, define a data source for each of these tables.
2. Create an entity for Customers and Orders.
3. Add Customer to the session, as that is the key to retrieving the next level of data.
4. Choose CustomerID as the session key.
5. To associate the one-to-many relationship between Orders and Customers, add an attribute to Customer called Orders, of entity type Orders. Since there are many Orders for one Customer, make it an array.
6. You can map all of the attribute values through the Customer entity mapping tab.

13.5.11 Adding Imported Java Classes

To add imported Java classes to your Inline Service, click **Advanced** next to the description.

13.5.12 Session Logic

The session element can accept scriptlets that are executed on initialization and exit of the session. The cleanup scriptlet is executed when the session is closed.

13.5.13 Session APIs

The following code returns the Object label and ID, respectively:

```
public String getSDOLabel();
public String getSDOId();
```

You can use `session()` to access the other entities of the Inline Service. For example:

```
session().getCustomer().getCustomerId();
```

where `Customer` is an entity and `customerId` is an attribute of that entity.

Use `session()` to access the instance of the application session. Session has the following APIs available:

```
public boolean isTemporary();
```

If no session keys have been passed in, the session is considered temporary.

The following code is used to access an Integration Point request outside of the Integration Point and returns the current request:

```
public IntegrationPointRequestInterface getRequest();
```

The following code returns whether the current instance of the session has been closed:

```
boolean isClosed();
```

The following code returns any known session keys. This may or may not be the complete set, depending on where it is called.

```
Set getKeys();
```

The following code closes the current session instance:

```
public void close();
```

The following code gets the application object of the current session:

```
public ApplicationInterface getApp();
```

13.5.14 Entity APIs

The following code returns the Object label and Id, respectively:

```
public String getSDOLabel();
public String getSDOId();
```

13.5.15 About Entity Classes

In addition to the normal classes generated, an array class is also generated for each Entity. The generated classes have a property, getter, and setter for each attribute. Hence, the definition of entities such as Customer, Account, and Call will result in classes with these names, as well as another class, representing a collection of that class.

For instance, for the Account entity, the following two classes are generated:

```
Account
SDAccountArray
```

The second class represents a collection of Accounts. So, when our Customer entity has an attribute named accounts of type Account (with multiplicity set to multiple), then the following gets generated for Customer:

```
Customer  {
SDAccountArray getAccounts() {
  }
void setAccounts(SDAccountArray accounts) {
  }
void addToAccounts(Account account) {
  }
}
```

13.5.16 Referencing Entities in Oracle RTD Logic

Because a class is generated for each Entity type, you create an Entity with the new operator as you would any Java object. For example:

```
Customer cust = new Customer();
```

Optionally, if the Entity is the child of another Entity, pass in the name of the parent Entity. Session can be a parent Entity of any other Entity.

```
Customer cust = new Customer(entityParent);
```

13.5.17 Adding Entity Keys

Most Entities are not very useful unless they have a value for the key attribute. The key attribute, as with any attribute, is set using a generated setter:

```
Customer cust = new Customer();
String newKey = '12345';
cust.setCustomerId(newKey);
```

13.5.18 Accessing Entity Attributes

As mentioned previously, getters are generated for each attribute. The form of the getter depends on whether the attribute has one value or more than one value. The sample Customer entity would have the following getters:

```
String id = cust.getCustomerId();
String name = cust.getName();
double age = cust.getAge();
Collection accounts = cust.getAccounts();
```

Corresponding setters are also generated. We have already seen the setter for `customerId`, but here are the others for our Customer example:

```
cust.setName("Fred Johnson");
cust.setAge(42);
cust.setAccounts(newCollection);
```

And, because Accounts is an attribute that has multiple values, you can also add to the collection:

```
cust.addToAccounts(anotherAccountObject);
```

An array can be added using:

```
cust.addAllAccounts(anotherAccountArray);
```

13.5.19 Resetting and Filling an Entity

Three special methods are provided to reset and to fill an Entity.

```
cust.reset();
```

Resets all keys except session keys and all attributes.

```
cust.resetAttributes();
```

Resets all attributes, but does not reset keys.

```
cust.fill();
```

`Fill` recursively fills the values of the attributes according to the Entity mapping, forcing it to refresh the data through the data source, calculated or constant value. Any attributes of Entity type are also filled.

Reset and fill should not be called on cached entities.

13.5.20 About Cached Entities

Entities can be cached on the server so that they are easily accessible. To cache entities, click the **Cache** tab of the entity.

Entities can have the following caching options:

- **Enable caching for this entity type:** Select this option to enable caching. Cached entities are treated exactly like non-cached entities and have the same API, except that cached entity keys may not be used as session keys.
- **Max number of items to cache:** The maximum number of items to cache. Items are flushed in a first in/first out manner.

In addition, entities can have the following caching strategy options:

- **Use fixed lifetime:** Number of seconds each object stays in cache before being refreshed.
- **Use fixed period:** Number of seconds before the entire cache is refreshed.
- **Never refresh cache:** Cached items stay in cache until the maximum number is reached.

If an entity is marked for caching, use the following code to set the attributes. Once you create the entity, set the key values and then get the attribute values from the cache. Cached entity attributes (other than the key) do not have setters. This keeps the entity in sync with the cached version.

```
Customer cust = new Customer();
String newKey = '12345';
cust.setCustomerId(newKey);
cust.getCustomerId();
cust.getName();
cust.getAge();
cust.getAccounts();
```

13.5.21 Enhanced Entity Attribute Logging

Problem

For debugging and logging requirements, developers must write a series of log statements, for example, using logDebug or logInfo, in order to examine the current values for the Oracle RTD entities they are setting. Typically, a minimum of one line of code is required for each entity attribute whose value should be output to the Oracle RTD log. This results in a large number of lines of code simply to log output.

Solution

A predefined API in the Oracle RTD platform that allows a user to output to the Oracle RTD log the value of all session entity attributes or of a specified entity and its attributes. The output is at all the log levels: Info, Debug, Error, Warning, and Trace.

General Form of the API

Notes:

1. In this section, only the `logInfo` command is specified. All the enhanced logging options apply also to the commands `logDebug`, `logError`, `LogWarning`, and `logTrace`.
2. For simplicity, for each logging option, only the essential "active" parameter is shown. You can concatenate extra textual information into each parameter for more informative logging messages.

To log all the attributes of a session:

- `logInfo(session());`

To log all the attributes of an entity, which must be a session attribute:

- `logInfo (session().get`*`Entity_id`*`());`

The output of the API displays the Entity Name, Attribute Name, and Value in the following format:

`Entity > ExternalName: <`*`label`*`> ; InternalName: <`*`id`*`> ; Attribute <`*`n`*`> Name: <`*`attribute_name`*`>; Type: <`*`attribute_type`*`>; Value: <`*`value`*`>` or `[<`*`list_of_array_values`*`>];`

Example

An Inline Service **session** entity has two attributes:

- **customer** (Customer)
- **customer id** (Integer)

The **customer** is an entity which has three attributes:

- **assets** (String array)
- **productId** (String array)
- **unfilled** (String)

The following line:

```
logInfo(session());
```

generates output similar to the following in the Test > Log tab of the Inline Service:

```
Entity > ExternalName: ; InternalName: ApplicationSession;
Attribute 1> Name: customer; Type: Customer; Value:

Entity > ExternalName: Customer; InternalName: Customer;
Attribute 1> Name: assets; Type: SDStringArray; Value: [1, 7, 8,
9, 9, null];

Entity > ExternalName: Customer; InternalName: Customer;
Attribute 2> Name: productId; Type: SDStringArray; Value: [1, 7,
8, 9, 1];

Entity > ExternalName: Customer; InternalName: Customer;
Attribute 3> Name: unfilled; Type: String; Value: <unfilled>;

Entity > ExternalName: ; InternalName: ApplicationSession;
Attribute 2> Name: customerId; Type: int; Value: 2;
```

13.6 Performance Goals

In designing a decision process for an organization, first consider the specific metrics that the organization wants to improve by way of implementing Oracle RTD decisions. Some common performance metrics are:

- Revenue per customer visit on a Web site
- Servicing costs per customer call in a contact center

The performance metrics are configured with an optimization direction (maximize or minimize) and a normalization factor.

Performance goals have the following characteristics:

- **Performance metric:** Metrics with which the organization has chosen to measure the success of Decisions.
- **Optimization:** A value, Minimize or Maximize, that indicates the direction in which to optimize the performance metric.
- **Required:** Check if scoring for the performance metric is required. If a metric is not marked required, and a score is not available through lack of data, Oracle RTD can provide a score by examining other scores. If it is marked required, a general score will not be provided and the metric is marked not available and dropped from the scoring process.
- **Normalization factor:** The relative value to the organization of this performance metric.

This section contains the following topics:

- Section 13.6.1, "Adding a Performance Metric"
- Section 13.6.2, "Calculating a Normalization Factor"

13.6.1 Adding a Performance Metric

Click **Add** to add performance metrics. Add a metric (for example, revenue), an optimization direction (maximize), and whether the metric is required to have scores available for a decision to be made.

After you have added all of your metrics, you must decide on the normalization factor.

13.6.2 Calculating a Normalization Factor

In order to let Inline Service developers and business users express scoring functions for their Performance Goals that have different natural measuring scales, Oracle RTD requires Inline Service developers to normalize Performance Goals by way of Normalization Factors.

In the definition of Performance Goals, one Normalization Factor is applied to each Performance Goal. Subsequently, these Performance Goals are selected to be used in one or more Decisions. As a result of score normalization, business users can change the relative weights applied to the Performance Goals for their Decisions. Note that the weights themselves are expressed as percentages between 0 and 100.

In general, a higher number is better, so choose to Maximize scores with respect to each goal. For goals where lower scores are better, such as Risk or Expense, then choose Minimize.

Considerations for Normalization

In some business areas, it may be possible to have one common single metric across Performance Goals. In general, where different Performance Goals have different units of measure and magnitude, then, without some degree of normalization across Performance Goals, one Performance Goal could unexpectedly affect the outcome of scoring and the outcome of Decisions.

The Oracle RTD approach is to let business users express scores using semantics that are specific to each Performance Goal.

Normalization Example

The following example with two Performance Goals illustrates why normalizing scores is important. Assume the following:

- The Performance Goals are Revenue and Likelihood.
- The measurement units for Revenue are monetary amounts between $0 and $1000, and for Likelihoods, numbers between 0 and 100.
- The average Revenue per sale is $500.
- The business wants to balance the relative importance of presenting offers that are Revenue generating with offers that are of high interest to the customer.

  For example, business users may be ready to linearly trade 100% of likelihood for every incremental $500 in revenue. In other words, they would consider presenting an offer which is 10% less likely to be accepted than one that would bring $50 more in Revenue.

To achieve this balance without using normalization factors, the following weights would need to be applied in the Decision:

- Weight for Revenue = 1/500 = 0.002
- Weight for Likelihood = 1 – (1/500) = 0.998

Applying such weights would be quite difficult to manage as small increments in weight would drastically change the outcome of Decisions. The key factors in this design are the following:

- Revenue and Likelihood use different natural measuring scale units
- Business users want to express how those two units need to be compared at the point of Decision

Therefore, in this case, a Normalization Factor should be applied.

To achieve equal importance for Likelihood and Revenue scores, define a linear normalization factor of 500 for Revenue, and 1 for Likelihood. For Decisions, set the weights for both Revenue and Likelihood to 0.5.

General Design Factors

It is important for Inline Service developers to understand that they should perform the normalization of their Performance Goals. This normalization can be implemented in two ways:

- A linear normalization is applied when setting Normalization Factors for each Performance Goal
- For some scoring functions a linear scale factor may not be sufficient. For example, if Revenue fell between $1500 and $1650, then it might be appropriate to define a custom scoring function to use in place of the normalization factor.

The **Required** flag for a Performance Goal indicates whether an actual value is required for Decisions to proceed:

- If this flag is set to **True**, a score for this Performance Goal is required for each eligible choice.

  When no such value can be found for a Choice, that is, the score is NaN, a random score will be generated.

- If this flag is set to **False**, a score for this Performance Goal is not required for each eligible choice.

  When no such value can be found for a Choice Score, that is, the score is NaN, this Performance Goal for the Choice is ignored and the weights of the other Performance Goals participating in the Decisions are adjusted by equally distributing the weight of the Performance Goal marked as not required.

> **Note:** In the context of Externalized Performance Goals and their weighting, it is the responsibility of Inline Service developers to normalize the weights themselves.

13.7 Choice Groups and Choices

Choices are the objects evaluated during the decision process. The decision process selects the best "choices" from a list of candidates Choices. Examples of Choices are:

- List of marketing offers from which to select
- List of products to recommend
- List of resources for a task

Choices can be organized into Choices Groups. Choice Groups and Choices are organized into a hierarchical tree like representation, where one parent Choice Group can have multiple child Choice Groups. The selection of Choices from a list of candidate Choices, that is, the decision process, is a step by step operation following this logic:

1. **Eligibility:** A set of rules that determines whether or not a Choice should be considered for a given Decision. Eligibility rules can be defined at each level of the Choice Group and Choice hierarchy.
2. **Scoring:** The computation of scores along each Performance Goal defined for the Decision.
3. **Normalization:** Brings the scores along the different performance metrics to a common scale that enables the comparison of scores.
4. **Totaling:** Produces a single number for each Choice. This number is a weighted sum of the normalized scores for each Performance Goal for the Segment to which the Decision applies.
5. **Selection:** Selects a set number of "best" Choices based on Choice total score.

Choices can either be Static or Dynamic.

With Static Choices, the Choices to present to the requesting application or self-learning model are completely defined within Oracle RTD. Static Choices are useful in cases where the Choices are known in advance, and are constant over a period of time.

Dynamic Choices are Choices that are built dynamically at run time. These Choices typically reside in external data sources. This allows for the management of Choices to be done at the source system, such as Choices based on offers defined in an offer management system.

> **Note:** Choice Groups are always Static, that is, defined in an Oracle RTD Inline Service.

This section describes the general features and processes applicable to Choice Groups and to both Static and Dynamic Choices. For information specific to Dynamic Choices, see Section 17.1, "Dynamic Choices."

This section contains the following topics:

13.7.1 About Choice Groups and Choices

Choice Groups and Choices have the following characteristics:

- **Attribute values:** The attributes that make up the choice. These can be inherited from the parent Choice Group, or assigned at the Choice level.
- **Scores:** Each choice will be scored according to the definition in the Scores tab. Choices are scored against all of the performance metrics that are defined for a given Decision.
- **Choice Events:** Choice events are only described at the Group level. These events identify important events in a Choice lifecycle. For instance, a Cross Selling Offer made may have events such as Offered, Accepted, and Product First Used.
- **Rules:** Two types of Rule can be applied to Choices, both of which can be created and modified by Decision Center users:
  - Eligibility rules govern the conditions under which Choices are considered for a Decision. Eligibility rules can be defined at any level of the Choice Group hierarchy. Choices inherit all the Eligibility conditions defined at a higher level of the hierarchy.
  - Scoring rules can be used to associate numeric scores to Choices. Those scores can be used as part of the Performance Goal scoring logic or as attributes of Choices.
- **Advanced:** The Advanced button lets you choose to show the element in Decision Center, and to change the label of the element. Changing the label of the element does *not* change the Object ID.

13.7.2 About Choice Group and Choice Attributes

Choice Group attributes are used to define attributes at the group level. Group attributes apply only at the Choice Group level, and so are not assignable for individual Choices.

Choice attributes are defined at the Choice Group level to ensure that each Choice in a Group has the same set of attributes. Choice attribute values are individually defined at the Choice level. Choice attributes may have default values that can be set and overridden at lower levels.

Choice Groups and Choices are defined hierarchically. The hierarchy should follow the logical taxonomy of the Choices. At the top level, it is necessary to consider the definition of Choice attributes that make sense for a whole subtree. At lower levels, the shape of the hierarchy is typically determined by organizational considerations.

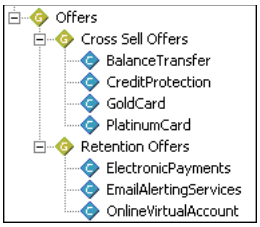
Choice attributes are typically defined at the higher levels of the hierarchy. Some attributes have a default value that may be marked as non-overrideable, which means that the value provided by default is the value that will be used. This is typically done when computations are involved. This is useful when you do not want a business user to update the attribute after deployment.

Choice attribute values can be one of the following:

- A constant
- An attribute or variable
- A function or rule call
- A model prediction

Figure 13–2 shows an example of a choice group.

Figure 13–2 Example Choice Group



The Choice attributes set at the Offers level for this example are shown in Table 13–7.

Table 13–7 Choice Attributes Set at the Offers Level for Offers Choice Group Example

Attribute	Type	Value
Message	String	No default value. Values assigned at Choice level for each Choice.
ShouldRespondPositively	Boolean	The function `ShouldRespondPositively()` that returns a Boolean value about whether a customer will respond positively to a given Choice. This specific function is used in the context of the `loadgen` simulator.
likelihood	Predictive/ Double	A placeholder attribute that is assigned by the choice model or choice event model where this choice group is registered. Used for likelihood that the choice will be accepted. There is no default value as this is calculated at run time.

Table 13–7 (Cont.) Choice Attributes Set at the Offers Level for Offers Choice Group

Attribute	Type	Value
Profit Margin	Double	An indicator of the profitability of the offer represented as a percentage. Default value of 0.5. Values assigned at Choice level for each Choice.

Each Choice overrides the Profit Margin and Message values with a value that is indicative of that Choice. However, the default value will be available at run time in case the server cannot respond in an effective time period.

No Choices override the ShouldRespondPositively attribute, as they all use the same function to determine that value. The likelihood is calculated by the model for each Choice at run time.

There is another attribute at the Group level. It is a Group Attribute called **averageLikelihood**, of type **Predictive/Double**. This attribute is used by the model as an average of likelihoods across all choices. It is used as a likelihood if a likelihood for a given Choice is not available. There is no default value, as this is calculated at run time.

13.7.3 Choice Attribute Characteristics

Choice attributes have the following characteristics:

- **Name:** The name of the attribute.
- **Category:** The category to which the attribute belongs. Categories are defined by the Category element.
- **Type:** Data type of the attribute.
- **Array:** Whether the attribute is a collection.
- **Type Restriction:** If you want to use a choice attribute or a choice group attribute in a rule, you can select a Type Restriction for the attribute. This is not a mandatory requirement, but it will help you in formulating the rule. For more information about creating and using type restrictions, see Section 13.18, "About Type Restrictions."
- **Inherited Value:** The value, if any, that the Choice Group or Choice attribute has inherited from its parent.
- **Value:** The value of the attribute. This value always overrides an inherited value.
- **Show in Decision Center:** Select this option to make the attribute visible to business users in Decision Center. Deselect for internally used attributes.
- **Use for indexing:** Select this option if you want to be able to look up the Choice by the attribute specified. For example, assume you have a choice attribute called `name`. A static method is generated on the Choice Group called `getChoiceWithName(String name)`. This method returns a choice.
- **Send to client:** Select this option if the attribute will be sent to the outside client calling the Inline Service that returns this choice.

To add or remove Choice attributes, click **Add** or **Remove**. To edit a Choice attribute, right-click the attribute and choose **Properties**. You can only edit Choice attributes at the highest level that they are defined.

13.7.4 About Choice Scoring

Choices inherit scoring functions from their parents. In scoring a Choice, you identify the performance metrics that apply to that Choice and then apply a scoring method to it. Scoring methods can be a scoring rule, function, constant, or the likelihood of an event occurring on a Choice Event Model.

For instance, assuming the Choice Group structure shown in Figure 13–2, some of the Choices may have scoring similar to the following:

- Mileage Plus Card
  - Performance Metric: Increase Revenue
  - Score: A function that uses the likelihood of the customer to accept the offer and the expected profit margin of the card to calculate the revenue potential of the offer. The likelihood is computed by a model.
- Gold Card
  - Performance Metric: Increase Revenue
  - Score: An inherited constant from the choice group level.
- Credit Analysis
  - Performance Metric: Increase Customer Retention
  - Score: A scoring rule that uses customer data to assign a score.

13.7.5 About Eligibility Rules

Eligibility rules are available at the Choice Group level and at the Choice level as follows:

- Choice Groups may have Choice Eligibility rules and Group Eligibility rules, located respectively on the **Choice Eligibility** and **Group Eligibility** tabs of the Choice Group editor.

  The Group Eligibility rules for Choice Groups are eligibility rules that apply to attributes defined at the Choice Group level.

  The Choice Eligibility rules for Choice Groups generically apply to attributes of choices defined in this Choice Group.
- Choices may have Choice Eligibility rules, located on the **Eligibility Rule** tab of the Choice editor.

These rules for Choices and Choice Groups determine their eligibility to participate in the decision. Eligibility rules determine whether the Choice is eligible to participate in the selection function or rule of the Decision or logic that makes the Choice selection.

Eligibility rules determine whether the subtree headed by a Choice Group or a Choice is eligible for a decision. Note that even if Choices themselves are eligible, they will not be eligible unless all their ancestors are eligible.

See Section 13.10, "Using Rule Editors" for information about how to use the editors for these rules.

13.7.6 Evaluating Choice Group Rules and Choice Eligibility Rules

Choice Group rules and Choice rules are inherited and additive. That is, if there are rules at the Choice Group (Group and Choice rule) and rules at the Choice level, it is as if there is a logical AND extending the rules. The inherited rules are shown in an

expandable section at the top of the rule labeled Inherited eligibility conditions. Use the Move Rule icons to expand and collapse the sections:

The following example illustrates the interaction between Choice Group rules and Choice rules:

```
Group1 has rules GroupRule1 and ChoiceRule1
Group2 is a child of Group1 and has rules GroupRule2 and ChoiceRule2
Group2 has a Choice, Choice1, and it has a rule, Rule1
```

In evaluating the rules for Choice1, the rules will be invoked in the following order:

```
GroupRule1 AND GroupRule2 AND ChoiceRule1 AND ChoiceRule2 AND Rule1
```

13.7.7 Determining Eligibility

When determining eligibility for a Choice, parent eligibility is tested first, to avoid the unnecessary evaluation of eligibility rules on Choices.

13.7.8 Choice Group APIs

The following code returns the Object label and Id, respectively:

```
public String getSDOLabel();
public String getSDOId();
```

The following code returns a Choice object from the Choice Group:

```
public Choice getChoice(String internalNameOfChoice);
```

When a Choice attribute is marked for indexing, the following method is used to return the Choice as referenced by the indexed attribute:

```
public Choice getChoiceWithAttributeID(AttributeType val);
```

13.7.9 Choice APIs

The following code returns the Object label and Id, respectively:

```
public String getSDOLabel();
public String getSDOId();
```

To get the Choice Group in which the Choice is contained, use the following code:

```
public ChoiceGroup getGroup();
```

Choice event tracking API consists of two methods defined on choices, as follows:

```
void recordEvent(String eventName);
void recordEvent(String eventName, String channel);
```

Typical code for an Integration Point recording a choice event is as follows:

```
String choiceName = request.getChoiceName();
String choiceOutcome = request.getChoiceOutcome();
ChoiceGroup.getChoice(choiceName).recordEvent(choiceOutcome);
```

Tracking of extended and accepted offers is required by many Inline Services for eligibility rules that depend on previous offers extended to, or accepted by, the same customer in the past.

Two kinds of questions, both related to a particular customer, can be answered by the Choice event history:

- How long ago was an offer extended or accepted?
- How many times was an offer extended or accepted during a given recent time period?

The answers to these questions are provided by API methods defined on Choices and Choice Groups:

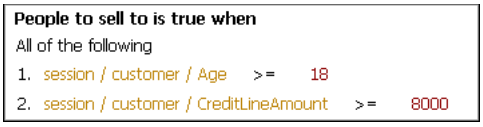
```
int daysSinceLastEvent(String eventName);
int daysSinceLastEvent(String eventName, String channel);
int numberOfEventsDuringLastNDays(String eventName, int numberOfDays);
int numberOfEventsDuringLastNDays(String eventName, int numberOfDays, String
channel);
```

13.8 Filtering Rules

As standalone rules, filtering rules can be used to segment population or be used as components of other rules. Standalone rules are reusable by many different elements.

A typical rule used to identify a segment of the population is shown in Figure 13–3.

Figure 13–3 Filtering Rule



The rule shown in Figure 13–3 targets customers over the age of 18 with a credit line amount over $8000.

See Section 13.10, "Using Rule Editors" for information on editing rules.

13.9 Scoring Rules

As opposed to eligibility rules that return Boolean values, scoring rules return numeric values. These values can be used throughout the Oracle RTD Decision logic. Typical use cases are:

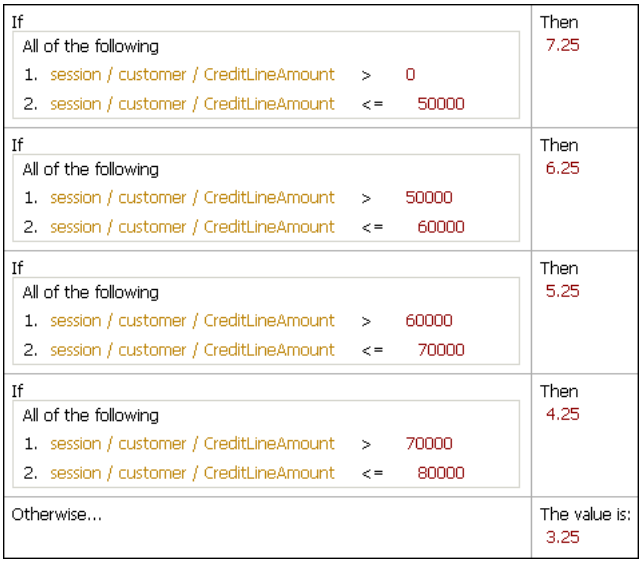
- Setting the score of a choice for a given performance goal
- Setting the value for a choice attribute

Scoring rules have a default value if none of the rule segments evaluate to true.

To add a value, click under **Then** or **The value is** in the **Value** column. Then, click the ellipsis and edit the value as you would any other rule value.

For instance, the scoring rule shown in Figure 13–4 assigns scores based on the credit line amount of a customer. If they do not fit into any of the credit line range categories, the score defaults to 3.25.

Figure 13–4 Example of Scoring Rules



Scoring rules also have the following options:

- **Description:** Scoring Rules can be adjusted by Decision Center users, so it is very important to describe your scoring rule adequately. It is suggested that you include the range that the score is to work over.
- **Advanced:** The Advanced button lets you choose to show the element in Decision Center, and to change the label of the element. Changing the label of the element does *not* change the Object ID.

See Section 13.10, "Using Rule Editors" for information on editing rules.

13.10 Using Rule Editors

Rules are used for several purposes within Decision Studio and Decision Center, namely:

- For determining the eligibility of Choice Groups and Choices to take part in a Decision
- As standalone, for creating filtering rules to be used to create population segments for decisioning
- As standalone, for creating scoring rules to be used for the scoring of choices

The Rule Editor toolbar provides access to features used to edit rules. Inside each rule editor, the editor toolbar provides users with the functionality required to create their rules. These functions become active based on the context of the rule creation and editing done by the user.

Figure 13–5 Rule Editor Toolbar

From left to right, the toolbar functions are as follows:

- Edit rule properties
- Add conditional value

- Add Rule
- Add Rule Set
- Delete
- Invert
- Move up
- Move down
- Copy
- Cut
- Paste

The editors that are used to create rules are very similar. The following sections describe how to create rules using these editors.

This section contains the following topics:

13.10.1 Oracle RTD Rule Terms and Statements

Oracle RTD has three types of rules:

- Filtering rules
- Scoring rules
- Eligibility rules

An Oracle RTD rule consists of one of more rule conditions, with logical operators governing how the conditions are combined. These conditions are expressed in the form of rule statements, as described in the section.

> **Note:** This section describes rules that have conditions, that is, they are not always true.

The following is an example of a simple filtering rule, used to illustrate rule statements in this section:

```
Select List Rule is true when
All of the following
1. session / customer / Age > 21
2. session / customer / MaritalStatus = "MARRIED"
```

Table 13–8 shows a more formal representation of the Oracle RTD rule grammar, using BNF (Backus-Naur Form)-type conventions, which describes the terms used in Oracle RTD rule statements.

Table 13–8 Oracle RTD Rule Grammar

Term	Term Component	Notes
<rtd rule>	*<header line> <logical operator> <rule entry>+* | *<header line> <array operator> <logical operator> <rule entry>+*	None.
<header line>	*<rule name>* **is true when**	Provides the name for the rule. After the rule is created, the header line is not editable.
<logical operator>	**All of the following** | **Any of the following** | **None of the following** | **Not all of the following**	The logical operator controls the *<rule entry>* immediately following. The logical operator **All of the following** is the default logical operator when you initially create a rule.
<rule entry>	*<boolean statement>* | *<rule set>*	Rule entries are always numbered. They may contain boolean statements or other rule entries.
<rule set>	*<logical operator> <rule entry>+* | *<array operator> <logical operator> <rule entry>+*	The second *<rule set>* component option is known as an *array-processing rule set*.
<boolean statement>	*<boolean>* | *<boolean function>* | *<left operand> <relational operator> <right operand>*	The *<boolean statement>* is the lowest level of a rule set - it cannot be decomposed further.
<array operator>	*<quantifier> <array variable>* **in** *<array name>* **,**	Used only for array-processing rule sets. For details, see Quantifiers and Array-Processing Rules.
<quantifier>	**For all** | **There exists**	None.

The Select List Rule example can be categorized as follows:

- **Select List Rule is true when** = *<header line>*
- **All of the following** = *<logical operator>*

The remainder of the rule consists of a *<rule entry>* made up of the following two statements:

- **session / customer / Age > 21** = *<boolean statement>*
- **session / customer / MaritalStatus = "MARRIED"** = *<boolean statement>*

Rule Sets and Boolean Statements

A *rule set* (as denoted by *<rule set>* in Table 13–8) is a composite statement, that consists of one or more numbered rule entries, each of which is either a boolean statement or another rule set.

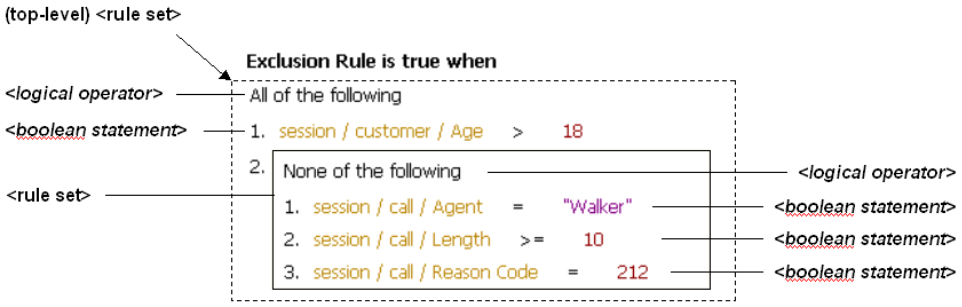
A *boolean statement* (as denoted by *<boolean statement>* in Table 13–8) contains a condition that evaluates to true or false when the rule is processed. The boolean statement is the lowest-level element of a rule set - it cannot be decomposed further.

You can optionally name the rule sets defined within higher-level rule sets.

Note: Each Oracle RTD rule has an implicit, unnamed, top-level rule set.

Each rule set is qualified by a logical operator, which controls the processing of the rule set statements. For more information, see Logical Operators.

The following Exclusion Rule example shows a rule set within a rule set.



In the example:

- The top-level unnamed rule set contains the logical operator **All of the following** and two rule entries. The first rule entry is a boolean statement, the second rule entry is a rule set.
- The rule set inside the top-level rule set contains the logical operator **None of the following** and three rule entries, each of which is a boolean statement.

Note: Oracle RTD also supports rule sets that depend on values in an array. For more details, see Quantifiers and Array-Processing Rules.

Logical Operators

The Oracle RTD logical operators are as follows:

- **All of the following** (logical and). The rule set evaluates to true when all of its boolean statements and lower-level rule sets are satisfied.
- **Any of the following** (logical or). The rule set evaluates to true when any one of its boolean statements or lower-level rule sets is true.
- **None of the following** (logical not and). The rule set evaluates to true when all of its boolean statements and lower-level rule sets are false.
- **Not all of the following** (logical not or). The rule set evaluates to true when any one of its boolean statements or lower-level rule sets is false.

Quantifiers and Array-Processing Rules

Rule sets may depend on values that occur in arrays, as follows:

- Rule sets can evaluate elements of an array
- An expression within a rule set can reference elements of an array

These types of rules are referred to as *array-processing rules*.

For these cases, there are rule sets where a "quantifier" expression - also referred to as a *quantifier* - qualifies the logical operator of the rule set. Either the statements of the rule set must be fulfilled for all array elements, or they must be fulfilled for at least one array element.

In the following example, a rule has been created which examines all of the agents contained in an array attribute, **session/agents**. In this example, the rule evaluates to true when all the agents are at least 30 years old and have a status of "Qualified."

```
Agent Rule is true when
For all people in session/agents, All of the following
1. people / Age >= 30
2. people / Status = "Qualified"
```

Full details of the syntax of array-processing rules appear in the section following. In the preceding example, the term `people` is an arbitrary term that is defined by the user and is used to identify individual array elements in the rule set.

Array-Processing Rule Qualification

The general formula for an array-processing rule qualification is:

<quantifier> <array_variable> in <array_name>, <logical_ operator>

where:

quantifier is one of the following: **For all**, **There exists**

- **For all**. This quantifier, together with the rule set logical operator, specifies that each array element must be examined, and for each array element, all the boolean statements and lower-level rule sets in the qualified rule set must be fulfilled.

  Rule evaluation will stop processing the array as soon as one array element is found where all the boolean statements and lower-level rule sets in the qualified rule set are *not* fulfilled. The rest of the elements in the array will be skipped.

- **There exists**. This quantifier, together with the rule set logical operator, specifies that all the boolean statements and lower-level rule sets in the qualified rule set must be fulfilled for at least one array element.

  Rule evaluation will stop processing the array as soon as one array element is found where all the boolean statements and lower-level rule sets in the qualified rule set are fulfilled. The rest of the elements in the array will be skipped.

array_variable is an arbitrary name to identify individual array elements of the array *array_name* in the boolean statements and lower-level rule sets in the qualified rule set

Notes:
1. The *array_variable* can still be referenced in the boolean statements of lower-level rule sets that have their own specific array variables.
2. The *array_variable* must be unique within the scope of the rule, that is, you cannot use the same *array_variable* name as the array variable for a lower-level array-processing rule.

array_name is the array to be examined

logical_operator is one of the following: **All of the following**, **Any of the following**, **None of the following**, **Not all of the following**

As an example of an array-processing rule set, consider the two entities **session** and **customer**.

The **session** entity contains the following attributes:

- The Integer attribute **AgentDept**

- The array attribute **CustInfo** of type **customer**

The **customer** entity contains the following attributes:

- The Integer attribute **CompSize**
- The String attribute **Region**

You require a filtering rule to satisfy both of the following conditions:

- The value of **AgentDept** must be 42.
- For at least one customer in the **CustInfo** array, the **CompSize** must be > 100, and the **Region** must be "West."

The filtering rule could then be defined as follows *(the array qualification expression is highlighted)*:

```
Cust Rule is true when
All of the following
1. session / AgentDept = 42
2. There exists some_customer in session / CustInfo, All of the following
   1. some_customer / CompSize > 100
   2. some_customer / Region = "West"
```

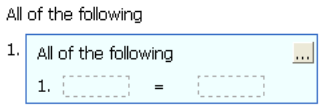
13.10.2 Adding Statements to Rules

Add Rule Set

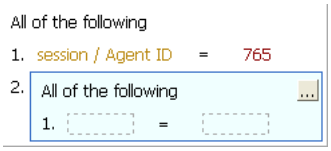
To add a rule set, click the **Add Rule Set** icon:

If this is the first element to be created in the rule, the following statements appear in the rule:

- The default logical operator **All of the following**
- A rule set entry as the numbered first line in the rule, which itself contains the default logical operator **All of the following**
- An empty two-operand boolean statement within the newly-defined rule set



Otherwise, a new rule set entry is added as the next entry in the current rule set, containing an empty two-operand boolean statement. For example, when you add a rule set to an existing rule set that already contains one boolean statement, the added rule set entry appears beside the line number 2, as in the following:

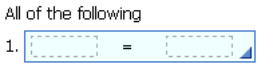


You can name the rule set, by clicking the top right corner. When a rule set is named, you can collapse or expand the rule set, by clicking the appropriate chevron icon in the top right corner of the rule set box.

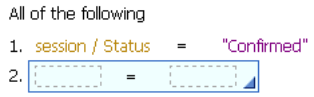
Add (Boolean Statement to) Rule

To add a boolean statement, click the **Add Rule** icon:

If this is the first element to be created in the rule, the default logical operator **All of the following** appears, followed by an empty two-operand boolean statement, as in the following:

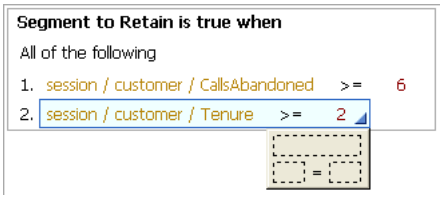


Otherwise, an empty two-operand boolean statement is added to the current rule set, as in the following example, where one boolean statement already exists:



By default, boolean statements have two operands, with an intervening operator.

To switch between single and double operands in boolean statements, click the line number of the boolean statement, then click the arrowhead icon in the lower-right corner of the boolean statement box, as in the following example:



Single operands always evaluate to a Boolean.

13.10.3 Selecting an Operator

Click the operator, then click the lower-right corner to select an operator:

Table 13–9 lists the available operators.

Table 13–9 Rule Operators

Operator	Description
none	A simple expression that has only one operand
=	Left is equal to Right
<>	Left is not equal to Right
<	Left is less than Right
<=	Left is less than or equal to Right
>	Left is greater than Right
>=	Left is greater than or equal to Right
in	Left value is contained in a List on the Right side

Table 13–9 (Cont.) Rule Operators

Operator	Description
not in	Left value is not contained in a List on the Right side
includes all of	Left list includes all the values of the Right list
excludes all of	Left list contains none of the values of the Right list
includes any of	Left list includes any one of the values of the Right list
does not include all of	Left list does not include all of the values of the list on the Right

13.10.4 Editing Boolean Statements

To edit the boolean statements of a rule, click the left side, then click the ellipsis. You can choose from a constant, attribute, or function call. Select **Array** at the top of the page to specify an array value.

- If you choose **Constant**, provide the **Data type** and a **Value** for the item. If you selected **Array**, add as many items to the array as needed. Then, for each item, choose a **Data Type** and provide a **Value**.
- If you choose **Attribute**, provide one of the following:
  - **Group attribute:** Attributes that are part of the Choice Group or its Choices that is selected in the Properties of the rule.
  - **Session attribute:** Attributes that are part of the Session entity.
  - **Application attribute:** Attributes that are a member of the Application element.
  - **Array variable:** Names used to identify individual array elements for rule sets within a quantified logical operator expression. For more information, see Quantifiers and Array-Processing Rules.

  Optionally, select **Apply filter type** and choose a **Data type** to filter the attributes by type. If you have selected **Array**, add as many items to the array as needed, then assign an attribute value for each.
- If you choose **Function call**, provide one of the following:
  - **Filtering rules:** Standalone filtering rules defined for the Inline Service.
  - **Scoring rules:** Standalone scoring rules defined for the Inline Service.
  - **Function calls:** Standalone functions defined for the Inline Service.

  Optionally, select **Apply filter type** and choose a **Data type** to filter the attributes by type. If you have selected **Array**, add as many items to the array as needed, then assign a function or rule for each.

13.10.4.1 Using Type-Restricted Objects in Rules

When you select a type-restricted object for an operand or part of an operand, you may view values from a dropdown list of values for the other operand. You may select a value from this dropdown list, but you do not have to.

For more information about type restrictions and type-restricted objects, see Section 13.18, "About Type Restrictions."

13.10.5 Editing Rule Properties

Both Filtering and Scoring rules have rule properties that can be set. To edit rule properties, click the Rule properties icon:

Edit rule properties appears.

Rule properties include call templates and negative call templates. Call templates provide a user-friendly way to describe how to call a rule from another rule.

To define a call template, add the number of parameters for the rule by clicking the **Add** button under **Parameters**. Using {0}, {1}, and so on as arguments, and phrasing to describe the rule, define the template for call. It is important to use good phrasing, as this is what will be shown when using the rule.

For instance, a rule that checks if there were at least *x* calls from the user in the last y days could be phrased as follows:

```
There were at least {0} calls in the last {1} days
```

The negative call template is used when a rule is inverted, and should express the opposite. For example:

```
There were less than {0} calls in the last {1} days
```

Rule properties also let you assign which Choice Group to use with the rule. By selecting **Use with choice group**, you can specify which Choice Group or its Choices will provide the Choice attributes for use by parameters. These attributes will be available when you edit the value of an operand.

13.10.6 Inverting Rule Elements

Use the **Invert** icon to invert different elements of a rule. By selecting the number of a boolean statement, you can invert the operator of the boolean statement. For instance, if the operand was =, it will be inverted to <>.

Logical operators for a rule set can also be inverted. To do this, select the logical operator and click **Invert**. For instance, **All of the following** becomes **Not all of the following**.

The final use for **Invert** is to invert a Boolean, or single operand, rule. When this type of rule is inverted, it is transformed to the negative call template of the function that defines the rule.

13.11 About Decisions

A Decision is used to select one or more Choices out of a group of eligible Choices. The most common use of a decision is within an Advisor. In an Advisor, two separate decisions can be called, one for regular processing, and another if control group functionality is being used in the Inline Service.

The setup of a Decision must include at least one Choice Group from which Choices are selected, and generally some numeric selection criteria so that Choices can be ordered.

Choices can be selected either at random, or with a custom selection function, or based on some user defined scoring logic configured at the choice group level for each associated Performance Goal.

Multiple scores can be defined for each choice using a variety of scoring techniques such as static scores or function driven values, rule based or predictive scores.

At run time, the Decision first identifies all the eligible choices in the subtree of its associated Choice Groups. Then all the eligible choices are scored (with one or multiple scores) and ordered.

Examples of scoring techniques for Choices include:

- Likelihood of interest in the Choice (as computed by an Oracle RTD self-learning predictive model)
- Business value of the Choice (as computed by a user defined rule or function)

Alternatively, a custom selection function can be written to select the choice.

Selection criteria include:

- **Select Choice from:** Used to assign the Choice Group or Groups that will be considered by the Decision.
- **Number of Choices to Select:** Indicates the maximum number of Choices that will be selected by the decision. The actual number of Choices returned at run time may be smaller or equal to this number, based on eligibility rules.

  This number can be overridden at the Advisor level, in the event where one decision is called by multiple touchpoints and each requires a different number of choices to be returned.

  The default and most commonly used number is 1.
- **Radio buttons for Type of Selection:** The radio button that you select controls whether you want to select a random Choice or if you want to use weighted Performance Goals as part of the Choice selection procedure:

  1. The **Select with Fixed Goal Weights** option enables you to specify population segments and set Performance Goal weights for each of the segments. The screen areas that appear when you select this option are:

     * **Target Segments:** Segment of the population that have been segmented using filtering rules. The default segment is everyone.

     * **Priorities:** Used to set Performance Goal weights for a given segment. The Performance Goals to use for scoring are selected in the Decision. Consequently, each specified goal must have a matching scoring method for each choice group selected for the Decision.

  2. The **Select with Custom Goal Weights** option allows you to have custom Performance Goal weights, typically by executing a function that returns Performance Goal weights at run time.

     If you select the **Select with Custom Goal Weights** option, and click the ellipsis button beside the Select with Custom Goal Weights box, you must select one of the following in the Value window:

     * **Function or rule call**, then a function that returns a data type of `com.sigmadynamics.sdo.GoalValues`

     * **Attribute or variable**, then an application parameter or session attribute of type `com.sigmadynamics.sdo.GoalValues`

  3. The **Select at Random** option assigns random selection of a Choice from the Choice Groups. This is often used for a Control Group Decision.

For more information on the **Select with Fixed Goal Weights** and **Select with Custom Goal Weights** options, see Section 13.11.1, "Segmenting Population and Weighting Goals."

To add a Choice Group, click **Select**, then select the Choice Group or Groups to use.

To select Performance Goals for the Decision, click **Goals**, then select the desired goals.

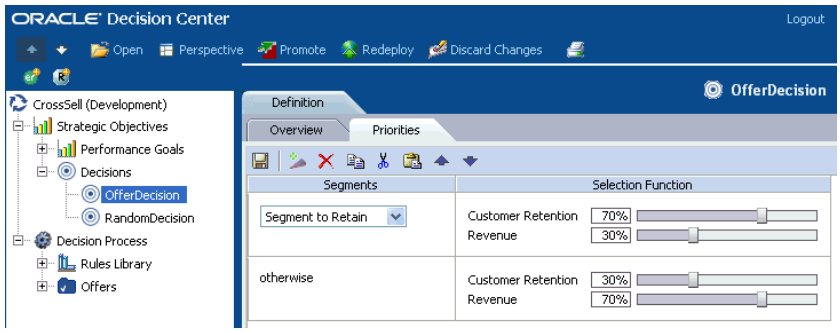
This section contains the following topics:

- Section 13.11.1, "Segmenting Population and Weighting Goals"
- Section 13.11.2, "Using a Custom Selection Function"
- Section 13.11.3, "Pre/Post-Selection Logic"
- Section 13.11.4, "Selection Function APIs for Custom Goal Weights"
- Section 13.11.5, "Adding Imported Java Classes and Changing the Decision Center Display"

13.11.1 Segmenting Population and Weighting Goals

Decisions can target segments of the population and weight the performance metrics attached to that Decision for each segment.

Note: Business users can modify Decision priorities (the weights applied to a Decision for a given segment) in Decision Center at any point throughout the lifecycle of the Decision. They can change the priorities in a Decision Center window similar to the following, which derives from the Cross Sell application released with Oracle RTD:



There are two ways that you can set up your Decision, depending on what kind of weights you want for your Performance Goals:

- Pre-defined weights, whose values are specified in the Inline Service
- Custom weights, whose values can be calculated or changed at run time

For ***pre-defined weights***, start the decision setup process by selecting **Select with Fixed Goal Weights** in the Selection Criteria for your Decision.

Then, add one or more target segments, which are defined by a rule or a function, as follows:

- To add or remove a Segment, click **Add** and **Remove**.

Finally, specify a fixed percentage weight for each Performance Goal that you select for each segment, by performing the following steps for each segment:

- Click the segment.
- Click **Goals** to select Performance Goals to use for this Decision.

  The selected Performance Goals are shown in the Priorities area.
- Specify a Weight for each Performance Goal in the Priorities area.

For instance, consider a Decision, Select Best Offer, for an Inline Service where two Performance Goals have been defined, Customer Retention and Revenue. We have also defined a segment of the population **People to retain** through filtering rules. The default remainder is the segment to which we will cross sell.

The weighting is for each Performance Goal and for each segment:

- People to retain
  - Customer Retention: 90%
  - Revenue: 10%
- Default
  - Customer Retention: 20%
  - Revenue: 80%

For ***custom weights***, start the decision setup process by selecting **Select with Custom Goal Weights** in the Selection Criteria for your Decision. The key feature is a function that you must write to retrieve or calculate Performance Goal weights at run time. This function must return a data type of `com.sigmadynamics.sdo.GoalValues`.

> **Terminology:** In this section, this function is referred to as the *custom goal weights function*.

Typically, the function would have parameters that can influence the values of the goal weights, but that is not mandatory.

Within the custom goal weights function, you do not explicitly create segments. Instead, you can include one or more conditions, each of which returns Performance Goal weights for a separate subdivision of your population.

When you set up the Decision, you can either select the custom goal weights function explicitly as the driving parameter of the process, or you can select an application parameter or entity attribute that receives the goals from the custom goal weights function or other sources.

For example, assume you have defined a condition, **IsMarried**, which effectively segments your target population. You have two Performance Goals **Max Rev** and **Min Costs**.

You create a function **Get GoalValues**, that returns a data type of `com.sigmadynamics.sdo.GoalValues`, and which calculates and returns the Performance Goals weights.

> **Note:** To illustrate the principle, the following example uses specific numbers for Performance Goals. These can be replaced by variables whose values are evaluated elsewhere in the function.

The key statements of the custom goal weights function **Get GoalValues** could be:

```
if (IsMarried)
{return new GoalValues().buildValueForMaxRev(100).buildValueForMinCosts(50);}
else
{return new GoalValues().buildValueForMaxRev(80).buildValueForMinCosts(120);}
```

With *both pre-defined and custom weights*, when the Decision is invoked, the performance metric scoring (whether function, scoring rule, function, and so on) is applied to all of the eligible Choices. Scores are leveled using the normalization factor of the performance metrics. Scores are then weighted according to performance metric weighting contained in the decision. A total score is achieved, and the Choice with the highest score is selected.

13.11.2 Using a Custom Selection Function

If, instead of using the standard scoring decision process, you would like to use a custom selection function, select the **Custom selection** option on the **Custom Selection** tab. Choose the selection function from the list, and add any parameters that the function requires.

13.11.3 Pre/Post-Selection Logic

Scriptlets in the Pre and Post Selection tab are executed before or after the scoring is done and the decision is made.

Pre-selection logic is executed after collecting all the eligible Choices, but before the selection happens. Post-selection logic is executed after the selection, but before the selected Choices are returned. Post-selection logic is more common. For example, this section can be used to record which choice Oracle RTD has recommended.

The logic here can use the variables defined for the computation of the choices. For example, the name of the Choice array, which contains eligible Choices before and selected Choices after the selection, is set in the Pre/Post Selection tab (by default, `choices`).

Decision returns a `choiceArray`. To access the individual elements, use an index into the array. The following example reads the `choiceArray`, and records the base event `Delivered` to the Choice Event Model. The method `choice.recordEvent` calls the model `recordEvent`, passing in the Choice to be recorded.

```
for (int i = 0; i < outputChoiceArray.size(); i++) {
  Choice choice = outputChoiceArray.get(i);
  choice.recordEvent("Delivered");
 }
 session().addAllToPresentedOffers(outputChoiceArray); /* Store presented offers
for future reference */
```

13.11.4 Selection Function APIs for Custom Goal Weights

The type of weights parameter is GoalValues. The GoalValues class has a getValue method for each of the goals defined in the Decision. For example, given the goals CustomerRetention and Revenue, it has the following methods:

```
public double getValueForCustomerRetention();
public double getValueForRevenue();
```

13.11.5 Adding Imported Java Classes and Changing the Decision Center Display

To add imported Java classes to your Inline Service, click **Advanced**, next to the description. You can also change the display label for Decision Center, and choose

whether the element is displayed in the Decision Center Navigator. Changing the display label does not affect the Object ID.

13.12 About Selection Functions

As an alternative to the standard scoring decision process, a *selection function* can be used by a Decision. Selection functions are completely user defined. However, selection functions have a well-defined signature. They take a Choice array as input, and return a Choice array as output.

Selection functions have a description, as well as the following parameters:

- **Input Choice Array:** The input parameter to the selection function. The data type of this variable is `SDChoiceArray`.
- **Output Choice Array:** The return variable, which specifies the name of the variable that contains the selected choices and should be returned to the caller of this selection function. The return variable can be the Input Choices Array that is passed in to this selection function, or it can be another variable defined locally within the Logic panel. The data type of this variable is `SDChoiceArray`.
- **Number of Choices Parameter:** The name of the function argument that represents the number of choices that the selection function should return. The default name of the parameter is `numChoices`. The data type of this argument is `int`.
- **Weights:** If goals are defined for the Decision that uses this selection function, those goals are passed to the Selection function under the parameter named in Weights. The type is a `GoalValue`. For more about `GoalValue`, see the section on Decisions.
- **Extra Parameters:** Any extra parameters the selection function needs.

This section contains the following topics:

- Section 13.12.1, "Selection Function Scriptlets"
- Section 13.12.2, "Adding Imported Java Classes and Changing the Decision Center Display"

13.12.1 Selection Function Scriptlets

Selection functions are used as a custom function for selection criteria. Many standard priority functions are available through templates. Priorities or selection functions are defined in Java. A set of these are predefined in the template, and usually either fill in the need, or provide an advanced prototype to modify.

Java code that does the actual selection of Choices from the list passed in as an Input Choices Array is entered in the Logic pane. Often, the Java code in the Logic section will want to refer to other classes. For the Java code and the function to compile correctly, the classes need to be imported into the function.

The `execute` method invokes the selection function.

A simple example of a selection function is shown in the following code sample:

```
double maxL = -1.0;
Choice ch = null;
for (int i = 0; i < eligibleChoices.size(); i++) {
  Cross_Selling_OfferChoice cso = (Cross_SellingOfferChoice)eligibleChoices.
get(i);
  double likelihood = cso.getLikelihood();
```

```
  if (ch == null || (!Double.isNaN(likelihood) && likelihood > maxL)) {
    maxL = likelihood;
    ch = cso;
  }
}
SDChoiceArray selectedChoices = new SDChoiceArray(1);
if (ch != null)
  selectedChoices.add(ch);
```

13.12.2 Adding Imported Java Classes and Changing the Decision Center Display

To add imported Java classes to your Inline Service, click **Advanced** next to the description. You can also change the display label for Decision Center, and choose whether the element is displayed in the Decision Center Navigator. Changing the display label does *not* affect the Object ID.

13.13 About Models

Models serve two primary purposes: prediction and reporting. Models should be defined to:

- Predict the likelihood that certain events associated with Choices will occur
- Analyze data correlated with those events

Oracle RTD models automate many of the tasks that need to be addressed when designing and integrating predictive models for the purpose of improving decisions in a business process. These automated tasks include:

- Applying data transformation to input data prior to applying predictive models
- Validating the quality of the models
- Rebuilding / recalibrating models when new target attributes occur
- Validating the accuracy of models with new data and / or new outcomes
- Introducing some degree of randomization in the decision making process to enable "exploration"

The rest of this section highlights how the Oracle RTD model features address these different situations.

> **Note:** Data accumulated in Oracle RTD predictive models can also be exported to external database tables, and subsequently analyzed using standard reporting and business intelligence products and techniques. For more information, see the chapter "Setting Up and Using Model Snapshots" in *Oracle Real-Time Decisions Installation and Administration Guide*.

Models are associated at design time with a Choice Group. This defines the list of Choices to which prediction and reports apply. The nature of the Choices to which Oracle RTD models apply is very flexible as demonstrated by the following example:

- A model can be associated with a **Marketing Offers** choice group that contains a list of offers that have been presented and purchased by customers over time. This model can then be used to:
  - Predict: Compute the likelihood that a given customer will purchase a given offer in the list

  - Analyze: Explore reports highlighting customer characteristics correlated with purchase events

- A model can be associated with a **Call Transfer** choice group that contains two choices: **Call was transferred** and **Call was not transferred**. This model can be used to:
  - Predict: Compute the likelihood that a given customer call will be transferred or not transferred
  - Analyze: Explore reports highlighting call, agent and customer characteristics correlated with transfer events

Using Oracle RTD terminology (and by contrast with other approaches), one single Oracle RTD model applies to a list of target attributes rather than to a single target attribute. An Oracle RTD model therefore defines a series of shared parameters that apply to all the Choices to which this model applies. Because Oracle RTD models are in essence defining shared metadata, Oracle RTD models can then apply to a set of Choices that evolves over time. For example:

- New Offers can be added to the **Marketing Offer** choice group and Oracle RTD will automatically apply the shared model parameters to those new occurrences

> **Note:** Changes to Oracle RTD models parameters can be made incrementally at any point in the lifecycle of a Model and will apply in a forward manner to all model operations.

For more technical information about Oracle RTD models, see:

`http://www.oracle.com/technetwork/middleware/real-time-decisions/overview/index.html`

This section contains the following topics:

- Section 13.13.1, "Model Types"
- Section 13.13.2, "Model Common Parameters"
- Section 13.13.3, "Model Attributes"
- Section 13.13.4, "Model APIs"

13.13.1 Model Types

When defining models with Oracle RTD, you can select from three types of model: Choice Event Model, Choice Model, and Model.

Each type of model corresponds to predefined usage patterns. While they share most of their characteristics, they differ by the degree of automation they provide for using prediction for the purpose of improving business decisions. All types of model can be used for prediction and reporting purposes.

- **Choice Models and Choice Event Models:** These models are associated with a target attribute of type Choice Group.

  A **Choice Event Model** is used to predict and analyze the conditional probability of events occurring, given a base event. Choice Event Models are defined with one base event and a list of positive events. The model can be used to predict the conditional probability that a positive event will happen given the base event happening. The model interprets the list of positive events as a sequence.

It is possible (and quite common) to define more than one Choice Event Model to track the same group of events in a given Choice Group. The differences between the models are typically related to time windows, base events or partitioning attributes.

A **Choice Model** is used to predict and analyze the probability of occurrences of Choices in general. Choice models do not define any events associated with their Choices. The model predicts whether the Choice is present or not.

This is useful for situations where the Choices represent items that can be present in a business process. Example include: the reason for a call, a product that is being purchased, a Web page that is visited.

- **Models:** These models are associated with a target attribute of type session attribute. Plain Models or Models are typically not used but represent general purpose predictive models that would not have inherent knowledge of abstractions such as Choices, Choice Groups and Events.

13.13.2 Model Common Parameters

Several of the model parameters are common to the three types of model:

- **Algorithm**

  The **Algorithm** dropdown list defines the type of predictive algorithm the model will use.

  There are two types of algorithm supported by Oracle RTD:

  – Bayesian algorithm

  – Regression algorithm

  Selecting an option will determine how correlations between input attributes and target attributes are handled to produce likelihood scores. Oracle recommends that you consult the mathematical literature to better understand the differences between these algorithms. In general terms, Bayesian algorithms are well suited when the proportion of positive outcomes is low, and Regression algorithms are well suited when the proportion of positive outcomes is high (typically above 15%).

- **Time Window**

  The **Default time window** check box and the **Time Window** dropdown list define the time lifecycle of models. The default time window is defined at the application level and is used for the model by default, expect when you uncheck **Default time window** and select from the **Time Window** dropdown list.

  Model time windows can be set to:

  – Week, Half-Month, Month, Two Months, Quarter, Half-Year, or Year

  Based on the defined model time window, Oracle RTD will create new model instances as time passes to ensure that old data does not have the same influence over predictions as new data. The purpose of the time-windowing strategy applied by Oracle RTD is to automatically create overlapping instances of models at each time window. Oracle RTD implements an overlapping time window approach with two models always present:

  – A primary model that continuously learns and predicts

  - A secondary model that starts half-way through the lifecycle of the primary model, and is only used for learning purposes until it becomes the primary model itself

- **Use for prediction** and **Randomize Likelihood**

  All types of model can be used for prediction and reporting purposes. Models that are to be used for prediction purposes must have the **Use for prediction** option selected. When selecting this option, you can also decide whether or not to use the **Randomize Likelihood** option.

  Models used for prediction are available to compute a mathematical likelihood for each eligible choice at runtime in the Decision Server.

  The **Randomize Likelihood** option will apply some randomization formula on those scores to avoid a local minimum or maximum situation. This frequent situation occurs when new Choices are introduced during the life cycle of a model and when Models associated with those new Choices are "competing" with models on established Choices.

  In the context of new Choices, it is sometimes useful to introduce some degree of randomization so that those new Choices get a chance to be presented, and as a result to accumulate learning. By selecting the **Randomize Likelihood** option, you let Oracle RTD decide the degree to which model scores will be randomized. The randomization factor applied by Oracle RTD when using this option is proportional to the observed error of predictions for that given model.

- **Premise Noise Reduction**

  One known data pattern that needs to be addressed when using predictive models for decision purposes is the case of self-fulfilling predictions. This situation typically occurs when an input data attribute is always associated with the model target attribute. If you expect this situation to occur, select the **Premise Noise Reduction** option.

  This option automatically identifies input variable values that are highly correlated with the model output and ignores those values for prediction purposes.

  For example:

  - Assume that a Model is built to predict products that will be added to a shopping cart
  - Assume that the array of products already in the shopping cart is used as input to the Model

  Using the **Premise Noise Reduction** option ignores the self-fulfilling correlation observed between a product and itself. Oracle RTD will also ignore products that are always associated with the input product, for example, as part of a bundle.

  Though ignored for prediction purposes, input attribute values that are identified as "noise" are still visible in Decision Center and displayed in graphs with gray bars.

- **Advanced:** The **Advanced** button lets you choose to show the element in Decision Center, and to change the label of the element. Changing the label of the element does *not* change the Object ID.

13.13.3 Model Attributes

Choice Tab for Choice Events and Choice Models

For **Choice Event and Choice Models**, the Choice Tab has the following common options:

- **Choice Group**
- **Label for Choice** - the label for the Choice Group shown in Decision Center

For **Choice Event Models**, the Choice Tab has the following specific options:

- **Base Event**: The event is the event associated with Choices that represent the baseline of the conditional probabilities computed by the Choice Event Model. Events can be selected from a list of Events defined at the Choice Group level.
- **Base Event Label**
- **Positive Outcome Events**: This is the list of events for which the Model will compute conditional probabilities.

  For example, assume a Model linked to a Product Choice Group has a baseline event **Delivered** and two positive events, **Interested** and **Purchased**. When computing the likelihood of purchase of a given product:

  – The Model will first compute the likelihood of the **Purchased** event based upon knowing the **Delivered** event, and use this number if available.

  – If not enough data is available to make such prediction (in other words, the Model returns NA), the Model likelihood function will then compute the likelihood of the **Interested** event based upon knowing the **Delivered** event, and return this number if available.

  – If not enough data is available to make such a prediction, the Model likelihood function will return NA.

  In that regard, the list of positive outcomes represents an ordered sequence of positive events that you attempt to predict. The Model likelihood function allows you to select the "most positive" event from an ordered list of events.

For **Choice Models**, the Choice Tab has the following specific option

- **Mutually exclusive choices**: This option is available for Choice Models and indicates that the Choices representing the target of the Model are mutually exclusive.

Attributes Tab for Choice Events and Choice Models

For **Choice Event and Choice Models**, the Attributes tab has the following common options:

- **Partitioning Attributes**

  Partitioning a Model by adding a **Partitioning Attribute** is equivalent to creating a separate Model for each value of the Partitioning Attributes.

  For example, when building a Model to predict likelihood of acceptance for offers, you might want to create a Partitioning Attribute indicating the interaction channel in which this offer was delivered and accepted. Having a Model for Web offer acceptance and Contact Center offer acceptance will result in better predictive Models as it is known that customers with the same characteristics have very different likelihoods of accepting an offer depending on the channel through which the offer is made.

The decision to partition a model need to be made very carefully for the following reasons:

- Creating attributes will reduce the number of observations for each choice and attribute value combination, and therefore slow down the Model conversation process.
- Model size is very sensitive to the cardinality of partitioning attributes.
- Excluded attributes. By default, all session and entity attributes are used as inputs to Models. To remove a specific attribute from the list of Model inputs, add them to the list of **Excluded Attributes**.

For **Choice Models**, the Attributes tab has the following specific option:

- **Aggregate by:** By choosing one of the partitioning attributes, you can create additional sub-models for learning on the attribute. As with partitioning, a decision to aggregate on an attribute has to be based on a radical difference in predicted likelihood depending on the value of the attribute in question.

Learn Location Tab for Choice Events and Choice Models

For **Choice Event and Choice Models**, the Learn Location tab has the following common options:

- (Learn) **On session close** or **On Integration Point**

  By default, all model learning operations occur when the session closes. Alternatively you can select a specific integration point (Informant or Advisor) at which point model learning operations need to be triggered.

Temporary Data Storage Tab for Choice Events and Choice Models

For **Choice Event and Choice Models**, the Temporary Data Storage tab has the following common options:

- Use temporary data storage

  A common data pattern is that events related to Choices might not all occur during the lifetime of the Oracle RTD session. To make sure the original input values use for predictions are also used to update the Model when such delayed positive outcomes occur, select the **Use temporary data storage** option.

- **Days to Keep:** Specifies the number of days to keep temporary data stored on the server.
- **Keys:** The data stored in temporary data storage is available for retrieval by having any one of the keys defined in the Temporary Data Storage tab.

13.13.4 Model APIs

The following code returns the Object label and Id, respectively:

```
public String getSDOLabel();
public String getSDOId();
```

This section contains the following topics:

- Section 13.13.4.1, "Querying the Model"
- Section 13.13.4.2, "Recording the Choice with the Model"
- Section 13.13.4.3, "Obtaining Model Object by String Name"
- Section 13.13.4.4, "Recording Choice Events for Choice Event Models"

- Section 13.13.4.5, "Recording Choices for Choice Models"
- Section 13.13.4.6, "Obtaining Model Choice Likelihood"

13.13.4.1 Querying the Model

The model can be queried using any of the getChoiceEventLikelihood methods shown in the following code sample. This will return the likelihood of a Choice being chosen by the model.

```
public static double getChoiceEventLikelihoods(GENOffersChoice choice, String
eventName);
public static double getChoiceEventLikelihoods(GENOffers choiceGroup, String
eventName);
```

13.13.4.2 Recording the Choice with the Model

For the Choice Event model, the model method `recordEvent` is executed when a call to the Choice method `recordEvent` is made. Therefore, it is not necessary to directly invoke this method on the model. This method is usually called from within the Integration Point where the Choice was extended to the calling application.

For instance, in an Advisor Integration Point:

```
if (choices.size() > 0) {
  Choice ch = choices.get(0);
  ch.recordEvent("Presented");
  session().setOfferExtended(ch.getSDOId());
}
```

For the Choice model, the following APIs are available:

```
public static SDStringArray getChoice()
public static void setChoice(SDStringArray _v)
public static void addToChoice(String _a)
public static void addAllToChoice(SDStringArray _c)
```

The Informant usually records a Choice with the model. For instance, in a case where we are recording the Choice of a call reason code with the Model Reason Analysis:

```
if (code == 17)
  ReasonAnalysis.addToChoice("BalanceInquiry");
else if (code == 18)
  ReasonAnalysis.addToChoice("MakePayment");
else if (code == 19)
  ReasonAnalysis.addToChoice("RateInquiry");
else
  ReasonAnalysis.addToChoice("Other");
```

If the Choices were not marked mutually exclusive, this call must include a call to `getModelData()` before recording the Choice:

```
if (code == 17)
  ReasonAnalysis.getModelData().addToChoice("BalanceInquiry");
else if (code == 18)
  ReasonAnalysis.getModelData().addToChoice("MakePayment");
else if (code == 19)
  ReasonAnalysis.getModelData().addToChoice("RateInquiry");
else
  ReasonAnalysis.getModelData().addToChoice("Other");
```

If you are working with a Choice Array, you should send an empty string to the model first:

```
ReasonAnalysis.getModelData().addToChoice("");
```

13.13.4.3 Obtaining Model Object by String Name

These APIs allow users to obtain a model object by its string name. There is one API for choice models, and another for choice event models.

General Form of the APIs

```
ChoiceModel <cm_model_instance_name> =
Application.getApp().getChoiceModel(<model_name>);
```

```
ChoiceEventModel <cem_model_instance_name> =
Application.getApp().getChoiceEventModel(<model_name>);
```

where:

<model_name> must be a type String model name

Example

For an example of the **getChoiceEventModel** API, see Example of Choice Event Model APIs.

For an example of the **getChoiceModel** API, see Example of Choice Model APIs.

13.13.4.4 Recording Choice Events for Choice Event Models

This API allows users to send in a choice id and event and automatically update the associated choice event model. Developers can write "record event" logic once. The code can then be reused, without additional editing, for additional choices and choice groups added subsequently to the Inline Service.

General Form of the API

```
<cem_model_instance_name>.recordEvent(<choiceId>,<eventId>);
```

where:

<cem_model_instance_name> must be of type ChoiceEventModel, previously retrieved by the **getChoiceEventModel** API

<choiceId> must be a type String choice identifier

<eventId> must be a type String event identifier

Example of Choice Event Model APIs

The following shows an example of the **getChoiceEventModel** and **recordEvent** APIs:

```
// Select the choice for the given classification
int classification = session().getDnaRecord().getClassification();

String modelName = "DnaClassesAPITest";
ChoiceEventModel model = Application.getApp().getChoiceEventModel(modelName);
// Definitions
SDChoiceArray choices = new SDChoiceArray();
ChoiceGroupInterface group = ClassificationsTestApi.getPrototype().cloneGroup();
// Get the group of choices to simulate
//group = ClassificationsTestApi.getPrototype().cloneGroup();
// Get all eligible choices
group.getEligibleChoices(choices);
```

```
int sz = choices.size();
// Iterate through all eligible choices in the group
for (int i = 0; i < sz; i++) {
ClassificationsTestApiChoice ch = (ClassificationsTestApiChoice) choices.get(i);
    String choiceId = ch.getSDOId();
    String eventId = "record";
    model.recordEvent(choiceId, eventId);

if (ch.getClassification() == classification) {
String cnm = ch.getSDOId();
eventId = "identified";
model.recordEvent(choiceId, eventId);
logInfo("ID " + session().getDnaRecord().getId()
+ " Classification: " + classification + " choice: "
+ ch.getSDOId() + " size was: " + sz);
}
}
```

13.13.4.5 Recording Choices for Choice Models

This API allows users to send in a choice id to automatically update the associated choice model.

General Form of the API

<cm_model_instance_name>.recordChoice(*<choiceId>*);

where:

<cm_model_instance_name> must be of type ChoiceModel, previously retrieved by the **getChoiceModel** API

<choiceId> must be a type String choice identifier

General Form of the API

<cm_model_instance_name>.recordChoice(*<choiceId>*);

where:

<cm_model_instance_name> must be of type ChoiceModel, previously retrieved by the **getChoiceModel** API

<choiceId> must be a type String choice identifier

Example of Choice Model APIs

The following shows an example of the **getChoiceModel** and **recordChoice** APIs:

```
double amount = targetNumber;
double spacing = Application.getApp().getSpacing();
double max = Application.getApp().getMaxOutput();
double min = Application.getApp().getMinOutput();
//Application.logInfo("amount is: "+amount);
int nDigits = (int)Math.log10(max);
nDigits = nDigits + 3;
//Application.logInfo("nDigits in Dynamic Choices: "+nDigits);
String targetChoiceGroupName = Application.getApp().getTargetChoiceGroupName();
String targetChoiceBaseName = Application.getApp().getTargetChoiceBaseName();
double t_max = max-spacing;
double t_min = min-spacing;
ChoiceModel model = Application.getApp().getChoiceModel(modelName);
for (double t= t_max; t>t_min; t = t-spacing) {
    String id;
```

```
   if (amount > t) {
   id = String.format(targetChoiceGroupName+"$"+targetChoiceBaseName+"%0
"+nDigits+".1fAbove", t);
        model.recordChoice(id);
       for (double it = t-spacing; it>t_min; it= it-spacing) {
        id = String.format(targetChoiceGroupName+"$"+targetChoiceBaseName+"%0
"+nDigits+".1fAbove", it);
             model.recordChoice(id);
            }
             break;
   }
}
```

13.13.4.6 Obtaining Model Choice Likelihood

These APIs allow users to obtain a likelihood score for either a choice or choice event model for the "All" version of the model or a specific partition of the model should a partition exist.

General Form of the APIs

<cm_model_instance_name>.getChoiceLikelihood(*<choiceId>*);

<cem_model_instance_name>.getChoiceEventLikelihood(*<choiceId>*,*<eventId>*);

where:

<cm_model_instance_name> must be of type ChoiceModel, previously retrieved by the **getChoiceModel** API

<cem_model_instance_name> must be of type ChoiceEventModel, previously retrieved by the **getChoiceEventModel** API

<choiceId> must be a type String choice identifier

<eventId> must be a type String event identifier

Example

The following shows an example of the **getChoiceEventLikelihood** API:

```
ChoiceEventModel model = Application.getApp().getChoiceEventModel(modelName);
return model.getChoiceEventLikelihood(choiceId, eventId);
```

13.14 About Integration Points

There are two types of Integration Point in Oracle RTD, Informants and Advisors. These represent the integration points between external systems and Oracle RTD.

An *external system* and an *order number* are also defined for each Integration Point. These are used to generate the process map presented in Decision Center. The system determines the swim-lane and the order the position, from left to right. The order can be any number, not just integers, allowing for the introduction of new Integration Points without modifying existing ones.

This section contains the following topics:

- Section 13.14.1, "About Informants"
- Section 13.14.2, "Adding Imported Java Classes and Changing the Decision Center Display"
- Section 13.14.3, "Informant APIs"

13.14.1 About Informants

Informants represent asynchronous integration points between external systems and Oracle RTD.

Informants are typically triggered to pass contextual information to Oracle RTD to support the decision process, as follows:

- Informants pass closed loop information to Oracle RTD to update its predictive models
- Informants pass session identifiers for Oracle RTD to fetch relevant information from external data sources

To add an Informant to the Inline Service, perform the following:

1. Create an external system to identify which system accesses the Integration Point.
2. Create a Choice Group to represent the targets for your analysis. For instance, a Choice Group may represent the reasons for calls to the service center.
3. Create an Informant that receives the session key information and gathers and processes data based on the session.
4. Create an analytical model that is the repository for the data and analyzes it.

Informants have a Description, as well as the following Request characteristics:

- **Session Keys:** One or more session keys used to uniquely identify a session. Any of the session keys within the message are sufficient for identifying a session, and therefore cause the message to be dispatched to an existing session, if any, already containing information related to this message.
- **External System:** Identifies the external system that will be sending the Informant a request. Associating the Informant with an external system allows the Informant to be displayed among other Informants and Advisors in Decision Center's process map.
- **Order:** This number identifies the position of the Informant in the sequence of Integration Points displayed in Decision Center's process map. An Integration Point with an order less than another Integration Point's order will be displayed before the other Integration Point. The order can be a decimal number; for example, 2.1 will be displayed before 2.2.

- **Force Session Close:** When selected, causes the Inline Service to automatically terminate the session of the Informant after all of the Informant's asynchronous logic has executed. The same effect can be achieved by placing the following Java statement anywhere in a subtab of the Informant's Logic tab:
  `session().close();`

This section contains the following topics:

13.14.1.1 Adding a Session Key

On the **Request** tab, click **Select** to select a session key for the Integration Point. This is one of the values that the operational system will supply to the Integration Point.

13.14.1.2 Identifying the External System and Order

On the **Request** tab, use the dropdown list to choose the external system that accesses the Integration Point. This menu is populated by creating external system identifiers using the external system element.

The order in which the Integration Points are accessed is represented by **Order**. This number and the **External System** determine how the end-to-end process is displayed in Decision Center.

13.14.1.3 Adding Request Data

On the **Request** tab, click **Add** to add request data. Assignments are the values that the operational system will supply to the Integration Point. Assignments have the following characteristics:

- **Incoming Parameter:** The name of the field in the request sent to the Informant whose value will be copied from the request to the session attribute. This name does not have to be the same as the session attribute; however, it generally is named the same.

  After the session key is created, incoming parameters are assigned to the session key attributes.

- **Type:** This is the data type of the session attribute into which the incoming argument will be copied. The valid types are: integer, string, date, or double.

  Note: If the type of the request field and the session key attribute do not match, you should use a transform method.

- **Array:** Marked if the type is a collection.

- **Session Attribute:** The attribute of the session to which the incoming parameter of a request will be mapped.

13.14.2 Adding Imported Java Classes and Changing the Decision Center Display

To add imported Java classes to your Inline Service, click **Advanced** next to the description. You can also change the display label for Decision Center, and choose whether the element is displayed in the Decision Center Navigator. Changing the display label does *not* affect the Object ID.

13.14.3 Informant APIs

The following code returns the Object label and Id, respectively:

```
public String getSDOLabel();
public String getSDOId();
```

13.14.4 Informant Logic

There are two tabs for Informant logic: the **Logic** tab and the **Asynchronous Logic** tab. You can access request data for an Informant in either of the tabs.

This section contains the following topics:

- Section 13.14.4.1, "Logic"
- Section 13.14.4.2, "Asynchronous Logic"
- Section 13.14.4.3, "Accessing Request Data From the Informant"

13.14.4.1 Logic

This script runs after any Request Data declared in Request Data are executed. If the primary purpose of the Informant is to transfer data from the operational system request fields to the session key and Request Data, logic may be unnecessary, as this happens automatically according to declarations in the Request Data tab.

Logic in Informants is typically used for tracing message reception in the log file, or for pre-populating entities whose keys are supplied by the Informant's message, in order to avoid having to do this later in an Advisor, where response time might be more important. Logic is executed directly following the Request Data.

Logic in the Informant can also be used to record Choices with a Choice Model. See the Choice Model APIs for methods to call.

13.14.4.2 Asynchronous Logic

This script runs after the script defined in the **Logic** tab, described in the previous section. Any additional processing that needs to be done can be placed in this area. The order of execution of Asynchronous Logic is not guaranteed.

13.14.4.3 Accessing Request Data From the Informant

Request data from an Informant is accessed one of several ways. If the incoming parameter is mapped to a session attribute, there is a `get` method for the parameter.

```
request.get$()
```

where `$` is the parameter name with the first letter capitalized.

If the attribute is not mapped, there are methods to achieve the same results using the field name of the parameter.

```
String request.getStringValue(fieldName)
SDStringArray request.getStringArrayValue(fieldName)
boolean request.isArgPresent(fieldName)
```

13.14.5 About Advisors

Advisors represent synchronous integration points between external systems and Oracle RTD. Advisors are typically triggered to initiate an Oracle RTD Decision.

Each Advisor can make use of two Decisions, the Optimized Decision and the Control Group Decision. This enables users to compare the respective performance of the two options.

Generally, the two Decisions are set up as follows:

- The Optimized Decision implements some decision logic that takes advantage of the advanced features of the Oracle RTD Decision Framework.
- The Control Group Decision is as close to the existing business process as possible, so that the Optimized Decision has a basis for comparison.

Default Choices can be defined for the Advisor. These Choices are used when the computation in the server can not be completed in time, or if the client loses communication with the server.

13.14.6 About the Advisor Decisioning Process

Advisors have a Description, as well as the following Request characteristics:

- **Session Keys:** One or more session keys used to uniquely identify a session. Any of the session keys within the message are sufficient for identifying a session, and hence cause the message to be dispatched to an existing session, if any, already containing information related to this message.

  When the Advisor is called, the session key creation is the first thing executed.
- **External System:** Identifies the external system that will be triggering the Advisor request. Associating the Advisor with an external system allows the Advisor to be displayed among other Informants and Advisors in Decision Center's process map.
- **Order:** This number identifies the position of the Advisor in the sequence of Integration Points displayed in Decision Center's process map. An Integration Point with an order less than another Integration Point's order will be displayed before the other Integration Point. The order can be a decimal number; for example, 2.1 will be displayed before 2.2.
- **Force Session Close:** When selected, this option causes the Inline Service to automatically terminate the Advisor's session after all of the Advisor's asynchronous logic has executed. The same effect can be achieved by placing the following Java statement anywhere in any subtab of the Advisor's Logic tab: `session().close();`

13.14.7 Adding Imported Java Classes and Changing the Decision Center Display

To add imported Java classes to your Inline Service, click **Advanced** next to the description. You can also change the display label for Decision Center, and choose whether the element is displayed in the Decision Center Navigator. Changing the display label does *not* affect the Object ID.

13.14.8 Adding a Session Key

On the **Request** tab, click **Select** to select a session key for the Integration Point. This is one of the values that the operational system will supply to the Integration Point.

13.14.9 Identifying the External System and Order

On the **Request** tab, use the dropdown list to choose the external system that accesses the Integration Point. This list is populated by creating external system identifiers using the external system element.

The order in which the Integration Points are accessed is represented by **Order**. This number and the **External System** determine how the end-to-end process is displayed in Decision Center.

13.14.10 Adding Request Data

On the **Request** tab, click **Add** to add request data. Request data is the values that the operational system will supply to the Integration Point. Request data has the following characteristics:

- **Incoming Parameter:** The name of the field in the request sent to the Advisor whose value will be copied from the request to the session attribute. This name does not have to be the same as the Session attribute; however it generally is named the same.

  After the session key is created, the assignment of incoming parameters to session attributes is made.

- **Type:** This is the data type of the session attribute into which the incoming argument will be copied. The valid types are: integer, string, date or double.

  Note: If the type of the request field and the session attribute do not match, you should use a transform method.

- **Array:** Select this option if the type is a collection.
- **Session Attribute:** The attribute of the session to which the incoming parameter of a request is mapped.

13.14.11 Adding Response Data

On the **Response** tab, click **Add** to add response data. Response data is the values that the operational system will send back to the Integration Point after a request is invoked. Response data has the following characteristics:

- **Response:** The response contains an array of selected Choice objects, with each Choice containing a collection of named attribute values. The Choice selection process is governed by one of two Decision objects referenced by the Advisor. One Decision is given to the calling application.
- **Decision to Use:** The name of the Decision object to use for normal sessions, as opposed to control-group sessions. This Decision becomes the Advisor's response to the calling system.
- **Control Group Decision to Use:** Control Group Decision is used for only a small percentage of sessions as a way to assess the effectiveness of the other Decisions by providing a baseline. The percentage of sessions that use the control-group decision is specified in the Application element of the Inline Service. The Control Group Decision should be designed to select choices using "business-as-usual" logic, meaning whatever rules the enterprise previously used before introducing the Inline Service. Reports are available through the Decision Center console that

compare the business effectiveness of the Advisor's normal Decision object with its Control Group Decision.

- **Parameters:** Input parameter defined by the decision. The Name and Type columns are descriptive only, surfaced here from the Decision object.
- **Default number of choices returned:** Default number of choices returned by the decision. This is the number of choices defined by the Decision.
- **Override default with:** The Advisor can override or accept the number specified by the referenced Decision. This area specifies the maximum number of qualified choices to be included in the Advisor's response.
- **Default Choices:** A list of Choices that are returned to the calling client application whenever it tries to invoke this Advisor, and the Advisor is not able to deliver its response within the server's guaranteed response time.

  Note that default Choices do not have to be specified for each Advisor. The Inline Service may also declare default Choices, which are used for Advisors that don't declare their own. Also note that the default Choice configuration is propagated to the client application and stored in the local file system by the Smart Client component. Hence, it is subsequently available to client applications that cannot connect to the server.

13.14.12 Logic in Advisors

There are two tabs for Advisor logic: the **Logic** tab and the **Asynchronous Logic** tab.

This section contains the following topics:

- Section 13.14.12.1, "Logic"
- Section 13.14.12.2, "Asynchronous Logic"

13.14.12.1 Logic

This script runs after any request data declared in the request data tab is executed, and before the response is sent back to the client.

Advisor logic is generally not needed. You may want to use it for preprocessing data coming in with the request, or for debugging purposes.

13.14.12.2 Asynchronous Logic

This script runs after the response has been handed off to the server-side mechanism that sends it back to the client. Depending on the type of endpoint used by the client, the client may be able to start processing the result before this script finishes, thus improving the effective response time by increasing parallelism.

13.14.13 Accessing Request Data from the Advisor

Request data from an Advisor is accessed in one of several ways. If the incoming parameter is mapped to a session attribute, there is a `get` method for the parameter.

```
request.get$()
```

where `$` is the parameter name with the first letter capitalized.

If the attribute is not mapped, there are several methods to achieve the same results.

```
String request.getStringValue(fieldName)
SDStringArray request.getStringArrayValue(fieldName)
boolean request.isArgPresent(fieldName)
```

13.15 About External Systems

External systems are only identified within Decision Studio. The external system represents the operational systems within the enterprise that integrate to the Inline Service. The external system is not accessible through the API. The external system is used by an Integration Point to identify which external system will access that Integration Point. External systems are used for display on the Integration Map in Decision Center.

External systems have a Description, and a Display Label. Changing the Display Label does *not* affect the Object ID.

13.16 About the Categories Object

Categories are used to organize Choices. All Choices of the same category appear together in Decision Center. No classes are generated for categories. They are only used by Decision Center for grouping and organizing Choices.

Categories have the following characteristics:

- **Name:** Name of the category, as entered in Decision Studio.
- **Description:** Description of the category, as entered in Decision Studio.
- **Display Label:** Lets you change the Display Label. This does *not* affect the Object ID.

13.17 About Functions

Functions can be used for calculation or for other processing that you want to make reusable. Functions are defined using Decision Studio. Functions are defined with the following characteristics:

- **Description:** Description of the function.
- **Return value:** Specifies whether the function returns a value.
- **Data Type:** Type of the returned value.
- **Array:** Select this option if the return type is an array.
- **Call Template:** The definition of how the function will be called. Using `{0}`, `{1}`, and so on as arguments, and phrasing to describe the function, define the template for call. It is important to use good phrasing, as this is what will be shown when using the function. For instance, a call template for multiply is `{0} multiplied by {1}`.
- **Parameters:** Named parameters that will be used in the logic of the function. This number must match the number of arguments in the call template. For instance, Multiply has the following parameters: `a, type Double; b, type Double`

  If you want to use a function parameter in a rule, you can select a Type Restriction for the parameter. This is not a mandatory requirement, but it will help you in formulating the rule. For more information about creating and using type restrictions, see Section 13.18, "About Type Restrictions."
- **Logic:** Java code for the function. The code for multiply is:

  ```
  return a * b;
  ```

This section includes the following topics:

13.17.1 Functions to Use with Choice Event History Table

As part of configuring an Inline Service, predefined functions are available to access any data that is stored in Oracle RTD's Choice Event History table. These functions are accessible when creating or editing choice eligibility rules, and can also be called through custom Java logic as well.

To access these functions in a choice eligibility rule statement, select **Function Call**, and expand the Functions folder. The following functions appear in the list of available functions:

Function Name	Parameters	Description
Days Since Last Event	Event Name	For a given choice and a specified event name, returns the number of days since that last recorded event.
Days Since Last Event on Channel	Event Name Channel	For a given choice, specified event name, and specified channel name, returns the number of days since that last recorded event.
Number of Recent Events	Event Name Number Of Days	For a given choice, specified event name, and specified number of days, returns the total count of times that event has been recorded.
Number of Recent Events on Channel	Event Name Channel Number Of Days	For a given choice, specified event name, specified channel, and specified number of days, returns the total count of times that event has been recorded.

13.17.2 About Maintenance Operations

Maintenance Operations are special functions, that enable administrators to perform specific Inline Service related tasks in JConsole, such as the following:

- Clearing an entity cache across a cluster
- Clearing the external rule cache
- Broadcasting messages within a cluster
- Performing blocking operations

The Maintenance Operations written for an Inline Service are exposed by Oracle RTD as JMX Operations.

A Maintenance Operation may be invoked on an individual cluster member, or across a cluster as a whole. Maintenance Operations are non-blocking.

> **Note:** You may invoke a Maintenance Operation on an individual cluster member directly, in a blocking fashion. If you broadcast the request to invoke the Maintenance Operation across a cluster, you will block only as long as it takes to deliver the message.

Each Maintenance Operation will appear twice in the MBeans tree for each Loadable Inline Service:

1. To allow for the Maintenance Operation invocation on a local server only.
2. To allow for the Maintenance Operation invocation across every node of a cluster.

Different versions of a particular Inline Service may have different Maintenance Operations. For example, it is possible to have more than one version of an Inline Service deployed - they may be in different deployment states. Only Inline Services in the "Loadable" state expose any Maintenance Operations.

There are certain design-time considerations for Maintenance Operations:

- Maintenance Operations may have 0 or more primitive type arguments (String, int, double, date, or boolean), and may return String, int, double, date, boolean, or void.

  Returned values (and Exception) are logged.

- Maintenance Operation operate within the global Inline Service context. Session attributes are null. If the session is accessed, an IllegalStateException is thrown.

Maintenance Operations for External Rule Caching

Oracle RTD provides specific maintenance operations for external rules caching. For details, see Section 17.2.3.2, "External Rule Caching."

13.17.3 Adding Imported Java Classes and Changing the Decision Center Display

To add imported Java classes to your Inline Service, click **Advanced** next to the description. You can also change the display label for Decision Center, and choose whether the element is displayed in the Decision Center Navigator. Changing the display label does *not* affect the Object ID.

Functions are called from other elements using the call template. For instance, if you wanted to use the `multiply` function described in the previous section, you would choose the function from the **Edit Value** dialog. The call template `{0} multiplied by {1}` provides the editor with the position and number of arguments.

13.18 About Type Restrictions

Type restrictions define constraints that can be attached to session and entity attributes, choice and choice group attributes, and function and application parameters, so long as their type is Integer, Double, Date, or String.

Type restrictions are typically used in the Rule Editor to simplify user inputs, When you define a rule in the Rule Editor that uses one or more type-restricted elements, you can view and select from lists of values that obey the constraints defined in the associated type restrictions. A type restriction acts as an aid to the Inline Service *designer*. It does not get evaluated at run time.

There are three kinds of type restriction:

- List of Values
- List of Entities
- Other Restrictions

List of Values type restrictions appear in a list with one or more values.

List of Entities type restrictions consist of entity attribute values that are returned from a function. List of Entities type restrictions can be used to generate dynamic lists of values from database tables and views.

Other Restrictions consist of either a range of values (for Date, Double, or Integer data types), or a JavaScript Regular Expression pattern.

JavaScript Regular Expression patterns are generally used for data such as post codes and telephone numbers, where each character of the data may be subject to a different real-world constraint, depending on its position in the code. For example, US telephone numbers, such as (650) 506 7000, and British postal codes, such as KT2 6JX.

13.18.1 Managing Type Restrictions

You create and edit type restrictions as individual Service Metadata objects. You can then associate type restrictions with the following Inline Service objects in the corresponding object editor:

- Session and entity attributes
- Choice group attributes
- Choice attributes
- Function parameters
- Application parameters

If you include a type-restricted object as an operand in a rule in the Rule Editor, you can get the following assistance as you create or edit the rule:

- For objects with List of Values or List of Entities type restrictions, you can view and select values for the object from dropdown lists that obey the type restriction constraints
- For objects whose type restrictions are Other Restrictions, you can move the mouse (cursor) over the object to see the range constraint or the JavaScript Regular Expression of the type restriction

This rest of this section consists of the following topics:

13.18.1.1 Creating and Editing "List of Values" Type Restrictions

For a List of Values type restriction, you must select a data type, then provide one or more values for individual elements.

You can define List of Values type restrictions for String, Integer, Double, or Date data.

13.18.1.2 Creating and Editing "List of Entities" Type Restrictions

In order to define a List of Entities type restriction, you must first fulfill certain prerequisites:

- You must have defined a function that returns data of a certain entity type
- The referenced entity type must contain at least two columns, that will serve as the "label" and "value" of the type restriction.

The "label" and "value" columns are used in the Rule Editor when you edit a rule statement, and you select an attribute or parameter to which you have attached a type restriction.

The "label" column holds the dropdown list data. When you select from the dropdown list, the corresponding data in the "value" column is placed in the rule.

For example, a type restriction on US state codes may enable a rule creator to select a state name from a dropdown list of US state names - such as Alabama, or California, and so on. The "state name" column is the "label" column.

After the rule creator selects a state name from the dropdown list, Oracle RTD places the corresponding state code - such as AL, or CA - into the rule. The "state code" column is the "value" column.

List of Entities type restrictions are defined with the following characteristics:

- **Entity:** The entity for which you want to define a type restriction.
- **Function:** The function that returns restricted data. The Return value for this function must specify the type restriction entity as its Data Type.
- **Label:** The attribute of the type restriction entity that holds the dropdown list values when rules are created that reference objects with this type restriction.
- **Value:** The attribute of the type restriction entity that holds the values returned into rules when users select dropdown Label list values in the Rule Editor.

You can define List of Entities type restrictions for String, Integer, Double, or Date data.

Note: After you have created or edited a List of Entities type restriction, you must deploy the Inline Service. This is to enable correct values to appear in dropdown lists when you create rules involving the type-restricted attribute or parameter.

13.18.1.3 Creating and Editing Other Restrictions

With Other Restrictions, you can define range constraints or JavaScript Regular Expression patterns or a combination of both.

You can define range constraints on data whose type is Integer, Double, or Date. You can specify lower and upper limits, and whether the limit value is included or not. For example, a "greater than" condition does not include the minimum value, whereas a "greater than or equal" condition does.

A JavaScript Regular Expression pattern uses standard Regular Expression pattern characters, such as {,}, [,], $,?,\, and so on, to define a data format or pattern. You can specify JavaScript Regular Expression patterns for String, Integer, Double, or Date data.

As an example, consider Canadian post codes, such as L5J 2V4 or V6Z 1M5. The JavaScript Regular Expression pattern that constrains these data values to the desired format is [A-Z]\d[A-Z] \d[A-Z]\d.

For Integer, Double, or Date data, you can define both a range constraint and a JavaScript Regular Expression pattern. When the type restriction is used, the two conditions are combined using the logical AND operation.

13.18.1.4 Associating Type Restrictions with Inline Service Objects

After you have created a Type Restriction, you can associate it with one or more of the following Inline Service objects in the corresponding object editor:

- Session and entity attributes
- Choice group attributes
- Choice attributes
- Function parameters
- Application parameters

For more information about using the appropriate object editors, see the corresponding object-related sections in this documentation.

13.18.1.5 Using Type Restrictions in Rules

Associating a Type Restriction with an object helps business users formulate rules for that object in the Rule Editor.

When you create or edit a rule, you have the following options:

- If you select a List of Values or List of Entities type-restricted object as an operand, a dropdown list of values becomes available for the other rule operand. You may select a value from this dropdown list, but you do not have to.
- If you select an Other Restrictions type-restricted object as an operand, and mouse hover over the operand, the range constraint or the JavaScript Regular Expression for the type restriction appears in the hover help.

> **Note:** Type Restrictions are an aid, not a strict requirement, for creating or editing rules. For example, you may enter values in rules for type-restricted elements other than those that appear in the dropdown lists. At run time, rules are evaluated and acted upon, not Type Restrictions.

In general:

- List of Values type restrictions present constant lists
- List of Entities type restrictions present dynamic run time lists
- Other Restrictions enable design type validation other than checking against list entries

Any violation of a type restriction constraint is treated as a warning rather than an error, as follows:

- The violating operand is underlined with a red wiggly line
- When you mouse hover over the offending operand, you get detailed information about the violation

The rule can still be compiled and deployed with warnings.

13.18.1.6 Examples of Type Restrictions

This section shows examples of how to create and use various type restrictions.

1. **"List of Values" Type Restriction**

Create a type restriction called Product_Size, with values "Large", "Medium", and "Small."

Attach the Product_Size type restriction to the attribute session/Product.

In the Rule Editor, create the filtering rule Large_Product as follows:

- session/Product = "Large"

2. **"List of Entities" Type Restriction**

   Create an entity New_Province, with String attributes Code and Fullname.

   Create a function Fn_New_Provinces, which returns a value of data type New_Province, with the following Logic:

```
SDProvinceArray provinces = new SDProvinceArray();
Province nb = new Province();
nb.setCode("NB");
nb.setFullname("New Brunswick");
provinces.add(nb);

Province nf = new Province();
nf.setCode("NF");
nf.setFullname("Newfoundland");
provinces.add(nf);

return provinces;
```

   Create a String type restriction, Province_Restriction, with the following characteristics:

   - Entity= New_Province
   - Function=Fn_New_Provinces
   - Label=Fullname
   - Value=Code

   Deploy the Inline Service. This is to enable correct values to appear in a subsequent dropdown list of values of the Rule Editor.

   Attach Province_Restriction as a type restriction for the entity attribute session/Address/Province.

   In the Rule Editor, create the filtering rule NF as follows:

   - session/Address/Province = "NF"

   The dropdown list for Province values shows Fullname values. The selected value, which is placed in the rule, is a Code.

3. **"List of Entities" Type Restriction Showing Dynamic Values**

   This example enables you to view and select values from a database table when you use a type-restricted object in a rule.

   Create an entity Look Up, with String attributes Statecode and Statename.

   Create a data source States DS based on a table States, with columns Abbr and Fullname.

   Create an entity US States Look Up, with the following characteristics:

   - An array attribute lookUp, of type Look Up
   - The attribute lookUp/Statecode mapped to States DS/Abbr

- The attribute lookUp/Statename mapped to States DS/AbbrFullname

Create a function Look Up States, which returns a value of data type Look Up, with the following Logic:

```
return new UsStatesLookUp().getLookUp();
```

Create a String type restriction, States Input, with the following characteristics:

- Entity= Look Up
- Function=Look Up States
- Label=Statename
- Value=Statecode

Deploy the Inline Service. This is to enable correct values to appear in a subsequent dropdown list of values of the Rule Editor.

Attach States Input as a type restriction for the entity attribute session/Address/State.

In the Rule Editor, create the filtering rule ND as follows:

- session/Address/State = "ND"

The dropdown list for State values shows Statename values. When you select "North Dakota", the corresponding Statecode value "ND" is placed in the rule.

4. **Other Type Restriction Using a Range**

   Create a Date type restriction called From_Now_On, with a minimum inclusive lower value of the current date.

   Attach the From_Now_On type restriction to the attribute session/Acct/Pay_By.

   In the Rule Editor, create the filtering rule PayUp as follows:

   - session/Acct/Pay_By = *<current_date> + 45 days*

5. **Other Type Restriction Using a JavaScript Regular Expression Pattern**

   Create a Date type restriction called Morning Rush Hour, that restricts values to times from 8:00 AM to 9:59 AM, with the following JavaScript RegEx pattern:

```
^(?=\d)(?:(?:(?:(?:(?:0?[13578]|1[02])(\/|-|\.)31)\1|(?:(?:0?[1,3-9]|1[0-2])(\/
|-|\.)(?:29|30)\2))(?:(?:1[6-9]|[2-9]\d)?\d{2})|(?:0?2(\/|-|\.)29\3(?:(?:(?:1[6
-9]|[2-9]\d)?(?:0[48]|[2468][048]|[13579][26])|(?:(?:16|[2468][048]|[3579][26])
00))))|(?:(?:0?[1-9])|(?:1[0-2]))(\/|-|\.)(?:0?[1-9]|1\d|2[0-8])\4(?:(?:1[6-9]|
[2-9]\d)?\d{2}))($|\ (?=\d))?(((0?[8-9])(:[0-5]\d){0,2}(\
AM))|(0?[8-9])(:[0-5]\d){1,2})?$
```

   Attach the Morning Rush Hour type restriction to the attribute session/Traffic/Morning Rush Hour.

   In the Rule Editor, create the filtering rule Mid_MRH as follows:

   - session/Traffic/Morning Rush Hour = <*current_date*> 9:00 AM

13.19 About Statistic Collectors

Statistic collectors manage the collection and lifecycle of ad hoc Inline Service statistics. A Choice Event Statistics Collector is created by default for each Inline Service. Choice Event Statistics Collectors automatically collect statistics for the events defined by your Inline Service. Statistics collectors have the following properties:

- **Description:** The description of the statistics collector.
- **Collect Statistics On:** Statistics can be collected either for each object, such as Choice or Choice Group, individually, or aggregated for all objects of the same type.
- **Aggregation:** Either record individual events, or record aggregated data. Care should be used in recording individual events, as high transactional systems may suffer from performance issues.
- **Aggregation Interval:** Amount of time in seconds to aggregate data before recording it through the Statistic Collector.
- **Expiration:** Choose either **Keep forever** or **Purge old statistics**. Care should be used in choosing **Keep forever**, as data size can be an issue.
- **Keep in database for:** Amount of time in days that data is kept before purging.

All parameters are configurable through the Decision Studio editor. Choice Event Statistics are displayed as a report in Decision Center.

13.19.1 Creating a Custom Statistics Collector

Using Decision Studio, you can create a custom Statistics Collector to record additional statistics about objects or classes. For instance, you can create a statistics collector to record statistics about Choices. Configure the parameters as described in the previous section.

In code in your Inline Service (for instance, in an Informant or through a Function Call), create a Statistics Collector Factory, passing in the Statistics Collector Name (String) or the statistic type (String):

```
StatisticCollectorFactory factory = Application.getCollectorFactory(<stat
collector name | statistic type>);
```

Using the factory, create a collector, passing in the event name on which you want to collect statistics (String) or the statistic name (String):

```
StatCollectorInterface collector = factory.getCollector(<event name | statistic
name> );
```

The event name or statistic name is an arbitrary string that represents what you want to collect.

Then, finally, using the collector, record the event passing in the object_type (String), object_id (String), event value (double), and extra data (string) to record:

```
Collector.recordEvent(<object_type>, <object_id>, event value, extra data);
```

The object_type must be a valid Object type, such as Choice, Choice Group, Entity, and so on. The object_id is the internal name of the object.

13.20 About Decision Center Perspectives

Like Decision Studio, Decision Center lets you work with an Inline Service from several perspectives. A perspective defines the initial layout in Decision Center. Each perspective provides a set of functionality aimed at accomplishing a specific type of task, and works with specific types of resources. Perspectives control what appears in certain menus and toolbars.

Decision Center has three default perspectives: Explore, Design, and At a Glance. The Inline Service Navigator changes according to the perspective you are using. Your system administrator may have added additional perspectives.

You can control access to Decision Center perspectives by assigning permissions to roles. See *Oracle Real-Time Decisions Installation and Administration Guide* for information about managing roles.

To assign permissions for perspectives, go to the Inline Service Navigator in Decision Studio and right-click the perspective for which you want to set access. Choose **Properties**, then click **Add**. Select the role to which you want to assign permissions, then select **Use perspective** under **Permissions**. Click **OK** to finish.

14

Deploying, Testing, and Debugging Inline Services

This chapter describes how to deploy, test, and debug Inline Services.

This chapter contains the following topics:

14.1 Deploying Inline Services

After you have configured your Inline Service, you deploy it locally or to a test environment for testing. You can deploy an Inline Service in three different states: Development, QA, and Production.

You can deploy in the QA state during your testing cycle, and then, after testing, into Production state. When you deploy to Production state, select **Release Inline Service locks**. After the Inline Service is deployed to business users, they can also update and redeploy the Inline Service.

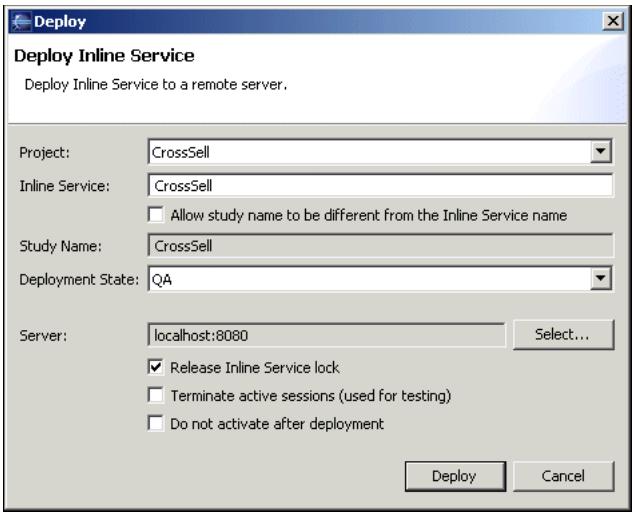
Note: Inline Services can be deployed both through Decision Studio and through a command line deployment tool.

This section describes deployment from Decision Studio. For information about command line deployment, see the chapter "Command Line Deployment of Inline Services" in *Oracle Real-Time Decisions Installation and Administration Guide*

Deploy the Inline Service using the **Project > Deploy** menu item, or click the Deploy icon on the task bar:

The Deploy dialog box appears. The options in the Deploy dialog box are shown in Figure 14–1.

Figure 14–1 Deploy Dialog Box



Note: You must have the proper permissions on the server cluster to deploy an Inline Service. See *Oracle Real-Time Decisions Installation and Administration Guide* for more information about cluster permissions.

Table 14–1 describes the options shown in the Deploy dialog box.

Table 14–1 Options in the Deploy Dialog Box

Option Name	Description
Project	Choose the project that you will deploy to Real-Time Decision Server.
Inline Service	The Inline Service contained in this project.
Study Name	Enter a study name for this Inline Service. Each Inline Service's learnings are associated with a study name. If you want to redeploy an Inline service and restart its learnings, deploy it with a new study name. Different study names can be used for Development, QA, and Production.
Deployment State	The default deployment states of Inline Services are Development, QA, or Production. Deployment state marks an Inline Service that is in development, testing, or production so that others are aware of its state. Your system administrator may have created custom deployment states.
Server	Click this option to enter the server and port to which you want to deploy. In the **Server** dialog box, provide a valid username and password that has deployment authorization on the server cluster to which you are deploying. Cluster authorization is granted through JConsole by your administrator.
Release Inline Service lock	A deployed Inline Service is automatically locked, and only the user who deployed it is able to redeploy the Inline Service. Once you have completed development and testing and are deploying the Inline Service for production, select **Release Inline Service lock** to allow Decision Center users to make changes and redeploy the Inline Service.

Table 14–1 (Cont.) Options in the Deploy Dialog Box

Option Name	Description
Terminate Active Sessions	If the Inline Service you are deploying is in production, there may be active sessions. If a new version of the Inline service is deployed while there are active sessions, the older version will be maintained to service those sessions. Select **Terminate Active Sessions** to terminate the active sessions if you are in testing. For a production Inline Service, keep this option deselected so that any active sessions will continue to run on the production version of the Inline Service. New sessions will be routed to the new version, and the old version will terminate when all active sessions have completed.
Do not activate after deployment	Use this option to deploy the Inline Service to the server, but not start the process. If you would like to activate the Inline Service at a later date, use JConsole. For more information about JConsole, see *Oracle Real-Time Decisions Installation and Administration Guide*.

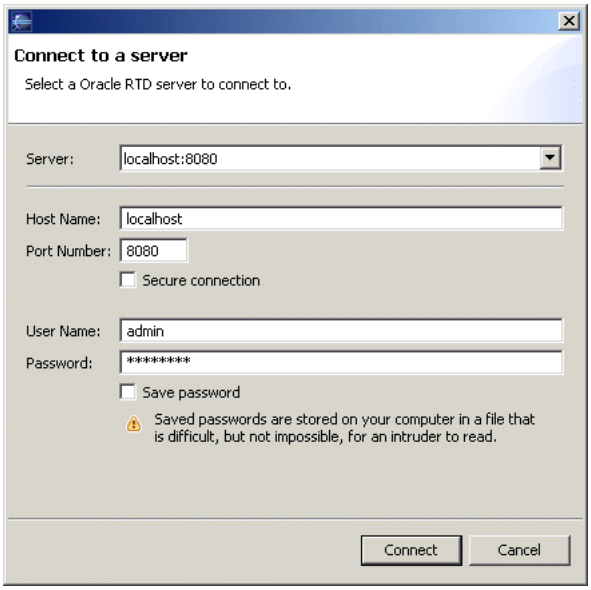
Note: After an Inline Service with custom pages is deployed to a certain deployment state, for example Development, the list of custom pages for this state will override the list of custom pages for the same Inline Service previously deployed to any deployment state.

14.2 Connecting to Real-Time Decision Server

When deploying or downloading Inline Services or importing data sources, you connect to Real-Time Decision Server. To connect, use the username and password you created on installation, or consult your Administrator for your username and password. To connect in a secure manner using https, select **Secure connection**.

Figure 14–2 shows the Connect to a Server dialog box.

Figure 14–2 Connect to a Server Dialog Box



14.3 Redeploying Inline Services

If you are going to make changes to a deployed Inline Service, it is important to follow these practices in order to preserve both your changes and the potential changes that have been made by the business user. If you are making changes to a deployed Inline Service, you can download it from Real-Time Decision Server using the **Download** icon on the toolbar. Use the following method:

1. Make sure that no business users are editing the deployed Inline Service.
2. Always lock an Inline Service when you download it, so that additional changes cannot be made by business users while you are enhancing it.
3. Make enhancements in Decision Studio.
4. Redeploy the Inline Service, releasing the locks.

During the period that you have the Inline Service locked, business users will be able to view, but not edit, the deployed Inline Service.

Note: While you must use Decision Studio for Inline Service enhancement, you can redeploy the Inline Service either in Decision Studio or by using a command line deployment tool. For more information, see the chapter "Command Line Deployment of Inline Services" in *Oracle Real-Time Decisions Installation and Administration Guide*

14.4 Testing and Debugging Inline Services

To enable you to test and debug Inline Services, Oracle RTD provides the following facilities and features:

- Problems and Test Views in Decision Studio
- System Logs
- Load Generator

For error messages that may occur during development, refer to Appendix A, "Development Error Messages."

This section contains the following topics:

- Section 14.4.1, "About the Problems View"
- Section 14.4.2, "Using the Test View"
- Section 14.4.3, "Using System Logs for Testing and Debugging Inline Services"

Load Generator is a tool used for debugging and benchmarking Inline Services by simulating users. Load Generator is used both for testing the Inline Service, and for performance characterization.

For more information about Load Generator, see Chapter 15, "About the Load Generator."

14.4.1 About the Problems View

The Problems view identifies compilation errors and validation errors as the Inline service is built. Double-click a compilation error to display the Java perspective with the error highlighted.

Double-click a validation error to display the Inline Service perspective with the element editor for the element that has validation errors.

14.4.2 Using the Test View

Decision Studio includes a Test view where you can test individual Integration Points. The Test view allows you to simulate the operational systems that will call the Integration Points. The Test view has a drop-down list of all Integration Points in the Inline Service. To test the Integration Point, insert values for the session key and request data and click the run icon to run. Three subtabs provide information about the Integration Point: Results, Trace, and Log.

The Results tab shows the results of calling an Advisor Integration Point. Only Advisors return results. For testing Informants and for debugging both kinds of Integration Points, use `logInfo()`.

You can use the statement `logInfo()` at various points in your code as a debugging device. This statement is helpful to use in elements such as Advisors or Informants, Decisions, functions, and so on. Insert the statement into the logic pane of the element and use it as a device to display in the log data at different stages.

This section contains the following topics:

- Section 14.4.2.1, "Using logInfo()"
- Section 14.4.2.2, "Testing for Incoming Request Data"

14.4.2.1 Using logInfo()

The **Log** tab gives a view of all `logInfo()` statements.

The `logInfo` method is part of the logging API described in the Decision Studio online help. The class `com.sigmadynamics.supportClass SDOBase` contains methods for logging messages at the informational, debug, warning, and error levels. These logging methods generally accept a string and another argument as parameters.

Two examples of using `logInfo` are as follows:

```
logInfo("Installation date = " + DateUtil.toString(session().getCustomer().
getInstallationDate());

logInfo("Customer age = " + session().getCustomer().getAge() );
```

> **Note:** For information on how to output to the Oracle RTD log the value of all session entity attributes or of a specified entity and its attributes, see Section 13.5.21, "Enhanced Entity Attribute Logging."

14.4.2.2 Testing for Incoming Request Data

When testing an Integration Point, you can check for the incoming request data using the following methods. If the incoming parameter is mapped to a session attribute, there is a `get` method for the parameter:

```
request.get$()
```

where `$` is the parameter name with the first letter capitalized.

If the attribute is not mapped, there are methods to achieve the same results using the field name of the parameter:

```
String request.getStringValue(fieldName)
```

```
SDStringArray request.getStringArrayValue(fieldName)
boolean request.isArgPresent(fieldName)
```

Outgoing response data is always stored in a SDChoiceArray:

```
SDChoiceArray choices = null;
```

The Decision is executed by the Integration Point, and the Choice is stored:

```
if (session().isControlGroup()) {
  choices = RandomChoice.execute();
} else {
  choices = SelectOffer.execute();
}
```

To find out what the Choice is, you can get them from the array and use `getSDOId` or `getSDOLabel`.

```
if (choices.size() > 0) {
  Choice ch = choices.get(0);
  ch.getSDOId();
}
```

The best place to do this is in the Post Selection Logic of the Decision. After the Decision executes, the post selection logic will run.

14.4.3 Using System Logs for Testing and Debugging Inline Services

You can use system logs to help you test and debug Inline Services, and also for performance monitoring and tuning.

The main Oracle RTD system log file is at the following location:

- *RTD_RUNTIME_HOME*\log\server.log

For more information, see Appendix A, "System Log and Configuration Files" in *Oracle Real-Time Decisions Installation and Administration Guide*.

15

About the Load Generator

Load Generator is a tool used for debugging and benchmarking Inline Services by simulating decision requests. Load Generator is used both for testing the Inline Service, and for performance characterization.

You can access Load Generator by opening *RTD_HOME*\scripts\loadgen.cmd. For a sample Load Generator script, see the etc directory of the Cross Sell example.

Load Generator has four tabs:

- **Run:** Runs a load generator session and provides feedback on the performance through measurement and graphs.
- **General:** Sets the general settings for Load Generator's operation, including the rate at which data is sent to the server and the location of the client configuration file.
- **Variables:** Used to create script, message, and access variables.
- **Edit Script:** Used to set up the script that specifies the Integration Point requests to be sent to the server.

This section contains the following topics:

15.1 Using Load Generator for Testing

Load Generator is used to generate load on the server to test it for performance and scalability. Intelligently random messages are sent to the Inline Service, allowing the models to learn. The capability of your models can be gauged after running Load Generator for a sufficient period of time.

15.2 Using Load Generator for Performance Characterization

Once an Inline Service is configured, Load Generator is used to evaluate how the service performs under load in order to assess how many servers are needed for specific loads. When you want to stress the server, typically one instance of Load Generator running on one client machine is sufficient, because Load Generator can engage many threads of execution to run multiple scripts concurrently. If additional load is desired and Microsoft Task Manager shows that Load Generator is already consuming the majority of the client's processing power, then several instances of Load Generator can be started on several client computers and pointed to one server. They send messages with some intelligently random generated messages in the context of sessions. The clients measure performance statistics, as well as the server.

15.3 Running a Load Generator Session

To start a session, first create a new script, or load an existing one. Then, select the **Run** option from the **Run** menu, or click the **Run** icon on the toolbar. You can alter the delay between data samples on the **General** tab.

15.3.1 Measuring the Server Load

The **Run** tab displays real-time information about the session running. Table 15–1 describes the options on the **Run** tab.

Table 15–1 Options on the Load Generator Run Tab

Option Name	Description
New Requests	The number of requests that have been closed since the previous data sample was taken.
New Errors	The number of errors, either client or server side, that have occurred since the previous data sample was recorded.
New Default Responses	The number of errors since the last data sample, that occurred for Advisor Integration Point requests (as opposed to Informant Integration Point requests) and a default response was defined by the Inline Service for the Advisor.
Active Scripts	Number of simulated users currently connected to the server from this load generator.
Peak Response Time	The length of time it took to close the oldest request during the current data sample.
Total Requests	The total number of requests that have been closed.
Total Errors	The total number of errors.
Total Default Responses	The total number of default responses.
Total Finished Scripts	The total number of simulated users.
Average Script Duration	The length in milliseconds of an average script's execution, from start to finish.

15.4 Viewing Performance Graphs

By default, the **Requests per Second** graph is visible. You can hide and show graphs by selecting **View > Graphs**. To clear the data in the graphs, select **View > Clear Graphs**, or click the **Clear Graphs** icon on the toolbar:

If you stop a script and restart it, all recorded data will be cleared. However, if you pause a session and then start it again, the data will not be cleared. The following graphs are available:

- **Average Response Time:** A histogram depicting the 40 most recent average response times.
- **Errors:** A line graph depicting the number of errors that occurred within the most recent 12,000 data samples.
- **Peak Response Time:** A line graph depicting the peak response time, in milliseconds, that occurred within each of the most recent 12,000 data samples.
- **Requests Per Second:** A line graph depicting the average number of requests per second that occurred within each of the most recent 12,000 data samples.
- **Requests Per Second distribution:** A histogram depicting the 40 most recent readings for requests per second.

15.5 About the General Tab

The **General** tab contains variables about Load Generator's configuration, timing, and which Inline Service is being specified. The **General** tab has five sections: Load Generator, Details, Think Time, Scripts, and Logging.

This section contains the following topics:

15.5.1 Load Generator Section

The Load Generator section of the **General** tab contains the following options:

- **Client Configuration:** Describes which endpoints Load Generator should use to contact the server.
- **Graphs Refresh Interval in Seconds:** Sets the delay between graph and counter updates. Click **Apply** for settings to take effect while a script is already running.

15.5.2 Details Section

The Details section of the **General** tab contains the following options:

- **Inline Service:** The name of the Inline Service to which this script will send requests.
- **Random Number Generator Seed:** If your script has any random elements in it, this gives you the ability to reproduce, to some extent, the random behavior. Repeatable randomness is not possible when running more than one concurrent script (see **Number of Concurrent Scripts to Run** in Section 15.5.4, "Scripts Section").

15.5.3 Think Time Section

The Think Time section of the **General** tab contains the following options:

- **Fixed Global Think Time:** The number of seconds that all simulated users will wait between requests.
- **Ranged Global Think Time:** A variable time that simulated users wait between requests. The think time changes by either a random number, or a sequentially increasing number from a set number range.
- **Minimum:** A nonzero number of seconds to wait at a minimum.
- **Maximum:** A nonzero number of seconds to wait at a maximum (must be greater than minimum).
- **Access Type Sequential:** At each access, increase the think time by one until you reach the maximum, when it will reset to the minimum.
- **Access Type Random:** At each access, choose a value between minimum and maximum, inclusive of each.

15.5.4 Scripts Section

The Scripts section of the **General** tab contains the following options:

- **Number of Concurrent Scripts to Run:** The number of simultaneous users to simulate.
- **Maximum Number of Scripts to Run:** A positive number in this field causes Load Generator to stop running after that number of sessions have completed. Zero means unlimited.

15.5.5 Logging Section

The Logging section of the **General** tab contains the following options:

- **Enable Logging:** When this option is selected, Load Generator statistics data is periodically written to a file.
- **Append to Existing File:** When this option is selected, and logging is enabled, Load Generator will append new statistics data onto the end of an existing log file, if any, or else it will create a new file.
- **Log File:** The full path to the log file, a tab-separated file whose contents are described in Section 15.8, "Load Generator CSV Log File Contents."
- **Logging Interval in Seconds:** The number of seconds to wait after appending values onto the log file before writing the next set of values.

15.6 About Variables

Variables allow a load simulation to draw its input from many different sources. Session variables are generated once per session. Subsequent accesses to a session variable use the same value. Message variables are held constant for a single request. Access variables may vary every time they are read. Variables are used in Message Actions.

This section contains the following topics:

15.6.1 Using Variables

To use a variable in a message (in the Edit Script tab) for a value to a message parameter, select it from the drop-down list in the Variable column. However, to use it as part of an concatenated string in the Value column, surround the variable name with braces (for example, C001-`{customerNum}`).

15.6.2 Variable Types

There are five types of variables:

- **Constant Value:** A constant value.
- **Integer Range:** Select an integer from a range. For example:

  ```
  Minimum: 0, Maximum: 50000, Access Type: Random
  Minimum: 0, Maximum: 1, Access Type: Sequential
  ```

- **String Array:** Select a string from the specified array. For example:

  ```
  List: [A, B, C], Access Type: Random
  List: [Male, Female], Access Type: Sequential
  ```

- **Weighted String Array:** Select from the specified array a string with some likelihood [0,1]. For example:

  ```
  List: [[0.3, Interested], [0.3, Accepted], [0.4, Rejected]]
  List: [[0.999, Interested], [0.001, Accepted]]
  ```

- **Text File:** Select a line of text from a file. For example:

  ```
  c:/data.txt, Access Type: Sequential
  ```

  This example shows an absolute reference to a file on the C: drive.

  ```
  inbox/data.txt, Access Type: Random
  ```

  This example shows a relative reference to a file in the inbox directory, under the directory containing the script file.

15.7 About Actions

In order to easily simulate multiple clients supplying realistic loads to the server, messages can be generated from patterns specified in metadata that are interpreted by Load Generator at run time. These patterns of actions are defined in the Edit Script tab.

The patterns specify message sequences, with fixed or random inter-message delays (think times), as well as patterns for generating values for message fields. Message field values can be literal strings, with optional embedded random characters, or they can be randomly selected from a set of values associated with the field.

Sessions are supported, allowing certain fields to remain constant across messages of the session, suitable for representing session keys (for example, a customer ID, call ID, or account number).

The patterns allow some flexibility in the sequencing of messages. For example, in a typical session, certain messages will come before others, or a predetermined number of messages of certain kind need to happen, and so on.

15.7.1 Types of Actions

There are two types of actions: Message and Loop.

Message has the following attributes:

- **Integration Point name:** The name of the Integration Point that will be sent the message.
- **Session Keys and values:** The values sent to the Integration Point request. Session keys have to be separated from other message fields because the server uses them for routing.

Loop has one attribute, **Number of times to execute**. This attribute can be a constant value, or a range value. A range value executes either sequentially, or randomly within the range.

15.8 Load Generator CSV Log File Contents

Table 15–2 describes the fields of the comma-separated value (CSV) file containing Load Generator statistics.

Table 15–2 Load Generator CSV File Fields

Field Name	Description
Date/Time	The time of day at which the current row of counters was appended to the file. Millisecond precision is available to facilitate correlations with messages in the server's log file.
Thread Pool Size	The number of threads engaged or available to run scripts. This is an implementation detail of little to interest to most people.
New Requests	The number of requests that have been closed since the previous data sample was taken.
Total Requests	The total number of requests that have been closed.
New Errors	The number of errors, either client or server side, that have occurred since the previous data sample was recorded.
Total Errors	The total number of errors.
New Default Responses	The number of errors since the last data sample, that occurred for Advisor Integration Point requests (as opposed to Informant Integration Point requests) and a default response was defined by the Inline Service for the Advisor.
Total Default Responses	The total number of default responses.
Active Scripts	Number of simulated users currently connected to the server from this Load Generator.
Total Scripts	The total number of simulated users.
Average Response Time (ms)	The average length of time it took to close the oldest request during the current data sample.
Max Response Time (ms)	The maximum length of time it took to close the oldest request during the current data sample.
Average Script Duration (ms)	The length in milliseconds of an average script's execution, from start to finish.
Snapshot Period (ms)	The number of milliseconds during which the current counter values were accumulated.

15.9 XLS File Contents

This section describes the contents of the Microsoft Excel file, `lg_perf.xls`, included in the `etc` directory of the installation for the purpose of rendering the Load Generator counters written to `log/loadgen.csv`.

At the top, cell A1 contains a comment describing how to link `lg_perf.xls` to the tab-separated counter file as a datasource, as follows:

"To specify the path to the Load Generator performance log, place cursor the in cell A2 and select "Import External Data" > "Edit Text Import" from the "Data" menu, and navigate to the path specified in your loadgen configuration, typically {$install_ directory}\log\loadgen.csv. Use default parsing settings when prompted. Data will then be automatically refreshed every 3 minutes. To change interval and other settings, select from the "Data" menu the selection "Import External Data" > "Data Range Properties"."

In row 2 are the headers containing the names of each counter. All of the headers from the CSV file, described above, appear here, with values below them.

Part IV

Miscellaneous Application Development

The chapters in Part IV provide an in-depth look at the concepts, components, and processes involved in Oracle RTD application development that require special processing, such as batch framework and external editors that enable modification of Oracle RTD application objects.

Part IV contains the following chapters:

16

Oracle RTD Batch Framework

Oracle RTD Batch Framework is a set of components that can be used to provide batch facilities in an Inline Service. This enables the Inline Service to be used not just for processing interactive Integration Point requests, but also for running a batch of operations of any kind. Typically, a batch will read a set of input rows from a database table, flat file, or spreadsheet, process each input row in turn, and optionally write one or more rows to an output table for each input row.

The following examples describe in outline form how you can use Oracle RTD batch processing facilities in a non-interactive setting:

- Create a "learning" batch to train models to learn from historical data about the effectiveness of offers previously presented to customers.
- Create an "offer selection" batch which starts with a set of customers, and selects the best product to offer to each customer.
- Create a "customer selection" batch which starts with a single product, and selects the best customers to whom to offer the product.
- Create a batch set of e-mails where Oracle RTD selects the right content for the e-mails

Within an Inline Service, the Inline Service developer defines one or more Java classes implementing the `BatchJob` interface, with one `BatchJob` for each named batch that the Inline Service wishes to support. In the Inline Service, each of the `BatchJob` implementations is registered with the Oracle RTD Batch framework, making the job types available to be started by an external batch administration application.

External applications may start, stop, and query the status of registered batch jobs through a BatchAdminClient class provided by the Batch Framework. The Batch Console, released with Oracle RTD, is a command-line utility that enables you to perform these batch-related administrative tasks.

Note: The following terms are referenced throughout the Oracle RTD documentation:

- *RTD_HOME*: This is the directory into which Oracle RTD is installed. For example, `C:\OracleBI\RTD`.
- *RTD_RUNTIME_HOME*: This is the application server specific directory in which the application server runs Oracle RTD.

For more information, see the Section "About the Oracle RTD Run-Time Environment" in *Oracle Real-Time Decisions Installation and Administration Guide*.

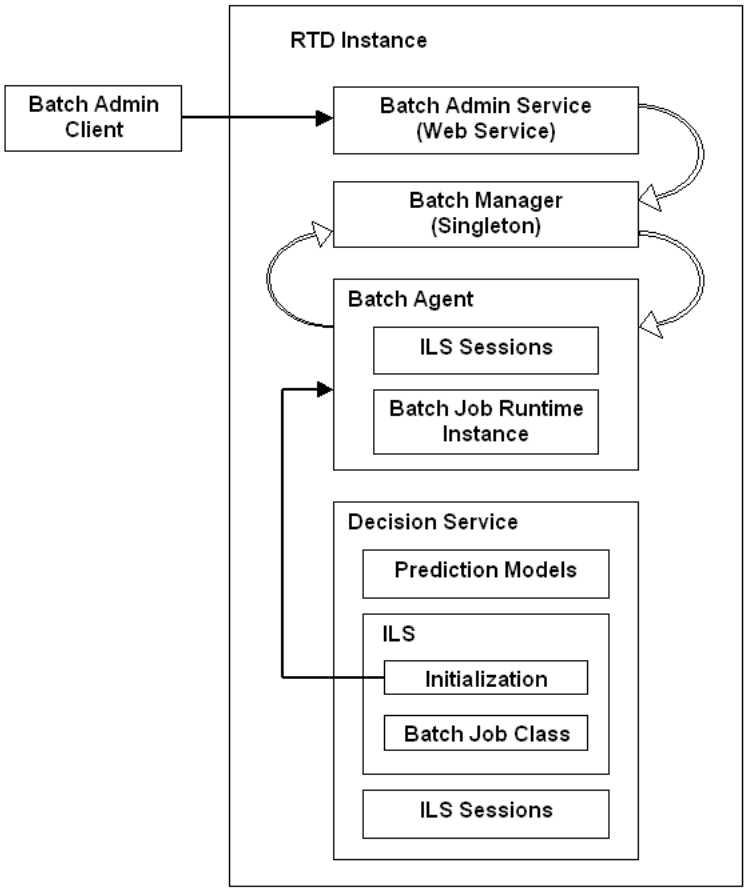
The topics in this section are the following:

16.1 Batch Framework Architecture

This section presents an overview of the components of the batch framework architecture and shows how batch facilities can be used across cluster servers.

16.1.1 Batch Framework Components

The following diagram shows the components of the batch framework architecture on a single Oracle RTD instance.



The main batch framework components and their functions are:

- Batch Admin Client

  The Batch Admin Client provides a set of Java APIs that can be used by Java client applications to manage batches registered on remote Real-Time Decision Servers. This includes starting and stopping batches, and obtaining batch status information.

Customers may create their own batch client application using the APIs provided in the Batch Admin Client.

The Batch Console is a client side command line utility that manages batches registered on remote Real-Time Decision Servers. Internally, the Batch Console uses the APIs provided by the Batch Admin Client.

- Batch Manager

  This is a cluster-wide singleton service, that executes client batch commands from client code from the Batch Admin Client.

  The Batch Manager manages each Batch Agent in the cluster.

  The Batch Manager also executes commands from the Batch Console.

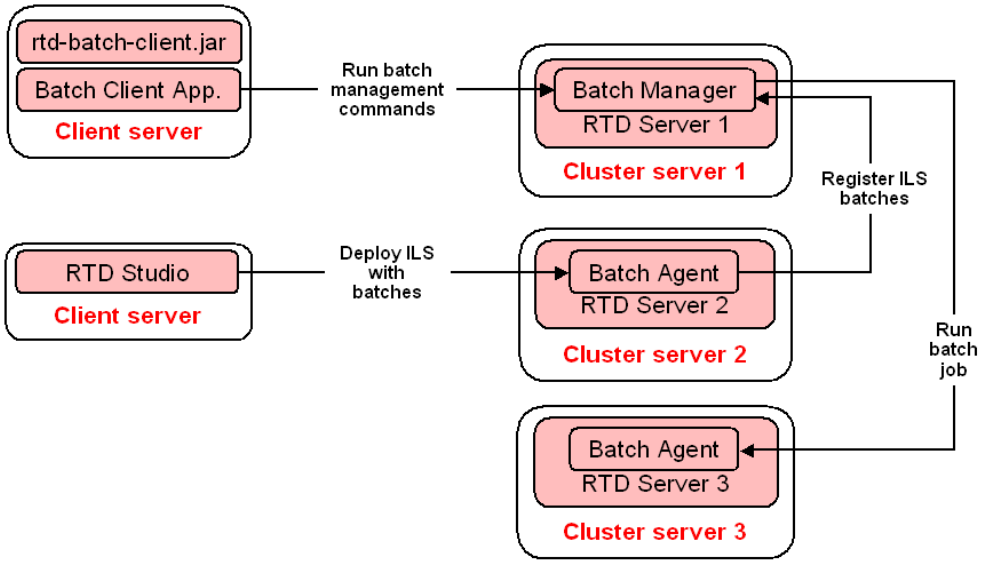
- Batch Agent

  The batch agent is the interface between a batch job and the batch framework. It is a service that registers batches with the Batch Manager when the batch-enabled Inline Service is deployed, and executes batch commands on behalf of the Batch Manager.

In a clustered environment, all the batch framework components appear in each Oracle RTD instance. However, the Batch Manager is only active in one of the instances, and that active Batch Manager controls all the Batch Admin Client and Batch Agent requests in the cluster.

16.1.2 Use of Batch Framework in a Clustered Environment

The following diagram illustrates an example of the use of the batch framework in a clustered environment.



A batch client application, such as the Batch Console, communicates with the Batch Manager, by invoking batch management commands, such as to start, stop, or pause a job.

Developers using Decision Studio can create and deploy Inline Services with batches to any instance where Oracle RTD is installed, such as that on Cluster server 2.

Note: In a clustered environment, Inline Services are deployed to all servers running the Decision Service.

The diagram shows the Batch Agent on the Cluster server 2 instance registering batches with the Batch Manager.

The Batch Manager can then run batch jobs on any instance, such as that on Cluster server 3, so long as they were previously registered.

16.2 Implementing Batch Jobs

This section presents an overview of the runtime object model required to implement batches.

In order for an Inline Service to be batch-enabled, it must contain one or more batch job Java classes implementing the `BatchJob` interface, and register them with the batch framework.

Note: The examples that appear in this section reference the **CrossSell** Inline Service released with Oracle RTD, which contains the batch job **CrossSellSelectOffers**.

This section consists of the following topics:

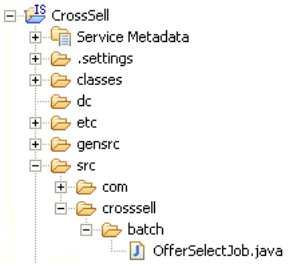
- Section 16.1.1, "Batch Framework Components"
- Section 16.1.2, "Use of Batch Framework in a Clustered Environment"

16.2.1 Implementing the BatchJob Interface

You start the implementation of a batch job in Decision Studio by creating a Java class that implements the `BatchJob` interface.

First, you create Java packages and classes under the `src` branch of the Inline Service.

The following image shows the "batch processing" Java class `OfferSelectJob.java` declared in the package `crosssell.batch`:



The easiest way to create the Java classes is to subclass from `BatchJobBase`, provided with the batch framework.

The principal methods of a batch job are called in the following sequence when the job is started:

1. **`init()`**

   Called once by the framework before starting the batch's processing loop.

2. **`getNextInput()`**

   Returns the next input row to be processed by the batch.

3. **`executeRow()`**

   The BatchJob implements this method to process the input row that was returned by `getNextInput`. Generally, this is called in a different thread from `getNextInput`.

4. **`flushOutputs()`**

   Called by the framework to allow the BatchJob to flush its output table buffers.

5. **`cleanup()`**

   Called by the framework after the batch is finished or is being stopped. Cleans up any resources allocated by the batch job, such as the result set created by its `init()` method.

For full details of the methods of the `BatchJob` interface, see the following Javadoc entry:

RTD_HOME`\client\Batch\javadocs\com\sigmadynamics\batch\BatchJob.html`

Batch Job Example

An example of a batch job, `OfferSelectJob.java`, appears in the CrossSell Inline Service released with Oracle RTD. This batch job selects the best offer for a set of customers, and saves the offers to a table.

16.2.2 Registering Batch Jobs with the Batch Framework

This section describes how to register the batch jobs with the Oracle RTD batch framework. You must register the Java classes that contain the batch jobs as imported Java classes, then you must explicitly register the batch jobs with the batch framework using the `batchAgent.registerBatch` method.

This section consists of the following topics:

- Section 16.2.2.1, "BatchAgent"
- Section 16.2.2.2, "Registering the Imported Java Classes in the Inline Service"
- Section 16.2.2.3, "Registering the Batch Jobs in the Inline Service"

16.2.2.1 BatchAgent

In a batch job, the batch agent is the interface between a batch job and the batch framework. You need to register the batch job with the batch framework.

An inline service can locate its batch agent through a getter in its Application object. For example, in a context where the Inline Service has access to a session, you can use the following command to access the BatchAgent:

- `BatchAgent batchAgent = session().getApp().getBatchAgent();`

16.2.2.2 Registering the Imported Java Classes in the Inline Service

You must register the Java classes in the Inline Service, as follows:

1. Click the Application object's **Advanced** button.

2. In the **Imported Java Classes** pane, enter one line for each batch job class in the Inline Service, of the form:

   <package>.<class>

   For example:

   ```
   crosssell.batch.OfferSelectJob
   ```

16.2.2.3 Registering the Batch Jobs in the Inline Service

An inline service must register its BatchJob implementations in the **Logic** tab of the Application, in the **Initialization Logic** pane, using the `batchAgent.registerBatch` API.

The Inline Service can locate its batch agent - its interface to the Batch Framework - through a getter in its Application object. Enter a line such as the following:

```
BatchAgent batchAgent = getBatchAgent();
```

followed by an invocation of `batchAgent.registerBatch` for each batch job in the Inline Service.

For full details of the parameters for `batchAgent.registerBatch`, see the following Javadoc entry:

RTD_HOME`\client\Batch\javadocs\com\sigmadynamics\batch\BatchAgent.html`

In summary form, the parameters for `batchAgent.registerBatch` are as follows:

- **batchName**: A short name used to register the batch class in the cluster. It should be unique across the cluster.
- **batchJobClass**: The fully qualified name of the batch's BatchJob implementation class.
- **description**: If non-null, a string describing the purpose of the batch.
- **parameterDescriptions**: An optional set of properties describing the parameters supported by the batch.
- **parameterDefaults**: An optional set of properties providing the default values for parameters supported by the batch.

For example, to register the following:

- The batch `CrossSellSelectOffers` that uses the class `crosssell.batch.OfferSelectJob`

enter the following in the Initialization Logic for the Application:

```
BatchAgent batchAgent = getBatchAgent();
batchAgent.registerBatch("CrossSellSelectOffers",
                "crosssell.batch.OfferSelectJob",
                OfferSelectJob.description,
                OfferSelectJob.paramDescriptions,
                OfferSelectJob.paramDefaults);
```

16.3 Administering Batch Jobs

The main way to administer batch jobs is though the command-line Batch Console utility, for example, to start, stop, and query the statuses of batches.

This utility uses the `BatchAdminClient` Java interface. The `BatchAdminClient` Java interface also provides methods for starting and managing batches for use by external programs.

This section contains the following topics:

- Section 16.3.1, "Using the BatchClientAdmin Interface"
- Section 16.3.2, "Using the Batch Console"

16.3.1 Using the BatchClientAdmin Interface

The `BatchAdminClient` Java interface provides methods for starting and managing batches for use by external programs.

Table 16–1 lists the methods for the `BatchAdminClient` interface.

Table 16–1 BatchAdminClient Methods

Return Type	Description
int	**clearBatchStatuses()** Removes batch status information for all batches that have completed.
int	**clearBatchStatuses(int numToKeep)** Removes batch status information for the oldest batches that have completed.
int	**clearBatchStatuses(java.lang.String batchName)** Removes batch status information for all batches that have completed and have the specified batch name.
int	**clearBatchStatuses(java.lang.String batchName, int numToKeep)** Removes batch status information for all batches that have completed and have the specified batch name.
BatchStatusBrief[]	**getActiveBatches()** Returns an ordered list, possibly empty, of brief status information for all batch jobs currently running, paused, or waiting to run.
java.lang.String	**getBatchDescription(java.lang.String batchName)** Returns a string, possibly empty, describing the purpose of the batch.
java.lang.String[]	**getBatchNames()** Gets a list of batches registered with the batch framework.
java.util.Properties	**getBatchParameterDefaults(java.lang.String batchName)** Gets properties containing the default values of the startup parameters supported by the batch.
java.util.Properties	**getBatchParameterDescriptions(java.lang.String batchName)** Gets properties describing the parameters supported by the batch.
BatchStatusBrief[]	**getJobHistory()** Returns an ordered list, possibly empty, of brief status information for all batch jobs whose status information is still retained by the batch manager -- those descriptions that have not been discarded by **clearBatchStatuses**.

Table 16–1 (Cont.) BatchAdminClient Methods

Return Type	Description
BatchStatusBrief[]	**getJobHistory(int maxToShow)** Returns an ordered list, possibly empty, of brief status information for all batch jobs whose status information is still retained by the batch manager -- those descriptions that have not been discarded by **clearBatchStatuses**.
BatchStatus	**getStatus(java.lang.String batchID)** Returns the status of a batch identified by the batchID that was returned when it was submitted by a call to **startBatch()**.
void	**pauseBatch(java.lang.String batchID)** Stops a batch and does not clean up its resources, so it can be resumed.
void	**restartBatch(java.lang.String batchID)** Restarts a stopped batch.
void	**resumeBatch(java.lang.String batchID)** Continues a paused batch.
java.lang.String	**startBatch(java.lang.String batchName)** Starts a batch in the default concurrency group with default start parameters.
java.lang.String	**startBatch(java.lang.String batchName, BatchRequest startParameters)** Starts a batch in the default concurrency group with the supplied start parameters.
java.lang.String	**startBatch(java.lang.String batchName, java.lang.String concurrencyGroup)** Starts a batch in the specified concurrency group using default start parameters.
java.lang.String	**startBatch(java.lang.String batchName, java.lang.String concurrencyGroup, BatchRequest startParameters)** Starts a batch in the specified concurrency group using the supplied start parameters.
void	**stopBatch(java.lang.String batchID)** Stops a batch and cleans up its resources by calling **BatchJob.cleanup()**.
void	**stopBatch(java.lang.String batchID, boolean discardSandboxes)** Stops a batch, cleans up its resources (by calling **BatchJob.cleanup()**), and optionally discards any learning data and output table records generated by the batch since its last checkpoint.

For full details of the `BatchAdminClient` interface, see the following Javadoc entry:

RTD_HOME`\client\Batch\javadocs\com\sigmadynamics\batch\client\BatchAdminClient.html`

16.3.2 Using the Batch Console

The Batch Console is a command-line utility, `batch-console.jar`. Use the Batch Console to start, stop, and query the status of batches.

To start the Batch Console, run the following commands:

1. `cd` *BATCH_HOME*

Typically, *BATCH_HOME* is `C:\OracleBI\RTD\client\Batch`.

2. `java [-Djavax.net.ssl.trustStore="<trust_store_location>"] -jar batch-console.jar -user <batch_user_name> -pw <batch_user_password> [-url <RTD_server_URL>] [-help]`

Notes:

1. You must enter batch user name and password information. If you do not specify values for the `-user` and `-pw` parameters, you will be prompted for them.
2. *<RTD_server_URL>* (default value `http://localhost:8080`) is the address of the Decision Service. In a cluster, it is typically the address of the load balancer's virtual address representing the Decision Service's J2EE cluster.
3. Use the `-Djavax.net.ssl.trustStore="<trust_store_location>"` parameter only if SSL is used to connect to the Real-Time Decision Server (that is, where `-sslConnection` is set to true), where *<trust_store_location>* is the full path of the truststore file. For example, `-Djavax.net.ssl.trustStore="C:\OracleBI\RTD\etc\ssl\sdtrust.store"`. In this case, *<RTD_server_URL>* should look like `https://localhost:8443`.
4. If you enter `-help`, with or without other command line parameters, a usage list appears of all the Batch Console command line parameters, including -help.

To see a list of the interactive commands within Batch Console, enter ? at the command prompt:

```
command <requiredParam>  -- [alias] Description

?                        -- Show this usage text
help                     -- Show this usage text
exit                     -- Terminate this program
quit                     -- Terminate this program
batchNames               -- [bn]      Show all registered Batch
batchDesc <batchName>    -- [bd]      Show Batch Description
paramDesc <batchName>    -- [pd]      Show a batch's Parameter Descriptions
paramDef <batchName>     -- [pdef]    Show a batch's Parameter Default values
addProp <key> <value>    -- [ap]      Add one Property for next job start
removeProp <key>         -- [rp]      Remove one startup Property
showAddedProps           -- [sap]     Show all Added startup Properties
removeAddedProps         -- [rap]     Remove all Added startup Properties
startJob <batchName>     -- [start]   Start a batch job, returning a jobID
startInGroup <batchName> <groupName>
                         -- [startg]  Start a batch job in a Concurrency Group
status <jobID>           -- [sts]     Show a job's detailed runtime Status
activeJobs               -- [jobs]    Show brief status of all running,
                                      paused, waiting jobs
jobHistory               -- [hist]    Show brief status of all submitted jobs
stopJob <jobID>          -- [stop]    Stop a job, without abililty to resume
stopJobDiscardSandbox <jobID>
                         -- [stopds]  Stop a job, without abililty
                                      to resume, discard learning sandboxes
restartJob <jobID>       -- [restart] Restart a batch job
pauseJob <jobID>         -- [pause]   Pause a job
resumeJob <jobID>        -- [resume]  Resume a paused job
discardStatusAll         -- [dsa]     Discard status information
                                      for all non-active jobs
```

```
discardStatusOld <numToKeep>
                          -- [dso]     Discard Status for oldest non-active jobs
discardStatusName <batchName>
                          -- [dsn]     Discard Status for non-active
                                       jobs of named batch
discardStatusNameOld <batchName> <numToKeep>
                          -- [dsno]    Discard Status for oldest
                                       non-active jobs of named batch
```

The rest of this section contains the following topics:

- Section 16.3.2.1, "Notes on Batch Console Commands"
- Section 16.3.2.2, "Running Jobs Sequentially"
- Section 16.3.2.3, "Running Jobs Concurrently"

16.3.2.1 Notes on Batch Console Commands

1. To get a list of registered batches, enter `bn` or `batchNames`.
2. To get the default parameter values for a batch, enter `paramDef <batchName>` or `pdef <batchName>`.

   For example, your batch may have the parameter values:

   - `sqlCustomers` - to select the customers to process
   - `rowsBetweenStatusUpdates` - to control how often to update the batch status

   The default values for these parameters could be as follows:

   - `sqlCustomers = SELECT Id FROM Customers WHERE Id < 300`
   - `rowsBetweenStatusUpdates = 1000`
3. To supply parameter values for the next batch invocation, use the `addProp` command, or its alias, `ap`.

   For example, you can override the `sqlCustomers` parameter to include all customers, with the following command:

   - `ap sqlCustomers SELECT Id FROM Customers`

   And if you want to update the batch status after every 1500 customers are processed, enter the following command:

   - `ap rowsBetweenStatusUpdates 1500`

   You can view all such explicitly added parameters with the `showAddedProps` command, or its alias, `sap`.

   For example, if you used the preceding `ap` commands, the `sap` output would be:

   ```
   Property                    Value
   --------                    -----
   rowsBetweenStatusUpdates    1500
   sqlCustomers                SELECT Id FROM Customers
   ```
4. To start a batch, use the `startJob` command, or its alias, `start`.

   The output will be similar to the following:

   - `batchID=batch-2`

The returned batchID, also known as a job-ID, identifies this job instance. You can use it to query the status of the job.

5. To see the runtime status of the job, pass its batchID value to the status command, or to its alias, sts.

   - sts batch-2

   The out put will be similar to the following:

```
ID         Name                  State         Rows Errors Restarts
--         ----                  -----         ---- ------ --------
batch-2    MyBatchJob1           Running      4,500      0        0

 SubmitDateTime     WaitTime     RunTime      Group    Server
 --------------     --------     ------       -----    ------
 06/24/08-10:25:37  0m, 0s       0m, 0s       Default  RTDServer
```

   If you run the status command later, you can see that the job finished without errors, after processing 50,000 customers in 9 minutes and 44 seconds:

```
ID         Name                  State         Rows Errors Restarts
--         ----                  -----         ---- ------ --------
batch-2    MyBatchJob1           Finished     50,000      0        0

 SubmitDateTime     WaitTime     RunTime      Group    Server
 --------------     --------     ------       -----    ------
 06/24/08-10:25:37  0m, 0s       9m, 44s      Default  RTDServer
```

16.3.2.2 Running Jobs Sequentially

When jobs are submitted to be started they are assigned to a concurrency group. If not specified, the default concurrency group is assigned, named **Default**.

Jobs in the same concurrency group run sequentially, one at a time, in the sequence that they were submitted to be started.

So if you start a second job before the first finishes, the second job will wait to start until after the first one finishes.

This section shows the starting of the batch MyBatchJob1, and then the starting of two other batches, MyBatchJob2, and MyBatchJob3.

Before starting MyBatchJob1, use the sap command to verify the console has the parameter values set for the two parameters, rowsBetweenStatusUpdates, and sqlCustomers.

After starting MyBatchJob1, clear these parameters using the removeAddedProps command (rap), so that the next two jobs will use default values for all their parameters.

The jobs command shows a brief status of all running and waiting jobs. It shows the first job running, and the other two waiting.

```
command: batchNames
        MyBatchJob1
        MyBatchJob2
        MyBatchJob3
        MyBatchJob4
        MyBatchJob5
command: showAddedProps
        Property                      Value
        --------                      -----
```

```
        rowsBetweenStatusUpdates       1500
        sqlCustomers                   SELECT Id FROM Customers
command: start MyBatchJob1
        batchID=batch-3
command: removeAddedProps
command: start MyBatchJob2
        batchID=batch-4
command: start MyBatchJob3
        batchID=batch-5
command: jobs
  ID         Name          State    Group    Server
  --         ----          -----    -----    ------
  batch-3    MyBatchJob1   Running  Default  RTDServer
  batch-4    MyBatchJob2   Waiting  Default  none
  batch-5    MyBatchJob3   Waiting  Default  none
```

16.3.2.3 Running Jobs Concurrently

The `startInGroup` command, or its alias, `startg`, may be used to assign a job to a specific concurrency group. Starting two jobs in different groups allows them to run at the same time.

For example:

```
command: startg MyBatchJob4 myGroup1
        batchID=batch-6
command: startg MyBatchJob5 myGroup2
        batchID=batch-7
command: jobs
  ID         Name          State    Group     Server
  --         ----          -----    -----     ------
  batch-6    MyBatchJob4   Running  myGroup1  RTDServer
  batch-7    MyBatchJob5   Running  myGroup2  RTDServer
```

Note: Jobs assigned to the same concurrency group may run on different servers, but the jobs cannot run concurrently. Only jobs in different groups are allowed to run concurrently.

17

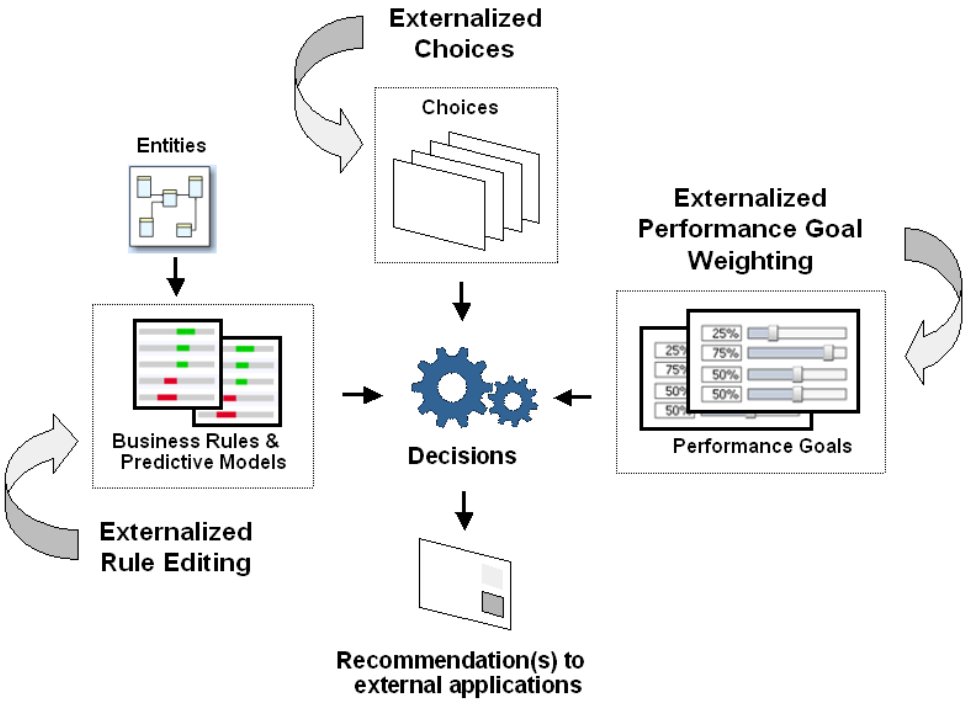
Externalized Objects Management

Oracle RTD produces adaptive enterprise software solutions by a process of continuously learning in real time from business process transactions as those transactions are executing. By continuously learning in real time, the adaptive solutions can optimize the outcome of each transaction and of the associated business process.

The basic framework of the Oracle RTD decisioning process is as follows:

- Oracle RTD makes analytic decisions for each interaction.
- Based on rules and predictive models, Oracle RTD decisions use real-time and historical data to make optimal recommendations from a variety of choices.
- In making the best recommendations, Oracle RTD optimizes across potentially conflicting business goals.

The following image shows the standard elements that form the framework for the Oracle RTD decision process, together with a significant set of inputs that enable external applications together with Oracle RTD to provide a composite decision service for their end users.



The standard elements of the Oracle RTD decision process - decisions, entities, choices, rules, models, and performance goals - are defined within Decision Studio. For general information about these elements, see Chapter 12, "About Decision Studio" and Chapter 13, "About Decision Studio Elements and APIs."

Oracle RTD can adapt to real-time changes to objects in external data systems, such as choices maintained in external content management systems.

Applications using Oracle RTD can also enable their users to make significant on-the-spot modifications to the decision process by creating and modifying the rules and by altering the performance goals that drive and control the overall decision process.

This chapter describes how these extensions to basic Oracle RTD elements are defined and used in the composite decision process, and how they are integrated with the external applications that they serve.

This chapter contains the following topics:

17.1 Dynamic Choices

In Oracle RTD, Choices represent the universe of alternatives, from which Oracle RTD can select its recommendations, such as the best offer in a cross selling application.

Choices can be either Static or Dynamic.

With Static Choices, the Choices to present to the requesting application or self-learning model are completely defined within Oracle RTD. Static Choices are useful in cases where the Choices are known in advance, and are constant over a period of time.

Dynamic Choices are Choices that are built dynamically at runtime. These Choices typically reside in external data sources. This allows for the management of Choices to be done at the source system, such as Choices based on offers defined in an offer management system.

Note: Although the main sources of Dynamic Choices are external data sources, you can also create Dynamic Choices with customized Java code.

The Dynamic Choices to be presented to an application may vary over time, but always reflect the up-to-date state of the application data. It is not necessary to redeploy the Oracle RTD Inline Service when dynamic choice content is updated.

Note: While this section focuses on Dynamic Choices, a Choice Group can contain a combination of Static and Dynamic Choices.

A Decision can be associated with one or more Choice Groups, no matter what type of Choice they contain.

This section contains the following topics:

17.1.1 Simple Example of Dynamic Choices

As a simple example, take the case of an **Insurance_Proposals** table, as shown in Figure 17–1. This table is defined outside of the Oracle RTD environment, and holds data about the Choices that Oracle RTD will evaluate and prioritize.

The **Insurance_Proposals** table contains rows for different insurance products, as identified by the common value **InsuranceProducts** in the **ChoiceGroupId** column. The column that categorizes or groups the Dynamic Choices is an important required key identifier for setting up Dynamic Choices.

Each row in the group shows a different type of insurance product being offered, such as **AutoInsurance**, and **DisabilityInsurance**. Each row represents a Dynamic Choice.

One column serves to identify the particular Dynamic Choice within the group. In this example, the **ChoiceID** column is the Dynamic Choice identifier column.

Other columns in the table, such as **ProfitMargin**, can be used by Oracle RTD in the evaluation process. These columns can also be sent back to the application as part of the Dynamic Choice recommendation, as a value for a defined Choice attribute.

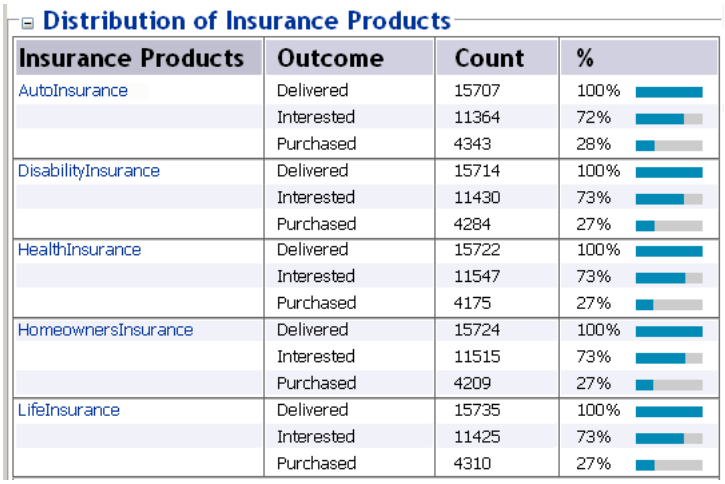
Figure 17–1 Insurance Products in the Insurance_Proposals Table

ChoiceId	ChoiceGroupId	ProfitMargin
AutoInsurance	InsuranceProducts	0.5
DisabilityInsurance	InsuranceProducts	0.5
HealthInsurance	InsuranceProducts	0.5
HomeownersInsurance	InsuranceProducts	0.5
LifeInsurance	InsuranceProducts	0.5

In short, the setup process is that, in Oracle RTD, you set up a Choice Group for Dynamic Choices, and associate this Choice Group with the required external Data Source or Sources. The Dynamic Choices are then available to be recommended by Oracle RTD.

After sufficient recommendations have been made and models have been updated for the corresponding Choice Group, you can analyze the performance of the various Dynamic Choices through Decision Center, as shown in Figure 17–2.

Figure 17–2 Decision Center Analysis of Dynamically Chosen Insurance Products



Distribution of Insurance Products

Insurance Products	Outcome	Count	%
AutoInsurance	Delivered	15707	100%
	Interested	11364	72%
	Purchased	4343	28%
DisabilityInsurance	Delivered	15714	100%
	Interested	11430	73%
	Purchased	4284	27%
HealthInsurance	Delivered	15722	100%
	Interested	11547	73%
	Purchased	4175	27%
HomeownersInsurance	Delivered	15724	100%
	Interested	11515	73%
	Purchased	4209	27%
LifeInsurance	Delivered	15735	100%
	Interested	11425	73%
	Purchased	4310	27%

17.1.2 Basic Dynamic Choice Design Implications

The basic design process for Dynamic Choices is similar to that for Static Choices. You must first set up a Choice Group, then define the required elements and parameters for Dynamic Choices in the Choice Group. For more detailed information on how to perform the setups, see Section 17.1.5, "Overview of Setting up Dynamic Choices in Decision Studio."

Using the Insurance_Proposals example, this section acts as an overview of the design process. It also introduces key terms used in the design process, as follows:

- The *set of all the Dynamic Choices* is identified as all the rows that have a common value in a "grouping" or categorizing column. In the Insurance_Proposals example, the categorizing column (or set identifier) is the **ChoiceGroupId** column.
- Each row in the database set represents a *single Dynamic Choice*. In the Insurance_Proposals example, the Dynamic Choice itself is identified by the unique value in the **ChoiceId** column.
- When you define the *Choice Group for Dynamic Choices* in Oracle RTD, you must link the Group to the set of rows that contain the Dynamic Choices.
- When you define the *Dynamic Choices in the Choice Group* in Oracle RTD, you must link each Dynamic Choice in the Group to the corresponding single Dynamic Choice row in the Data Source.

17.1.3 Multiple Category Dynamic Choices from a Single Data Source

In the simplest Dynamic Choice case, all the rows of the database table belong to the same category, that is, have the same value in a categorizing column.

You can provide different Dynamic Choices from either the same database table or a variety of data sources. The following example, as illustrated in Figure 17–3, shows the case where the Insurance_Proposals table is extended to provide Choices for both Insurance Products and Insurance Services.

Figure 17–3 Insurance Products and Insurance Services in the Insurance_Proposals Table

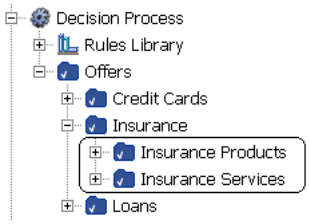
ChoiceId	ChoiceGroupId	ProfitMargin
AutoInsurance	InsuranceProducts	0.5
DisabilityInsurance	InsuranceProducts	0.5
HealthInsurance	InsuranceProducts	0.5
HomeownersInsurance	InsuranceProducts	0.5
LifeInsurance	InsuranceProducts	0.5
AutoServices	InsuranceServices	0.5
DisabilityServices	InsuranceServices	0.5
HealthServices	InsuranceServices	0.5
HomeownersServices	InsuranceServices	0.5
LifeServices	InsuranceServices	0.5

For this situation, you set up two Choice Groups in Oracle RTD, making both sets of data available for recommendations in the application.

After sufficient recommendations have been made and models have been updated for the corresponding Choice Group, you can analyze the performance of either or both of the Insurance Products and Insurance Services Dynamic Choices.

For example, the Choice Groups could have been set up as two groups in a group hierarchy, and available for analysis in Decision Center as shown in Figure 17–4.

Figure 17–4 Choice Groups in the Decision Center



Analyzing the Insurance Products provides the same results as shown in Figure 17–2. Figure 17–5 shows an equivalent analysis report for Insurance Services.

Figure 17–5 Decision Center Analysis of Dynamically Chosen Insurance Services

Distribution of Insurance Services

Insurance Services	Outcome	Count	%
AutoServices	Delivered	104	100%
	Interested	75	72%
	Purchased	29	28%
DisabilityServices	Delivered	103	100%
	Interested	71	69%
	Purchased	32	31%
HealthServices	Delivered	104	100%
	Interested	78	75%
	Purchased	26	25%
HomeownersServices	Delivered	104	100%
	Interested	74	71%
	Purchased	30	29%
LifeServices	Delivered	103	100%
	Interested	74	72%
	Purchased	29	28%

17.1.3.1 Different Dynamic Choice Categories in the Same Data Source

Choice Groups, unlike Choices, must be pre-defined. In effect, Choice Groups are static. You can group Dynamic Choices in separate Choice Groups if necessary to support reporting or decisioning requirements.

The design considerations for and components of each Choice Group are the same as described in Section 17.1.2, "Basic Dynamic Choice Design Implications."

For general information on how to set up Choice Groups, see Section 17.1.5, "Overview of Setting up Dynamic Choices in Decision Studio."

For specific information on how to set up different Choice Groups from the same Data Source, see Section 17.1.12, "Creating a Multi-Category Choice Group."

17.1.4 Prerequisite External Data Source for Dynamic Choices

The data required for Dynamic Choices exists in an external Data Source.

For the sake of simplicity, the following description assumes that the external Data Source is a database table or view in the calling application.

To be useful for Dynamic Choices, the data must include:

- One column to be used for categorizing and extracting the data.

  For a single Dynamic Choice, the rows to be extracted will all have the same value in the categorizing column, and this column is used to control the extraction.

  For example:

  – The database table **Special_Events** has a column **Event_Type**.

  – There are three distinct values of **Event_Type** across all the rows, namely **Promotion**, **Product_Launch**, and **Mailshot**.

  In this example, **Event_Type** is the categorizing column, and for a single Dynamic Choice, Oracle RTD will extract all the rows of one event type, such as all the **Promotion** rows.

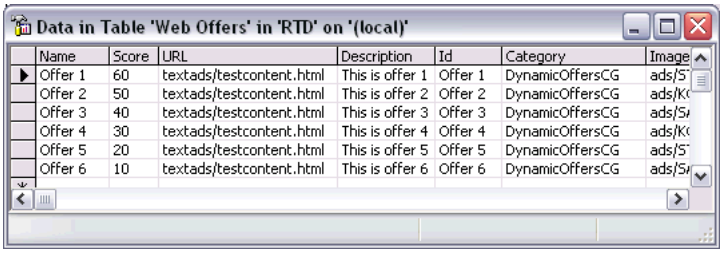
- One column that uniquely identifies the rows extracted for a particular Dynamic Choice.

  The column does not need to have unique values across all the rows, just within the extracted data set.

  Any column that provides a unique identifier within the extracted data is sufficient. Oracle recommends that the column values include some textual component. These values appear as headers for some Decision Center reports, and an identifier that is meaningful in the real world sense is more useful than a strictly numeric identifier.

Figure 17–6 is an example of a database table Web Offers, that could be used as the external data source for a Dynamic Choice data source.

Figure 17–6 Example of an External Database Table



Name	Score	URL	Description	Id	Category	Image
Offer 1	60	textads/testcontent.html	This is offer 1	Offer 1	DynamicOffersCG	ads/S
Offer 2	50	textads/testcontent.html	This is offer 2	Offer 2	DynamicOffersCG	ads/K
Offer 3	40	textads/testcontent.html	This is offer 3	Offer 3	DynamicOffersCG	ads/S
Offer 4	30	textads/testcontent.html	This is offer 4	Offer 4	DynamicOffersCG	ads/K
Offer 5	20	textads/testcontent.html	This is offer 5	Offer 5	DynamicOffersCG	ads/S
Offer 6	10	textads/testcontent.html	This is offer 6	Offer 6	DynamicOffersCG	ads/S

The table illustrates the following features:

- The categorizing column is **Category**, and the common value in all the **Category** columns is **DynamicOffersCG**.
- You could select either **Name** or **ID** as the Dynamic Choice identifier column for the DynamicOffersCG category.

17.1.5 Overview of Setting up Dynamic Choices in Decision Studio

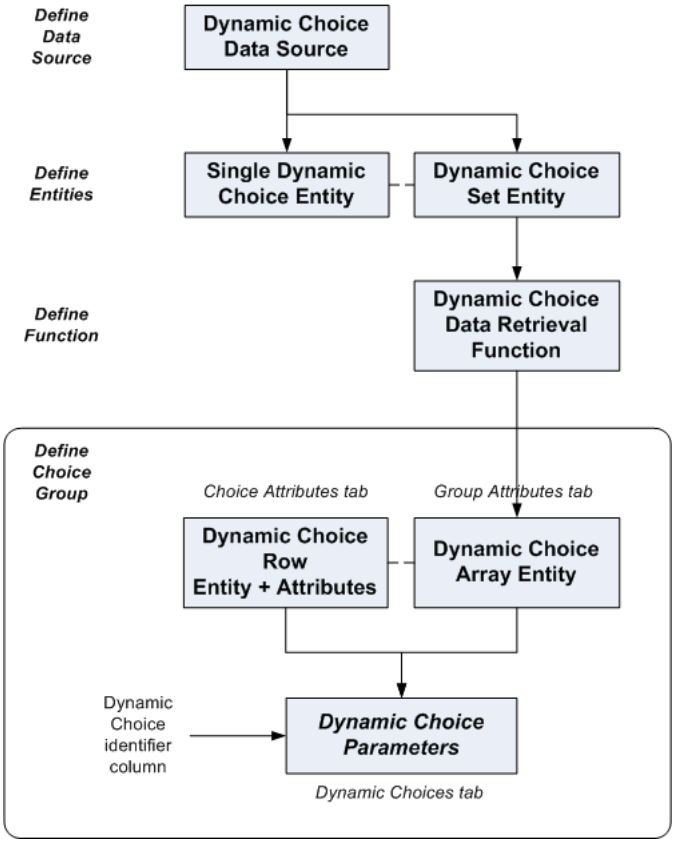
> **Note:** The diagrams and Decision Studio screen captures illustrating the setup process, which appear later in these sections on dynamic choices, are based on the DC_Demo Inline Service that is released with Oracle RTD.

The process of setting up of Dynamic Choices in Decision Studio consists of the following topics:

- Section 17.1.6, "Creating the Dynamic Choice Data Source"
- Section 17.1.7, "Creating the Single Dynamic Choice Entity"
- Section 17.1.8, "Creating the Dynamic Choice Set Entity"
- Section 17.1.9, "Creating the Dynamic Choice Data Retrieval Function"
- Section 17.1.10, "Considerations for Choice Group Design"
- Section 17.1.11, "Creating a Single Category Choice Group"
- Section 17.1.12, "Creating a Multi-Category Choice Group"

Figure 17–7 shows an overview of setting up a simple, single category Choice Group for Dynamic Choices. The elements in the diagram are referred to in the more detailed process descriptions that appear later in this chapter.

Figure 17–7 Overview of Setup Process for Single Category Dynamic Choices



17.1.6 Creating the Dynamic Choice Data Source

To create the Dynamic Choice Data Source:

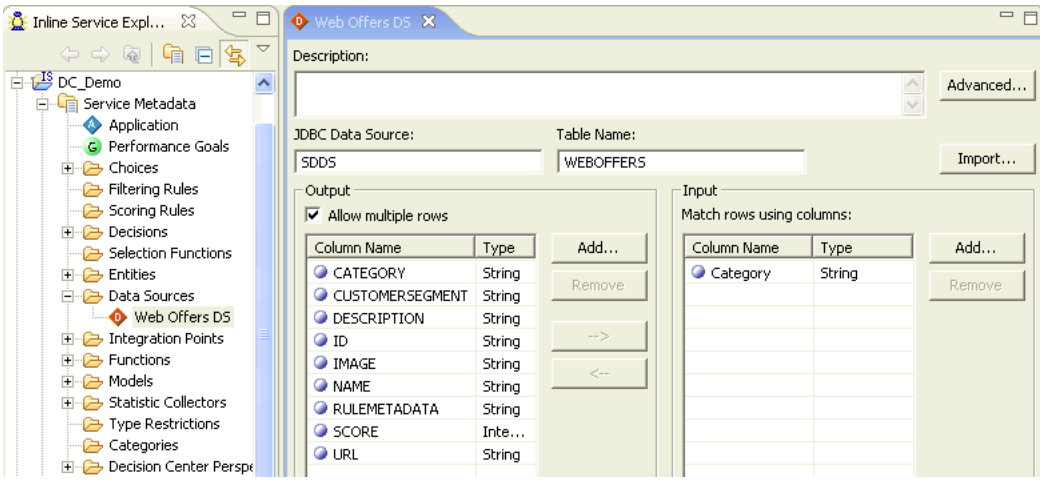
1. Create a new Data Source that maps to the table described in Section 17.1.4, "Prerequisite External Data Source for Dynamic Choices," using the **Import** button to point to the external data source.
2. In the Output column area, check **Allow multiple rows**, and select all the columns that you require for a Dynamic Choice.

   In the Input column area, select the column that contains the common value that categorizes and groups the Dynamic Choice rows.

   Note: You do not have to select the Dynamic Choice identifier column from among the Output columns at this stage.

Figure 17–8 shows how the Data Source **Web Offers DS** is set up from the table **SDDS.WEBOFFERS**, with **Category** as the Input identifier, and a number of other columns that represent attributes of the Dynamic Choice itself.

Figure 17–8 Defining the Web Offers DS Data Source



17.1.7 Creating the Single Dynamic Choice Entity

The Dynamic Choice data exists in the Data Source. You must create a Single Dynamic Choice Entity in Oracle RTD that consists of all the information associated with a particular category, but not the category itself.

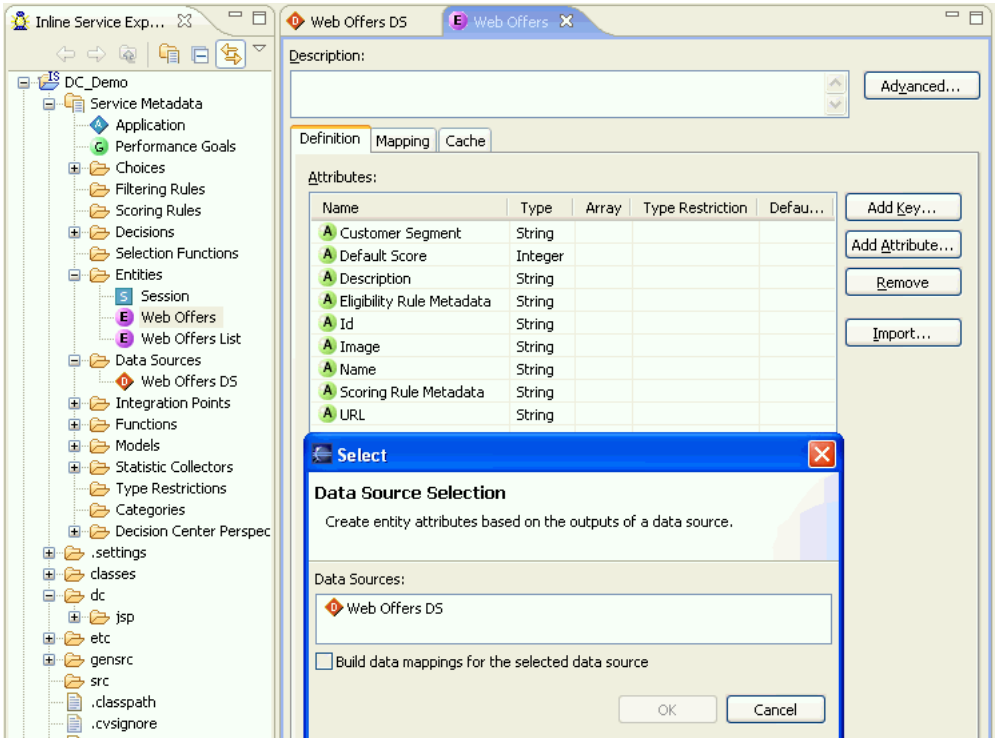
In terms of the Data Source that you created, the Entity attributes for the Single Dynamic Choice Entity are the Output attributes of the Data Source.

To create the Single Dynamic Choice Entity:

1. Create an Entity for the Dynamic Choice data, using the **Import** functionality to bring in all the Output columns from the Data Source described in Section 17.1.4, "Prerequisite External Data Source for Dynamic Choices."
2. When selecting the Data Source, be sure to uncheck the **Build data mappings for the selected data source** option found in the Select window that appears when you import.

Figure 17–9 shows the Definition tab for the setup of the Single Dynamic Choice Entity **Web Offers**. The attributes are the Output columns of the Data Source **Web Offers DS**.

Figure 17–9 Defining the Web Offers Entity



17.1.8 Creating the Dynamic Choice Set Entity

In addition to the Single Dynamic Choice Entity, you must create a Dynamic Choice Set Entity, that includes the following Attributes:

- A Key Attribute, which is the input, categorizing column of the Data Source that contains the Dynamic Choice data
- An array Attribute that stores the Single Dynamic Choice Entity data

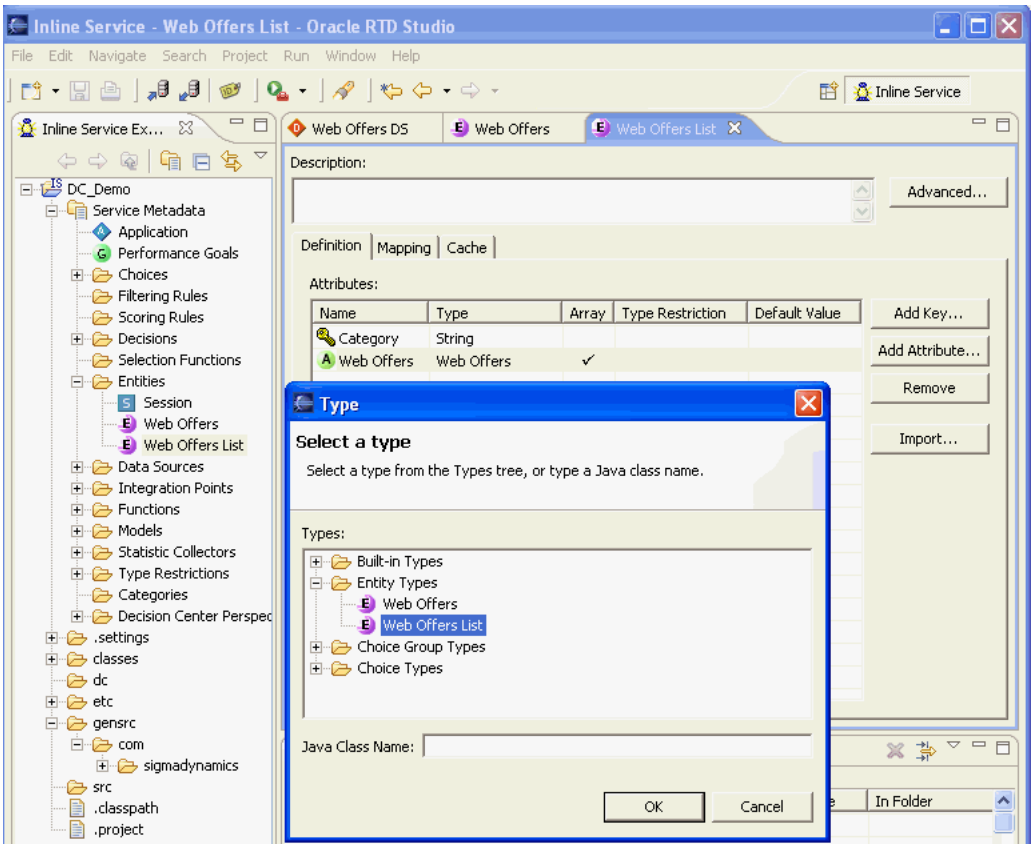
  This array Attribute must be of the same Entity type as the Entity that you created in Section 17.1.7, "Creating the Single Dynamic Choice Entity." This array is the container for all the Attributes of the data to be extracted from the Data Source required for the Dynamic Choice except for the categorizing Attribute.

To create the Dynamic Choice Set Entity:

1. Create an Entity in Oracle RTD.
2. For the key Attribute, click **Add Key**, and select the Dynamic Choice categorizing Attribute from the Data Source.
3. Create an Attribute whose type is the name of the Entity created in Section 17.1.7, "Creating the Single Dynamic Choice Entity."
4. Mark this entity-type Attribute as an **Array**.

   Figure 17–10 shows the Definition tab for the setup of the Dynamic Choice Set Entity **Web Offers List**. The Key Attribute is the Input column **Category** of the Data Source **Web Offers DS**, and the second Attribute is an array Attribute of type **Web Offers**.

Figure 17–10 Defining the Dynamic Choice Set Entity Web Offers List

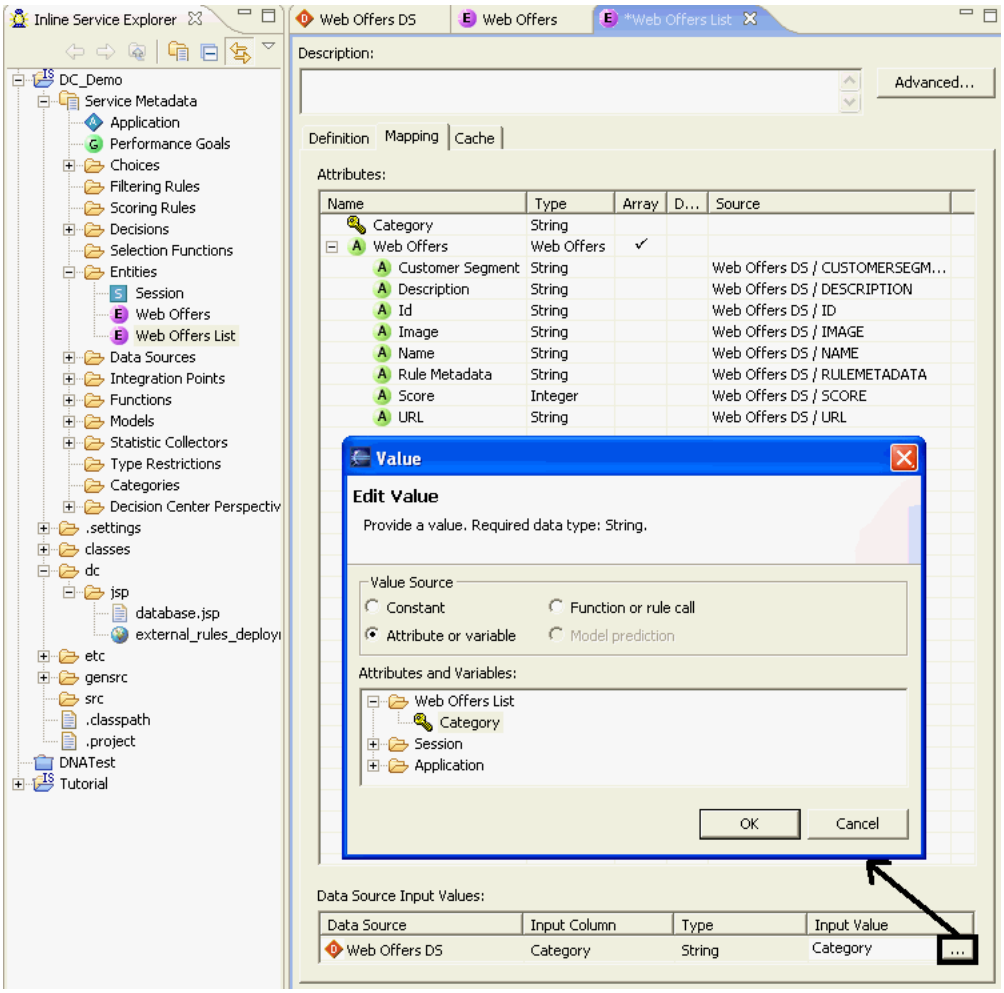


5. Click the Mapping tab, and map each Attribute within the entity-type Attribute to the appropriate column in the original Data Source.

6. In the Data Source Input Values region, for the Input Value of the Data Source, select the Dynamic Choice categorizing Attribute that you created in step 2.

   Figure 17–11 shows the Mapping tab for the setup of the Dynamic Choice Set Entity **Web Offers List**. Each of the Attributes within the array Attribute is mapped to the corresponding column in the **Web Offers DS** Data Source. In the Data Source Input Values region, the Attribute selected for the Input Value is the key Attribute **Category**.

Figure 17–11 Mapping the Web Offers Attributes in the Web Offers List Entity



7. Click the Cache tab.
8. Select the check box to **Enable caching for this entity type**.

> **Note:** It is important to enable caching on the Dynamic Choice Set Entity. Enabling caching will keep the Real-Time Decision Server from repeatedly pulling the Dynamic Choices from the data source with each new session.

17.1.9 Creating the Dynamic Choice Data Retrieval Function

To extract the Dynamic Choice data from the database, you must create a Function that will perform the data retrieval. This function will be called by the Choice Group that you will create in the steps that follow. The properties of the Function are as follows:

- The Function returns a value.
- The return value is of type Array.
- The Data Type of the array elements is the Single Dynamic Choice entity that you created previously

- The Function has a Parameter that is the same as the Data Source Input Value of the Dynamic Choice Set Entity that you created previously.
- The logic of the Function instantiates a new occurrence of the Dynamic Choice Set Entity, and uses the Parameter to retrieve the Dynamic Choice data into the array.

To create the Dynamic Choice Data Retrieval Function:

1. Create the Function, and select the **Return value** check box.
2. Select the **Array** option, to ensure that the return value is of type Array.
3. For the **Data Type**, select the name of the entity that you created in Section 17.1.7, "Creating the Single Dynamic Choice Entity."
4. In the Parameters area, add the **Name** and **Type** of the Key attribute that you created in step 2 of Section 17.1.8, "Creating the Dynamic Choice Set Entity."
5. In the **Logic** field, enter code similar to the following, adapting it as required for the names of your entities and attributes:

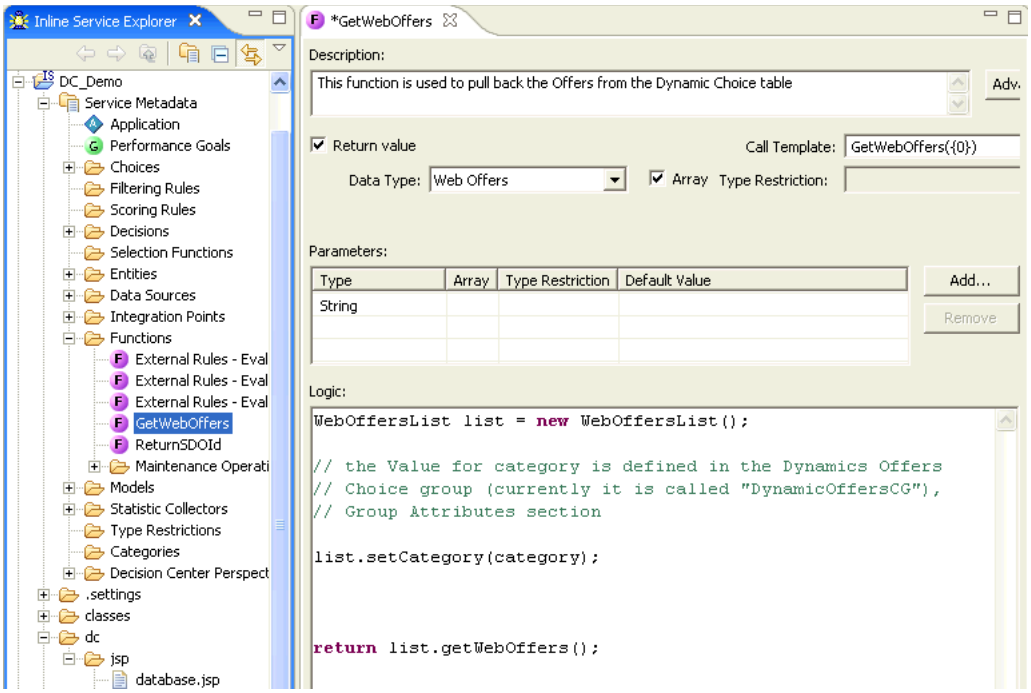
```
WebOffersList list = new WebOffersList();
list.setCategory(category);
return list.getWebOffers();
```

   where:

   - **WebOffersList** is the object name, with internal spaces deleted, for the Entity created in Section 17.1.8, "Creating the Dynamic Choice Set Entity."
   - **list.setCategory** references the Entity key that you created in Section 17.1.8, "Creating the Dynamic Choice Set Entity," step 2.
   - **getWebOffers()** refers to the Entity created in Section 17.1.7, "Creating the Single Dynamic Choice Entity," that is mapped inside the Dynamic Choice Set Entity.

     Figure 17–12 shows the definition of the **GetWebOffers** Function.

Figure 17–12 Defining the GetWebOffers Function



17.1.10 Considerations for Choice Group Design

Dynamic Choices enable application data administrators to control the choices that Oracle RTD recommends to the application. Unlike Static Choices, Dynamic Choices may be added, edited, and deleted in the application tables without requiring any changes in the interfacing Oracle RTD Inline Service.

If there is a requirement to have both type of Choice in a single Inline Service, Oracle recommends that Static Choices and Dynamic Choices are clearly separated in the designing of the Choice Groups. This section concentrates on the design of Choice Groups for Dynamic Choices.

You can design Dynamic Choices as follows:

- In a single Choice Group
- In completely separate Choice Groups - in effect, multiple independent single Choice Groups
- In a Choice Group hierarchy

There can be many factors that influence your design, for example:

- You may have a reporting requirement that a customer must have Choice Group reports for an explicit set of Dynamic Choices
- You may have decisioning requirements that some shared eligibility rules must apply for one set of Dynamic Choices as opposed to another set

This section outlines the high level design steps required for a single Choice Group and a Choice Group hierarchy.

Single Choice Group

Where all Dynamic Choices are required to be in one Choice Group, then the recommended design strategy is:

1. Design a single Choice Group.
2. Enter and select the required parameters in each of the following tabs for the Choice Group: Group Attributes tab, Choice Attributes tab, Dynamic Choices tab.

In Decision Studio, this Choice Group has no subgroups.

Choice Group Hierarchy

Your design factors may lead you to group Dynamic Choices within a Choice Group hierarchy. The following steps describe in outline form the setup of a two-level hierarchy:

1. For the top-level Choice Group, enter and select the required parameters in the Choice Attributes tab, but not the Group Attributes tab, nor the Dynamic Choices tab.
2. For each separate Dynamic Choice category, specify one lower-level Choice Group. In each of the lower-level Choice Groups, enter and select the required parameters in the Group Attributes tab and the Dynamic Choices tab, but not in the Choice Attributes tab.

> **Note:** You only need to fill in Dynamic Choices tab parameters in the lowest-level Choice Groups of a multi-level Choice Group hierarchy.

In Decision Studio, the lower-level Choice Groups have no subgroups.

17.1.11 Creating a Single Category Choice Group

To use Dynamic Choices, you must create one or more Choice Groups. Where the Dynamic Choices refer to data that belongs to one type or category, create a single category Choice Group.

> **Note:** In Decision Studio, when you create a Choice Group for Dynamic Choices, the individual Dynamic Choices do not appear in any of the Decision Studio windows.
>
> In Decision Center reports, you can see all the Dynamic Choices which satisfy the following conditions:
>
> - They have been returned by Decisions called by the front-end applications.
> - They have had RTD models updated for those Choices.

In Decision Studio, the Choice Group is configured to be able to extract the Choices dynamically at runtime through options that you set up in the following tabs:

- Group Attributes Tab
- Choice Attributes Tab
- Dynamic Choices Tab

These are the main tabs where you configure Dynamic Choices.

Note: You can also use the Choice Eligibility tab, to filter the Dynamic Choice data as it is extracted from the Data Source.

Eligibility rules created for a Dynamic Choice are shared across all Choices retrieved for the same Dynamic Choice Group.

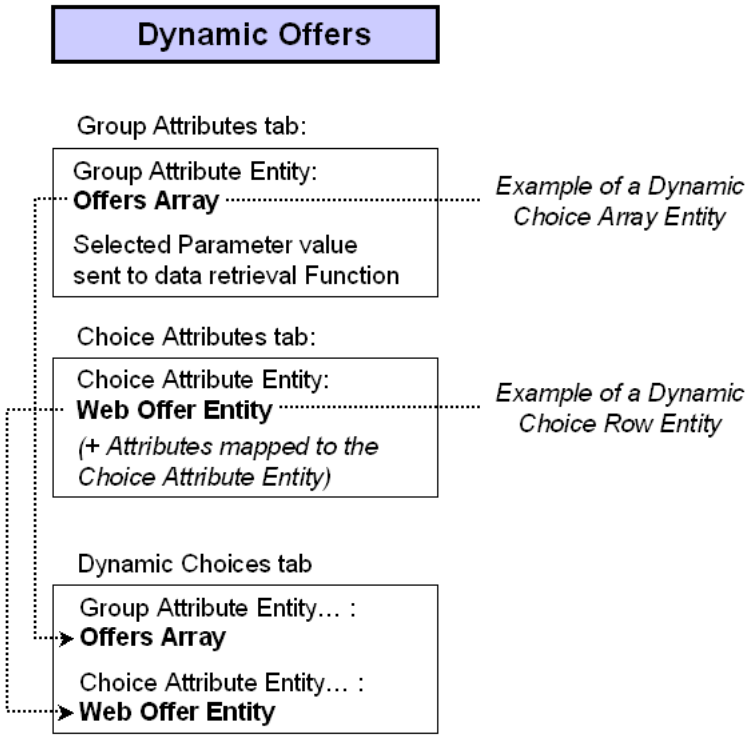
Figure 17–13 shows an example of the main elements required to set up a single category Choice Group, **Dynamic Offers**.

The Group Attribute setup indicates that all the data to be retrieved for the Dynamic Choices will be of one category only, and you must specify the exact category here.

The Choice Attribute setup describes the individual attributes that will be retrieved.

The Group and Choice Attributes are then referenced in the Dynamic Choices tab for this single category Choice Group.

Figure 17–13 Defining the Choice Group Dynamic Offers



17.1.11.1 Group Attributes Tab

In the Group Attributes tab, you specify an array Attribute of the same Entity type as that which you created in Section 17.1.7, "Creating the Single Dynamic Choice Entity." This Attribute is referred to as the Dynamic Choice Array Entity in Figure 17–7, which shows an overview of the single category Dynamic Choice setup process.

At this level, you also specify the Function that retrieves the Dynamic Choice data. You must choose a value for the Function parameter. This enables the function to retrieve just the Dynamic Choice data relevant for one particular real world type or category.

To create a Choice Group and specify the Group Attributes:

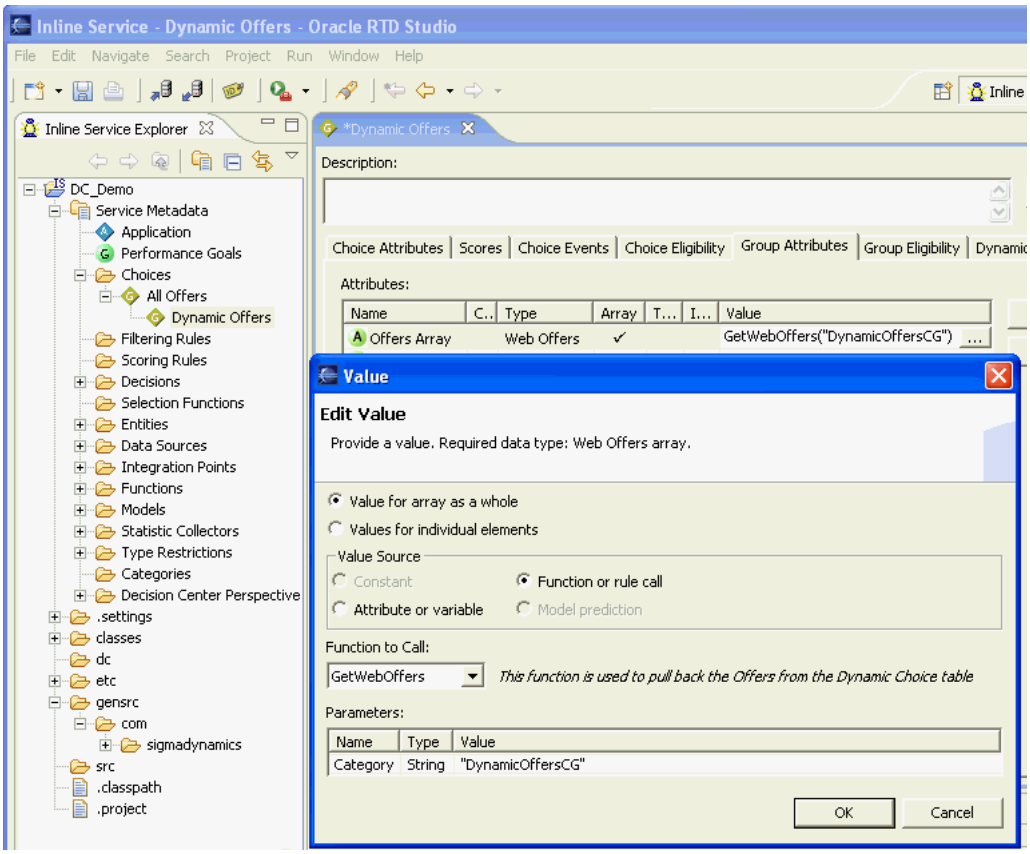
1. Create a Choice Group.
2. Click the Group Attributes tab.
3. Create a new entity-type Group Attribute (the Dynamic Choice Array entity), whose type is the name of the entity that you created in Section 17.1.7, "Creating the Single Dynamic Choice Entity."
4. Specify that this Attribute is an **Array**.
5. Click the right-hand end of the **Value** box to expose the ellipsis (...) button, then click the ellipsis button to open the Value window.
6. In the Value window, select the option **Value for array as a whole**.
7. For Value Source, select **Function or rule call**, then select the Function that you created in Section 17.1.9, "Creating the Dynamic Choice Data Retrieval Function."
8. In the Parameters area, choose the **Value** of the parameter that will retrieve the corresponding rows in the Data Source whose Input Attribute contains that value.

Note: This string Value is the exact value in the database that categorizes all the Dynamic Choice rows for a Choice Group.

For example, for a Choice Group set up for the Insurance_Proposals table as described in Section 17.1.1, "Simple Example of Dynamic Choices," the Value is **InsuranceProducts**.

Figure 17–14 shows the Group Attributes tab for the Choice Group **Dynamic Offers**. The Function to call is **GetWebOffers**. The Value in the Parameters area is the string **DynamicOffersCG**.

Figure 17–14 Defining the Group Attributes for the Choice Group Dynamic Offers



17.1.11.2 Choice Attributes Tab

In the Choice Attributes tab, you must:

- Specify an entity-type Attribute of the same type as the Entity that you created in Section 17.1.7, "Creating the Single Dynamic Choice Entity."

  This Attribute is referred to as the Dynamic Choice Row entity in Figure 17–7, which shows an overview of the single category Dynamic Choice setup process.

- For each of the component attributes within this Dynamic Choice Row Entity, create a separate Choice Attribute, which you must then map to the corresponding attribute within the Dynamic Choice Row entity that you just created.

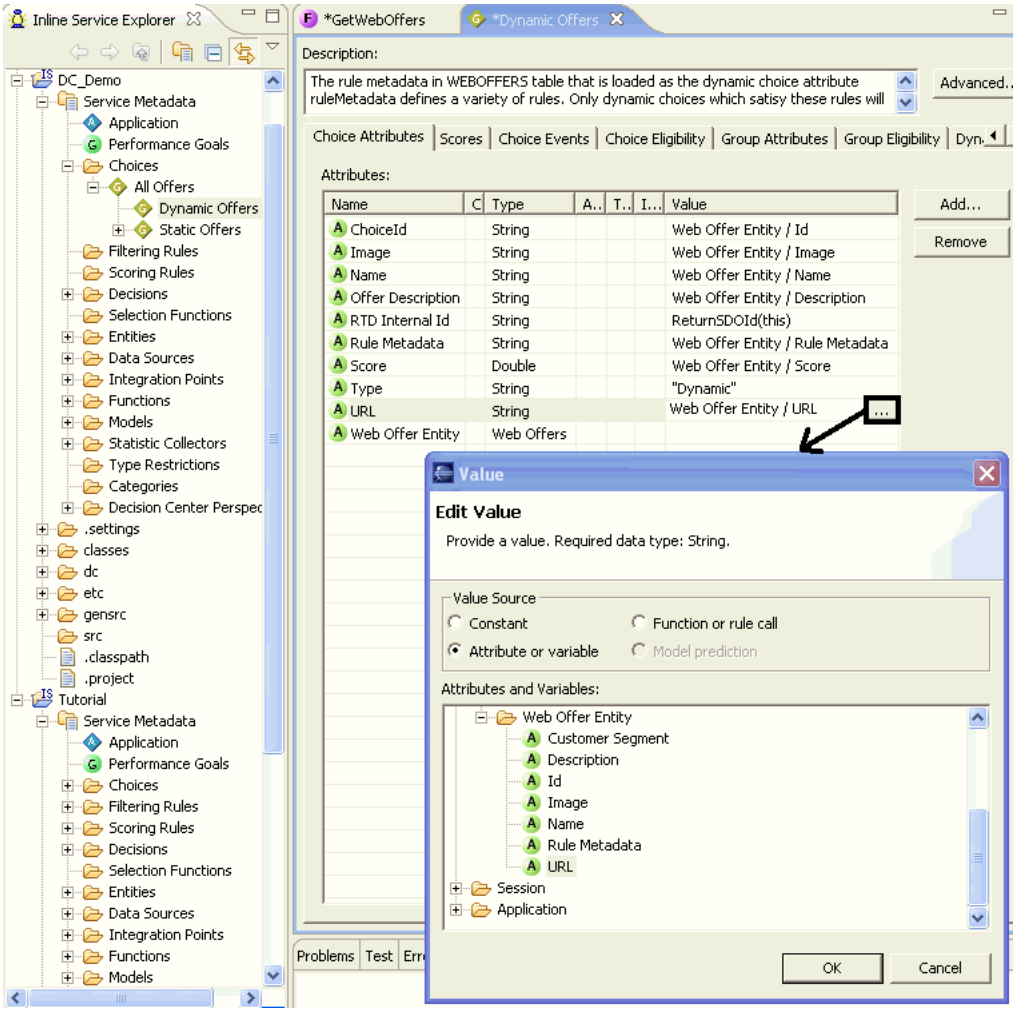
To specify the Choice Attributes of the Choice Group:

1. Click the Choice Attributes tab.
2. Create a new entity-type Attribute (the Dynamic Choice Row entity), whose type is the name of the entity that you created in Section 17.1.7, "Creating the Single Dynamic Choice Entity."
3. Ensure that the **Array** check box is not selected.
4. For each attribute of the new Dynamic Choice Row entity, create a corresponding Choice Attribute.
5. For each Choice Attribute created in the previous step, map its Value to the corresponding attribute within the Dynamic Choice Row entity that you created in step 2.

Figure 17–15 shows the Choice Attributes tab for the Choice Group **Dynamic Offers**. The Choice Attributes are the following:

- One Dynamic Choice Row Entity **Web Offer Entity**
- Several other Attributes, each of whose Values derives from the corresponding Attribute of the Dynamic Choice Row Entity **Web Offer Entity**

Figure 17–15 Defining the Choice Attributes for the Choice Group Dynamic Offers



17.1.11.3 Dynamic Choices Tab

In the Dynamic Choices tab, you provide the following information:

- You explicitly select this Choice Group to be for Dynamic Choices.
- You specify the Group and Choice Attributes that you set up in the corresponding Group Attributes and Choice Attributes tabs.
- You select the Attribute that identifies each Dynamic Choice.
- You describe how you wish the Dynamic Choices to appear in Decision Center reports. Because the number of Dynamic Choices could be considerable, you can choose to break up a potentially long list of Dynamic Choices into smaller units or "folders," and you indicate how you want the data grouped in the folders.

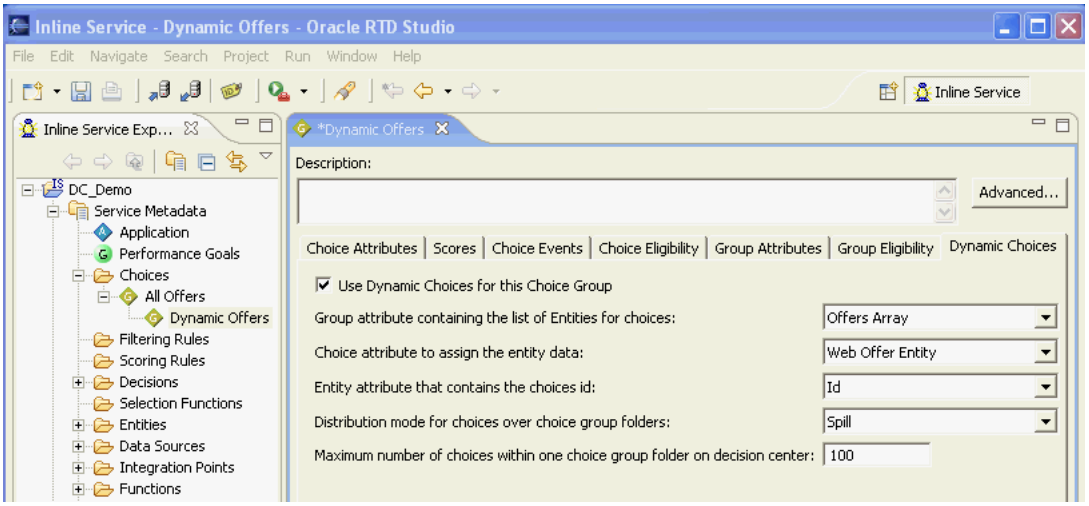
To specify the Dynamic Choice parameters:

1. Click the Dynamic Choices tab.
2. Select the check box option to **Use Dynamic Choices for this Choice Group**.
3. For the **Group attribute containing the list of Entities for choices**, select the Dynamic Choice Array attribute that you created in Section 17.1.11.1, "Group Attributes Tab."
4. For the **Choice attribute to assign the entity data**, select the Dynamic Choice Row attribute that you created in Section 17.1.11.2, "Choice Attributes Tab."
5. For the **Entity attribute that contains the choices id**, select the Attribute that serves as the unique identifier for each of the extracted Dynamic Choice rows.
6. For the **Distribution mode for choices over choice group folders**, select Spill or Even.

> **Note:** For more information about this parameter, and the parameter in the following step, see Section 17.1.13.4, "Distribution of Choices Across Decision Center Folders."

7. Select the **Maximum number of choices within one choice group folder on decision center**.

Figure 17–16 Defining the Dynamic Choice Parameters for the Choice Group



17.1.12 Creating a Multi-Category Choice Group

To use Dynamic Choices, you must create one or more Choice Groups. Where you want to be able to select different groups of data from the same data source, create a multi-category Choice Group. This section describes the standard way to set up a multi-category Choice Group.

Note: In Decision Studio, when you create a Choice Group for Dynamic Choices, the individual Dynamic Choices do not appear.

In Decision Center reports, you can see all the Dynamic Choices which satisfy the following conditions:

- They have been returned by Decisions called by the front-end applications.
- They have had RTD models updated for those Choices.

In Decision Studio, a Choice Group is configured to be able to extract the Choices dynamically at run time through options that you set up in the following tabs:

- Group Attributes tab
- Choice Attributes tab
- Dynamic Choices tab

These are the main tabs where you configure Dynamic Choices.

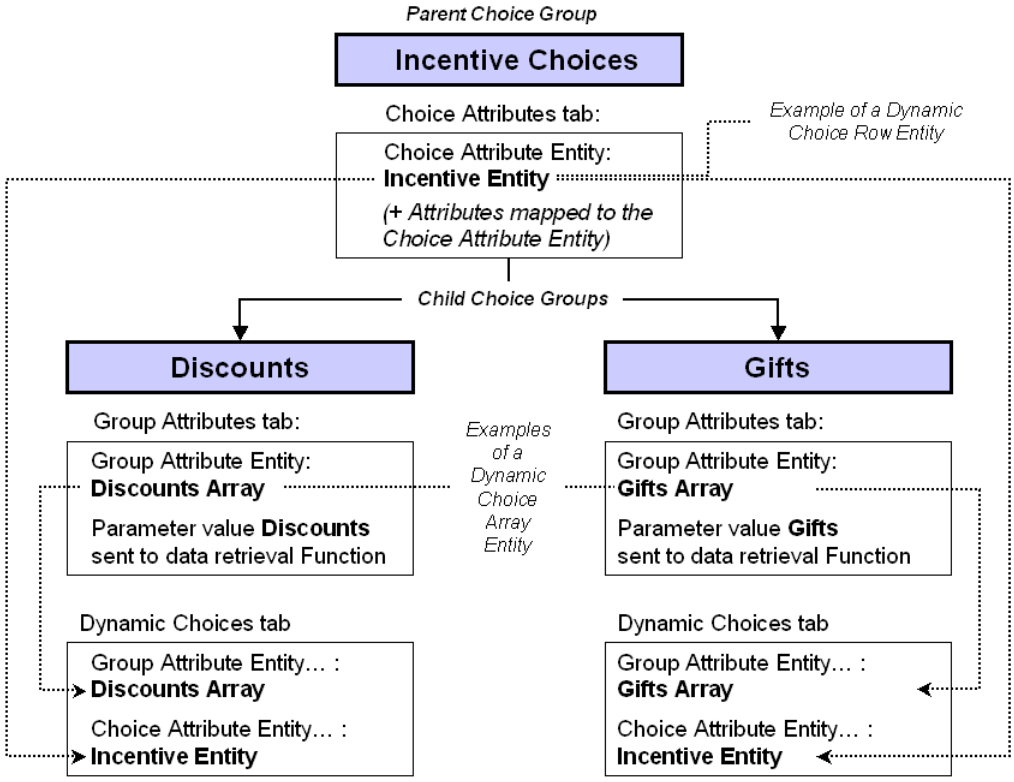
Note: You can also use the Choice Eligibility tab, to filter the Dynamic Choice data as it is extracted from the data source.

Eligibility rules created for a Dynamic Choice are shared across all Choices retrieved for the same Dynamic Choice Group.

To allow for multiple Dynamic Choice categories, you must create a hierarchy of Choice Groups, and set up the Choice Group elements at different levels.

Figure 17–17 shows an example of the main elements required to set up a two-category Choice Group, **Incentive Choices**.

Figure 17–17 Example of Defining a Choice Group Hierarchy



The Choice Group **Incentive Choices** is the parent Choice Group, with two child Choice Groups, **Discounts** and **Gifts**.

You specify the Choice Attributes at the top level, in the parent Choice Group. These Choice Attributes are then inherited by the two child Choice Groups.

Note: In the parent Choice Group, or in any higher level Groups of a multi-level Choice Group hierarchy, you do not enter or select any values in the Dynamic Choices tab. Dynamic Choice parameters are only specified in the lowest level Group of any Choice Group hierarchy.

Each child Choice Group enables a different category set of data to be retrieved, through the Group Attributes setup. The Group and Choice Attributes are then referenced in the Dynamic Choices tab for both of the child Choice Groups.

To compare this process with the equivalent single category Choice Group setup, see Figure 17–7.

17.1.12.1 Choice Attributes Tab in the Parent Choice Group

In the Choice Attributes tab, you specify an entity-type Choice Attribute of the same type as that which you created in Section 17.1.7, "Creating the Single Dynamic Choice Entity."

This Choice Attribute is also known as the Dynamic Choice Row Entity, as in the equivalent single category Dynamic Choice setup process shown in Figure 17–7.

For each of the attributes within this Dynamic Choice Row entity, create a separate Choice Attribute, which you must map to the corresponding attribute in the Dynamic Choice Row entity that you just created.

To create the Parent Choice Group and Choice Attributes:

1. Create the parent Choice Group.
2. Click the Choice Attributes tab.
3. Create a new entity-type Choice Attribute, whose type is the name of the entity that you created in Section 17.1.7, "Creating the Single Dynamic Choice Entity."
4. Ensure that you do not specify that this is an **Array**.
5. For each of the attributes of the new entity-type Choice Attribute, create a corresponding Choice Attribute.
6. For each of the attributes created in the previous step, map its Value to the corresponding attribute within the Choice Attribute that you created in step 2.

17.1.12.2 Group Attributes Tab in the Child Choice Groups

For each child Choice Group, in the Group Attributes tab, you specify an entity-type array Attribute of the same type as that which you created in Section 17.1.7, "Creating the Single Dynamic Choice Entity."

This Group Attribute is also known as the Dynamic Choice Array Entity, as in the equivalent single category Dynamic Choice setup process shown in Figure 17–7.

At this level, you also specify the function that retrieves the Dynamic Choice data. You must choose a value for the Function parameter. This enables the Function to retrieve just the Dynamic Choice data relevant for one particular real world type or category.

First, you need to create the child Choice Groups under the previously created parent Choice Group, then enter the required elements in the Group Attributes tab.

To create the Child Choice Groups and Group Attributes:

1. Create the first child Choice Group.
2. Create extra child Choice Groups as required, one for each separate Dynamic Choice category.

   Within each child Choice Group, you must now set up the required elements and parameters in the Group Attributes tab and the Dynamic Choices tab.

   The steps following in this section describe the actions required in the Group Attributes tab for each child Choice Group. Section 17.1.12.3 describes the actions required in the Dynamic Choices tab for each child Choice Group.
3. Click the Group Attributes tab of the child Choice Group.
4. Create a new entity-type Group Attribute, whose type is the name of the Entity that you created in Section 17.1.7, "Creating the Single Dynamic Choice Entity."
5. Specify that this Attribute is an **Array**.
6. Click the right-hand end of the **Value** box to expose the ellipsis (...) button, then click the ellipsis button to open the Value window.
7. In the Value window, select the option **Value for array as a whole**.
8. For Value Source, select **Function or rule call**, then select the Function that you created in Section 17.1.9, "Creating the Dynamic Choice Data Retrieval Function."

9. In the Parameters area, choose the **Value** of the parameter that will retrieve the corresponding rows in the Data Source whose Input attribute contains that value.

17.1.12.3 Dynamic Choices Tab in the Child Choice Groups

For each child Choice Group, in the Dynamic Choices tab, you must provide the following information:

- You explicitly select this Choice Group to be a Choice Group for Dynamic Choices.
- You specify the Group and Choice Attributes that you set up in the corresponding Group Attributes and Choice Attributes tabs.
- You select the Attribute that identifies each Dynamic Choice.
- You describe how you wish the Dynamic Choices to appear in Decision Center reports. Because the number of Dynamic Choices could be considerable, you can choose to break up a potentially long list of Dynamic Choices into smaller units or "folders," and you indicate how you want the data grouped in the folders.

To specify the Dynamic Choice parameters:

1. Click the Dynamic Choices tab.
2. Select the check box option to **Use Dynamic Choices for this Choice Group**.
3. For the **Group attribute containing the list of Entities for choices**, select the attribute that you created in Section 17.1.12.2, "Group Attributes Tab in the Child Choice Groups."
4. For the **Choice attribute to assign the entity data** select the attribute that you created in Section 17.1.12.1, "Choice Attributes Tab in the Parent Choice Group."
5. For the **Entity attribute that contains the choices id**, select the Attribute that serves as the unique identifier for each of the extracted Dynamic Choice rows.
6. For the **Distribution mode for choices over choice group folders**, select Spill or Even.

> **Note:** For more information about this parameter, and the parameter in the following step, see Section 17.1.13.4, "Distribution of Choices Across Decision Center Folders."

7. Select the **Maximum number of choices within one choice group folder on decision center**.

17.1.13 Dynamic Choice Reporting Overview

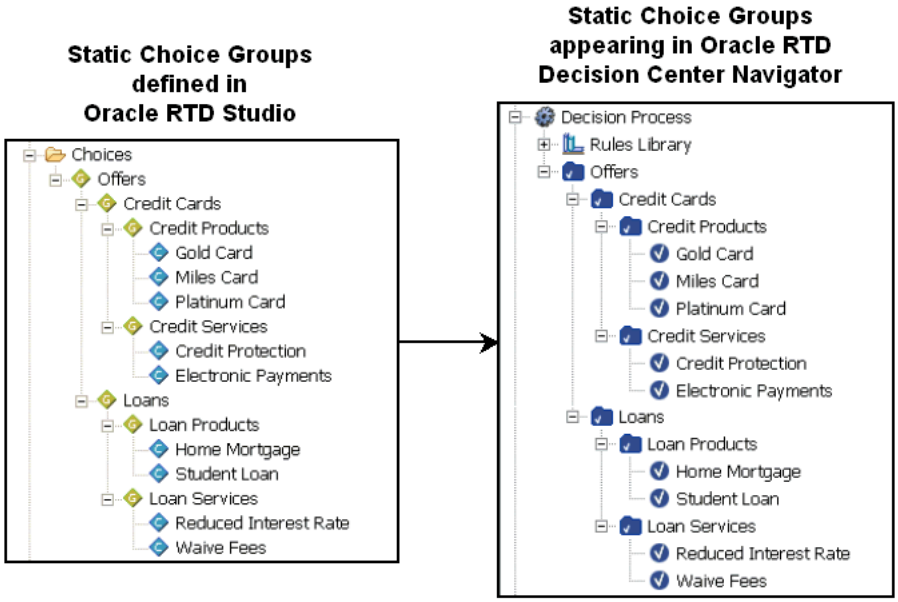
This section consists of the following topics:

- Section 17.1.13.1, "Applications with Static Choices Only"
- Section 17.1.13.2, "Dynamic Choice Visibility"
- Section 17.1.13.3, "System-Created Range Folders"
- Section 17.1.13.4, "Distribution of Choices Across Decision Center Folders"
- Section 17.1.13.5, "Example of a Decision Center Report with Dynamic Choices"

17.1.13.1 Applications with Static Choices Only

If your application has been configured to use only Static Choices, there is no impact on Decision Center reporting. The Choice Groups, subgroups, and Static Choices that you defined in Decision Studio will appear in the same hierarchical layout in the Decision Center Navigator, as shown in the example in Figure 17–18.

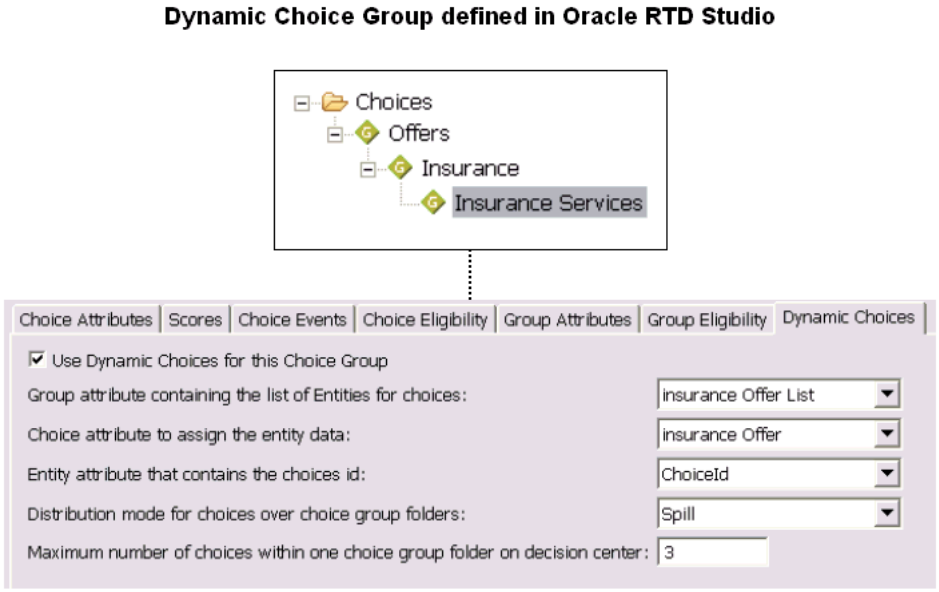
Figure 17–18 Example of Definition and Reporting with Static Choices Only



17.1.13.2 Dynamic Choice Visibility

Dynamic Choices, by their very nature, cannot be predefined in Decision Studio. A Choice Group can be configured to hold dynamically-extracted external data, from which Dynamic Choices can be recommended. Figure 17–19 shows an example of a Choice Group set up to display Dynamic Choices for insurance services.

Figure 17–19 Example of Dynamic Choice Group Definition



In Decision Center, only those Dynamic Choices that have actually been recommended and added to model learning data appear in the Decision Center Navigator, and have Performance and Analysis reports.

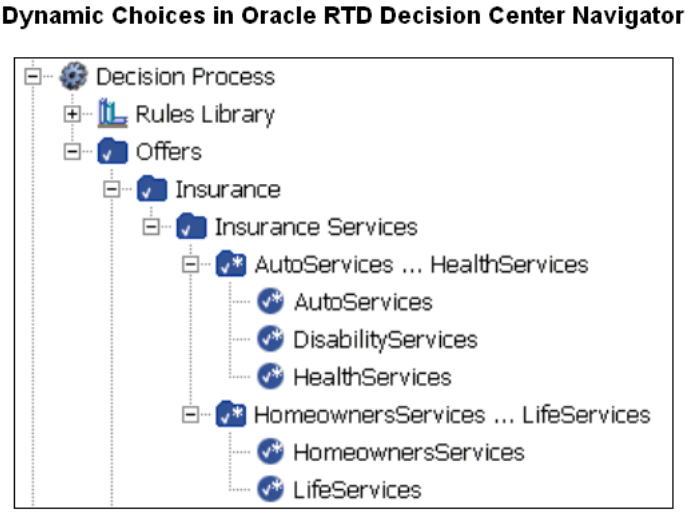
The other factor that influences how the Dynamic Choices appear in the Decision Center is the parameter **Maximum number of choices within one choice group folder on decision center**, which you specify when you define the Dynamic Choice Group. If the number of choices exceeds this maximum, the choices appear under system-created range folders, otherwise they appear directly under the Choice Group name.

For more information on range folders, see Section 17.1.13.3, "System-Created Range Folders."

The example Decision Support Navigator menu in Figure 17–20 shows the following:

- Five Dynamic Choices were recommended and added to model learning data.
- The maximum number of choices per choice group is 3.
- Each Dynamic Choice appears under one of the two system-created folder names.

Figure 17–20 Example of Dynamic Choice Layout in Decision Center



17.1.13.3 System-Created Range Folders

The name of each system-created folder is made up of the names of the first and last Choices in the folder, with the string "..." separating the two Choices. System-created folders are also known as *range folders*.

> **Note:** The Choices within a range folder can be a mixture of Static and Dynamic Choices. Both components of the range folder name can therefore be either a Static or a Dynamic Choice.
>
> In general, Oracle recommends that applications keep Static and Dynamic Choices in separate Choice Groups or separate Choice Group hierarchies.

If the total number of (Static choices + Dynamic Choices recommended and added to model learning data) exceeds the maximum defined for the Choice Group folder, the choices appear in system-created "groups" or subfolders, otherwise they appear directly under the Choice Group name.

17.1.13.4 Distribution of Choices Across Decision Center Folders

When configuring a Choice Group for Dynamic Choices in Decision Studio, there are two parameters that affect how choices appear in Decision Center.

Both parameters are in the Dynamic Choices tab, and they are only enabled if the Choice Group is selected to be used for Dynamic Choices. The parameters are:

- **Distribution mode for choices over choice group folders**
- **Maximum number of choices within one choice group folder on decision center**

For simplicity, these parameters are referred to as **Distribution mode** and **Maximum number of choices** in this section.

The **Maximum number of choices** parameter determines how choices appear in the Decision Center directly under the Choice Group name or under a system-created

range folder. For more information on range folders, see Section 17.1.13.3, "System-Created Range Folders."

> **Note:** In Decision Center reports, range folders are not dedicated to Static or Dynamic Choices, that is, both Static and Dynamic Choices may appear together in the same range folder.

The **Maximum number of choices** parameter limits the number of Choices, regardless of whether they are Static or Dynamic, in each range folder.

The **Distribution mode** parameter specifies how the range folders are populated:

- In **Spill** mode, each range folder is filled up to the maximum, and the final range folder typically has less values than the maximum.
- In **Even** mode, the aim is to distribute the Choices evenly across the range folders.

For example, if there is a total of 106 Static or Dynamic Choices to display in the Decision Center, and the maximum per range folder is 25:

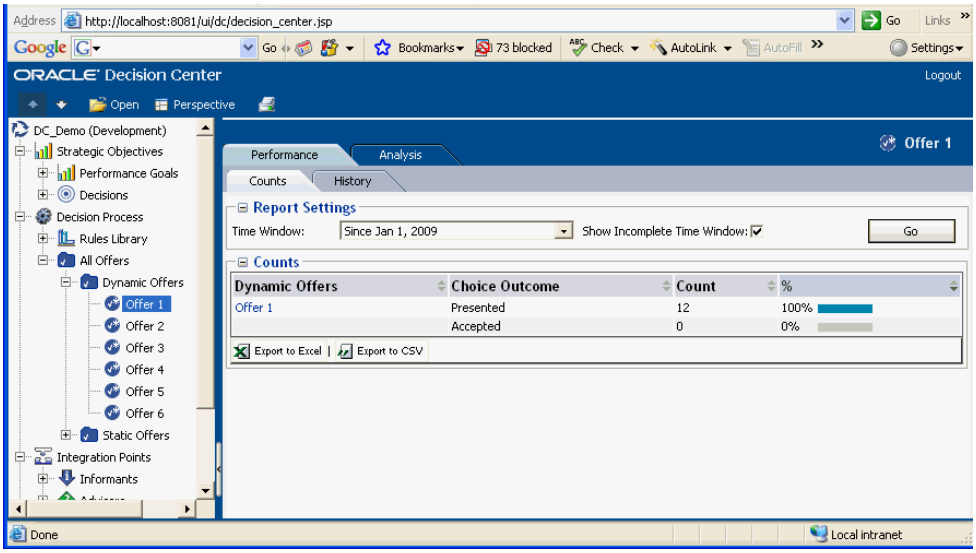
- In **Spill** Mode, the distribution across the range folders is 25,25,25,25,6.
- In **Even** Mode, the distribution across the range folders is 22,21,21,21,21.

17.1.13.5 Example of a Decision Center Report with Dynamic Choices

Decision Center can be used to view reports for each Dynamic Choice defined in a content database, which were actually recommended and added to model learning data. This is done by logging into a Decision Center Inline Service and opening the Decision Process section in the Decision Center navigator.

From here, any defined Dynamic Choice groups will be listed and will contain all dynamic offers defined in database tables for each Dynamic Choice group, that were recommended and added to model learning data. Choices in the database tables that were not added to model learning data do *not* appear in Decision Center reports.

The following is an image of a Decision Center report, with the navigator tree showing the DC_Demo Dynamic Choices:



17.2 External Rules

External rules enable end users running an application integrated with Oracle RTD to influence the Oracle RTD decision logic itself at run-time without the need to recompile the Inline Service.

A typical use would be where specific rules, such as choice eligibility rules and choice group eligibility rules, are attached to dynamic choices, and need to be modified at run-time without Inline Service recompilation.

External rules can also be attached to static choices.

This section contains the following topics:

17.2.1 Introduction to External Rules

The main components of the External Rules feature are:

- External Rules Editor

  Oracle RTD provides an embeddable Rule Editor widget that can be plugged in to customer front-end Web based applications, for example through an HTML iframe.

- External Rules Framework

  The external rules framework consists of:

  – External Rule Functions

    Users specify rule evaluation functions in Decision Studio that can be called to evaluate external rules. There are three functions: one to evaluate choice rules, one to evaluate choice group rules, and one to evaluate filtering rules.

  – External Rule Caching

    External rules caching is provided to improve the performance of external rule evaluation.

    If a newer version of a cached rule is submitted for evaluation, Oracle RTD does not execute the stale version. To avoid the whole request timing out in the event of a rule cache miss, Oracle RTD provides the Inline Service developer a mechanism for specifying a default response to return immediately.

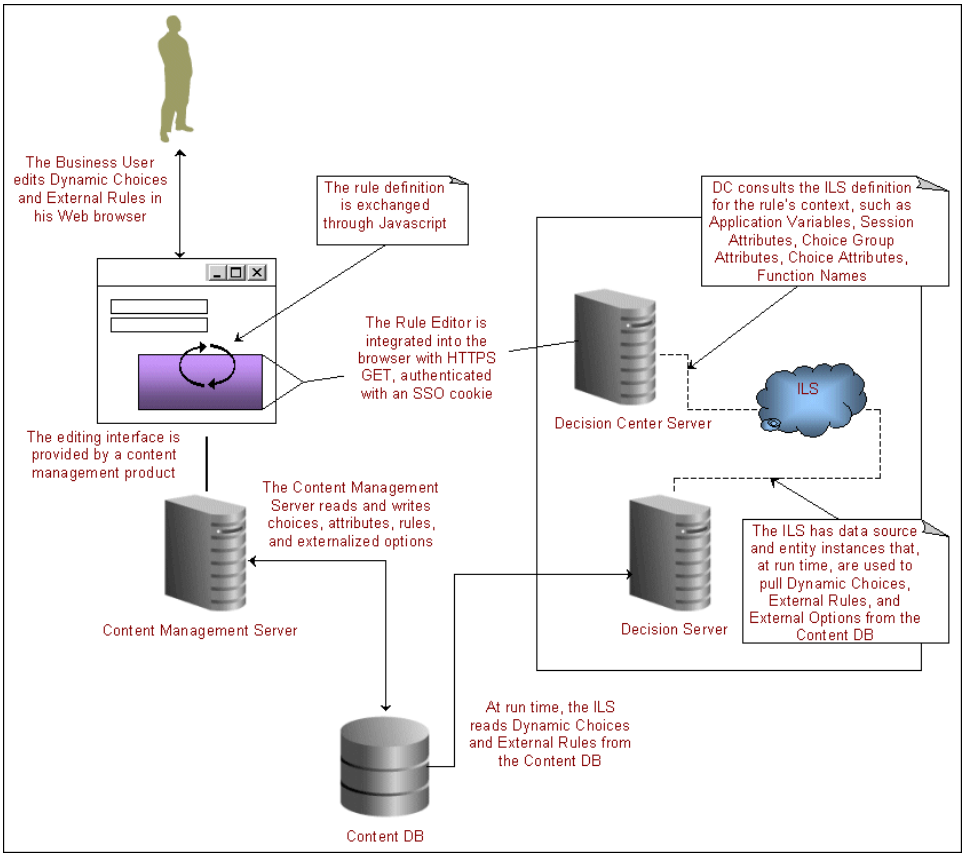
  – External Rule APIs

    Oracle RTD provides a set of Java APIs related to external rules, which can be used in Decision Studio Java functions and logic sections.

Overview of the External Rules Process

Figure 17–21 shows the external rule process flow during editing and rule evaluation.

Figure 17–21 External Rules Process Flow



The external rules are stored in metadata form in an external content repository, which can be any external data system, but is typically a database. The content management server controls read and write access to the rules stored in the external content repository.

Business users edit rules through an Oracle RTD Rule Editor embedded in a third party external interface provided by a content management product.

The external interface dynamically sets the context in which the rule needs to be edited in the embedded Rule Editor. For example a rule can be attached to a specific group's choice, a choice group or a filtering rule.

In the Rule Editor, the business user creates and edits rules, which can reference any of the objects defined in the Inline Service, such as any of the dynamic choices, functions, and session attributes.

After the user has finished editing the rule in the Rule Editor, the rule metadata is passed to the external interface, which saves the rule metadata to the external content repository.

At run time, the Inline Service accesses the edited external rule from the external content repository.

Example of External Interface in DC_Demo Inline Service Helper File

To serve as a starting-point for a third party external interface, Oracle RTD provides an External Rules Deployment Helper HTML file with the DC_Demo Inline Service.

For more information about how to set up and use this helper, see Section 17.3, "Example of End to End Development Using Dynamic Choices and External Rules."

17.2.2 External Rule Editor

Oracle RTD provides a browser embeddable user interface for editing external rules. This Rule Editor widget contains functionality comparable to the rule editors contained in Decision Studio.

A third party user interface together with an embedded Oracle RTD Rule Editor must be able to perform the following actions:

- Load external rule metadata into the embedded Rule Editor
- Edit loaded external rule metadata in the embedded Rule Editor
- Export the new rule metadata with a unique ID and timestamp for the rule loaded into the embedded Rule Editor
- Dynamically sets the context in which the rule needs to be edited. For example a rule can be attached to a specific group's choice, a choice group or a filtering rule.
- Set a user-defined callback Javascript function that will be called after every action submitted by the embedded Rule Editor
- Provide Javascript methods to determine whether an edited external rule is valid or has been modified

17.2.3 External Rule Framework

The external rule framework consists of three rule evaluation functions, external rule caching, and a set of Java APIs.

This section consists of the following topics:

- Section 17.2.3.1, "External Rule Evaluation Functions"
- Section 17.2.3.2, "External Rule Caching"
- Section 17.2.3.3, "External Rule APIs"
- Section 17.2.3.4, "External Rule Error Handling and Logging"

17.2.3.1 External Rule Evaluation Functions

Decision Studio provides three rule evaluation functions that can be used to evaluate external rule metadata. Each function evaluates the passed-in external rule metadata against either a choice, a choice group, or a filtering rule. A boolean value is returned from each function that specifies whether a rule evaluated successfully. The three functions are:

- **External Rules - Evaluate Choice Eligibility Rule**
- **External Rules - Evaluate Choice Group Eligibility Rule**
- **External Rules - Evaluate Filtering Rule**

Table 17–1 shows the parameters for these functions.

Note: For the eligibility rule functions, one of the parameters sets the context for rule evaluation. For example, the parameter **choice** in the function External Rules - Evaluate Choice Eligibility Rule specifies the particular choice name where the external rule will be evaluated.

Table 17–1 External Rule Function Parameters

Function	Parameter	Description
External Rules - Evaluate Choice Eligibility Rule	Rule Metadata	Attribute containing metadata form of the external rule
	choice	Choice where the external rule will be evaluated.
	Return value	Status if rule is invalid
External Rules - Evaluate Choice Group Eligibility Rule	Rule Metadata	Attribute containing metadata form of the external rule
	choice group	Choice group where the external rule will be evaluated.
	Return value	Status if rule is invalid
External Rules - Evaluate Filtering Rule	Rule Metadata	Attribute containing metadata form of the external rule
	Return value	Status if rule is invalid

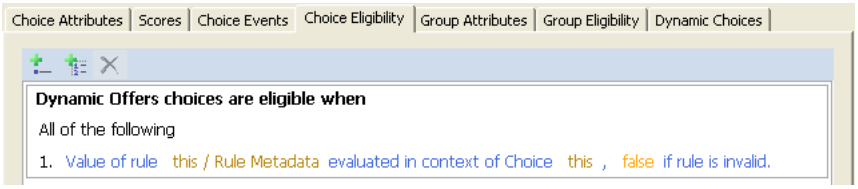
The call templates for both **External Rules - Evaluate Choice Eligibility Rule** and **External Rules - Evaluate Choice Group Eligibility Rule** are released as:

- Value of rule **{0}** evaluated in context of Choice **{1}**, **{2}** if rule is invalid

The call template for **External Rules - Evaluate Filtering Rule** is released as:

- Value of rule **{0}**, **{1}** if rule is invalid

These templates appear in the Rule Editor when the function is selected for the rule. The following image from the DC_Demo Inline Service shows the call template and selected parameters in the Choice Eligibility tab for the choice group Dynamic Offers.



Blocking Evaluation Option

Each function allows for the setting of evaluation options.

One of the options controls blocking and non-blocking evaluation. Setting the blocking evaluation option forces the rule evaluator caller to wait for the Real-Time Decision Server to return with the evaluation result. Non-blocking evaluation returns a default value back to the rule evaluation caller.

By default, each of the external rule evaluation functions will evaluate the passed-in external rule metadata in a non-blocking manner. Decision Studio users can change this behavior by editing the Java code of the selected function to evaluate rules with the blocking option set.

Modifying the External Rules Functions

The External Rules functions can be altered to suit individual Inline Services. One possible change is to alter the blocking behavior of the rule evaluation. Each function evaluates the passed-in rule metadata in a non-blocking manner by default. The API

that controls blocking behavior, default return value, and whether exceptions are thrown is as follows:

```
public interface EvaluationOptions  {
     public static EvaluationOptions getEvaluationOptions(
          boolean defaultReturnValue,
          boolean blockEvaluationUntilCached,
          boolean propagateExceptions);
}
```

The External Rules functions are all similar to one another, in that each function creates a rule definition, obtains a rule evaluator and rule cache, defines evaluation options, and then evaluates the rule. The difference between the functions is in their scope: the functions evaluate as a filtering rule or for a choice or choice group. The following example shows a choice evaluation function in more detail.

```
//compile, cache and evaluate the eligibility rule and return a boolean
if (ruleMetadata == null || ruleMetadata.trim().equals(""))
        return true;
RuleDefinition def = new RuleDefinitionImpl(ruleMetadata);
RuleEvaluator evaluator = Application.getRuleEvaluator();
RuleCache cache = Application.getRuleCache();

// public static EvaluationOptions getEvaluationOptions(
//                 boolean defaultReturnValue,
//                 boolean blockEvaluationUntilCached,
//                 boolean propagateExceptions)
// boolean defaultReturnValue: Return this value when rule evaluation fails
//                  with an exception or while the rule is being compiled
//                  during non-blocking evaluation
// boolean blockEvaluationUntilCached: Wait for the rule to be compiled before
//                  returning a value. (May cause integration point timeout)
// boolean propagateExceptions: Set to true if ILS developer decides to
//                  handle ValidationException and EvalutionException thrown by
//                  RuleEvaluator.evaluate() instead of returning defaultReturnVal

EvaluationOptions opts = EvaluationOptions.getEvaluationOptions(
    returnValue, false, false);
/*
The evaluate method attempts to retrieve the compiled bytecode for the rule
definition from the cache. If the bytecode for the rule is found in the cache, the
rule is evaluated and the resulting boolean is returned. Otherwise, the rule is
queued for compilation. Until the rule is compiled and cached, evaluate function
behaves as specified by the EvaluationOptions.
*/
return evaluator.evaluate(choice, def, cache, opts);
// parameters: ruleMetadata, choice
// return: boolean
```

You can change the **blockEvaluationUntilCached** and **propagateExceptions** parameters on the **getEvaluationOptions** call in any or all of the External Rules functions.

> **Note:** Each External Rules function change operates at the Inline Service level.

17.2.3.2 External Rule Caching

The Real-Time Decision Server includes an external rule cache in order to improve rule evaluation performance. Each Inline Service application will maintain its own rule

cache and each application rule cache will be replicated on each Real-Time Decision Server in a cluster. The external rules caching functionality provides the following additional features:

- Rule Cache Maintenance Operations

  Decision Studio provides maintenance operation functions that can be used to determine rule cache size and to clear the cache. These functions are:

  – **External Rules - Clear Cache**

  – **External Rules - Get Cache Size**

  – **External Rules - Remove Inactive Cached Rules**

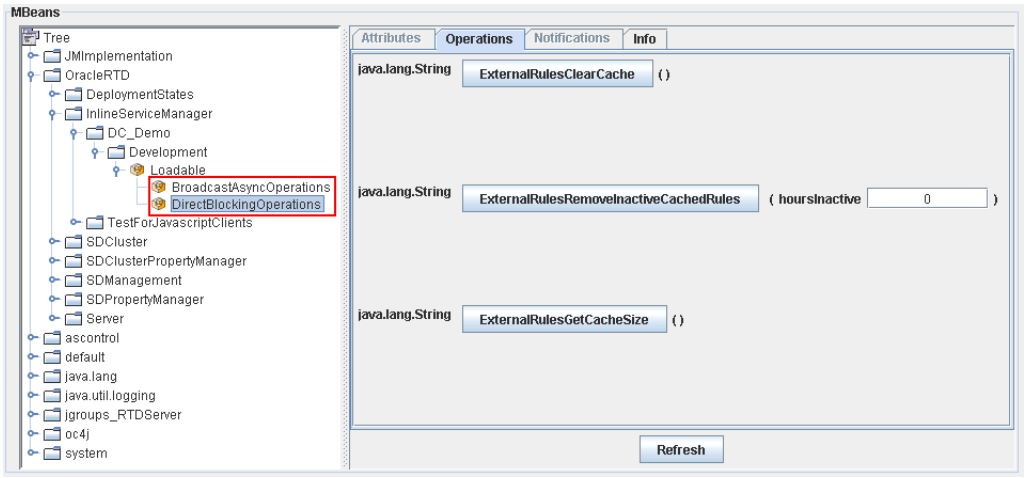
  These operations can be triggered externally by an MBean client such as JConsole to clear the cache of an Inline Service deployed on a Real-Time Decision Server. Each operation uses the external rule caching Java APIs for clearing an Inline Service rule cache and obtaining the current size in bytes of the rule cache.

- Non-Blocking Rule Evaluation

  This feature guarantees that the evaluation of a rule will be non-blocking if the rule is not found in the cache. During the very short time that it takes to compile a single rule, Oracle RTD returns a default true/false value for an uncached rule while the rule is being compiled in the background.

JMX Clients (JConsole)

External rule caching maintenance operations are accessible as MBean operations in JMX clients such as Sun's JConsole. These maintenance operations are created for each deployed Inline Service and can be found in the MBean OracleRTD > InlineServiceManager tree path for the Inline Service. The following image shows the MBean operations for DC_Demo:



17.2.3.3 External Rule APIs

The external rules framework provides a set of Java APIs introduced with the external rule caching feature. The APIs are provided by the following Java interfaces and available for use in Decision Studio Java functions and logic sections:

- **Rule** - A rule instance returned.
- **RuleDefinition** - A rule definition created by the user and passed in to an application rule evaluator.

- **RuleCache** - A rule cache maintained by a deployed Inline Service and exposed through the Inline Service application interface.
- **RuleEvaluator** - A rule evaluator maintained by a deployed inline server and exposed through the Inline Service application interface.
- **EvaluationOptions** - A collection of user defined options that can be passed in to a rule evaluator. These options include the runtime exception policy options and the evaluator options.

In addition, two new external rules exceptions can be caught while using these API interfaces:

- **ValidationException** - Thrown when a rule fails to compile because of problems in rule metadata
- **EvaluationException** - Thrown when rule execution fails with a RuntimeException

17.2.3.4 External Rule Error Handling and Logging

The external rule errors and the corresponding Oracle RTD behavior are listed in the following table. Note that the behavior can be tuned through modifying external rule evaluation functions.

Table 17–2 External Rule Errors and Oracle RTD Behavior

Error Event	Oracle RTD Action
Rule compilation error - Unparseable rule metadata - Rule metadata does not conform to schema - Missing/misspelled Inline Service attribute reference - Java compilation error	**if** (RuleCache.get(rule) == null) **if** (propagateExceptions && blockEvaluationUntilCached) throw underlying exception wrapped in ValidationException, do not log **else** log.ERROR rule metadata, underlying cause and full stack trace **else** **if** (propagateExceptions) throw generic ValidationException **else** log.ERROR one line generic error message
Rule execution error - Referenced Inline Service attribute not initialized by execution time - Other exception thrown by a callee of Rule.execute()	**if** (propagateExceptions) throw underlying exception wrapped in EvaluationException, do not log **else** log.WARN one line underlying cause
Rule uuid error - Unparseable uuid - Unparseable timestamp - Missing uuid and/or timestamp	**if** (propagateExceptions) throw ValidationException with underlying cause, do not log **else** log.ERROR one line underlying cause

17.2.4 Setting Up External Rules in Decision Studio

You can set up the following types of external rule:

- Choice Group Eligibility Rule
- Choice Eligibility Rule

- Filtering Rule

The eligibility rules are defined as part of a choice or choice group definition. Filtering rules are standalone rules, in that they are created independently of any other Oracle RTD object. After creation, filtering rules can be attached to one or more decisions.

This section describes the process of setting up external rules.

The examples in this section are based on the DC_Demo Inline Service, which is released with Oracle RTD.

This section contains the following topics:

- Section 17.2.4.1, "Prerequisite - Setting Up Objects in an External Content Repository"
- Section 17.2.4.2, "Defining the Inline Service Objects for the Rules"
- Section 17.2.4.3, "Defining External Rules for Inline Service Objects"

17.2.4.1 Prerequisite - Setting Up Objects in an External Content Repository

The metadata version of an external rule is stored in an external data source, typically in a column in a database table, for example, the column **RuleMetadata** in table **WEBOFFERS**.

When a rule is related to a dynamic choice, it is customary to store the associated dynamic choice as another column in the same external data source. For more details related to dynamic choices, see Section 17.1.4, "Prerequisite External Data Source for Dynamic Choices."

For example, in the table **SDDS.WEBOFFERS**, the column **RuleMetadata** contains the external rule, and the columns **Category** and **Id** are used to identify dynamic choices.

17.2.4.2 Defining the Inline Service Objects for the Rules

In the Inline Service, you define the data source that contains the external object or objects, then define an entity based on the data source.

Choice groups and choices that require external rules must define choice attributes that are derived from the appropriate entity attributes.

Example

For rule set up, the data source **Web Offers DS** is based on the table **SDDS.WEBOFFERS**, and the entity **Web Offers**, derived from Web Offers DS, includes the attribute **Rule Metadata**.

For dynamic choice setup, the entity **Web Offers** contains **Id**, and the entity **Web Offers List** (also derived from **Web Offers DS**) contains the attribute **Category**.

The choice group **Dynamic Offer**s includes **Rule Metadata** among its choice attributes, as well as the attributes required for dynamic choice definition.

For more details, see Section 17.3, "Example of End to End Development Using Dynamic Choices and External Rules."

17.2.4.3 Defining External Rules for Inline Service Objects

In contrast to rules completely defined and created in Decision Studio, external rules by their very nature are created outside of Decision Studio. However, you must define an external rule for an object by launching the appropriate rule editor within Decision Studio for the object, as follows:

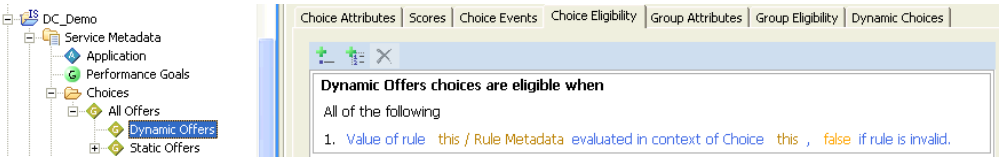
- For external filtering rules, create or edit the filtering rule. For general information, see Section 13.8, "Filtering Rules."
- For external rules for choices and choice groups, select the Choice Eligibility tab or the Group Eligibility tab. For general information, see Section 13.7.5, "About Eligibility Rules."

In each case, as you edit the Boolean statement of the rule, first select the external rule evaluation function that you require:

- External Rules - Evaluate Choice Eligibility Rule
- External Rules - Evaluate Choice Group Eligibility Rule
- External Rules - Evaluate Filtering Rule

Then select or fill in values for the function parameters.

For example, in the DC_Demo Inline Service, the external rule for choice group Dynamic Offers is defined in the Choice Eligibility tab as follows:



For more information on External Rules functions and parameters, see Section 17.2.3.1, "External Rule Evaluation Functions."

17.2.5 Setting Up the External Interface and Embedded Rule Editor

The external third party interface editor is responsible for connecting to Oracle RTD and passing sufficient information for Oracle RTD to launch the Rule Editor within the third party interface. For more information, see Section 17.2.2, "External Rule Editor."

The examples used to illustrate the setup of the external interface and embedded Rule Editor are based on the External Rules Development Helper released with the DC_Demo Inline Service. The files to generate this helper are located in the DC_Demo `dc/jsp` folder.

This section contains the following topics:

- Section 17.2.5.1, "Defining the Rule Editor Widget"
- Section 17.2.5.2, "Changing the Rule Editor Context and Scope"
- Section 17.2.5.3, "Defining the Callback Function"

17.2.5.1 Defining the Rule Editor Widget

The Rule Editor can be embedded in a third party interface by creating an HTML form inside the third party editor HTML for use with the Rule Editor widget.

The form should set the action to `/ui/workbench` and create an iframe to house the embedded editor.

Note: For cross-domain actions, Web browsers have security mechanisms that prevent Javascript from interacting with a frame or widget whose content is from another domain.

To resolve the cross-domain problem:

- Disable the browser security that prevents this cross-site-scripting communication channel
- Host the external editor and widget on the same server
- Use a proxy in front of the two servers that rewrites their URLs such that the browser thinks they came from one server

Table 17–3 shows the parameters for each type of rule. The HTML form must create form inputs with values for each of the parameters listed in Table 17–3.

Table 17–3 Parameters for Embedded Rule Editors

Parameter	Description
app	Inline Service identifier.
url	sdo/editor.jsp This is the url of the editor jsp file. DO NOT CHANGE!
object	Identifier of the object containing the rule. See Table 17–4 for details.
type	Type of the object containing the rule. See Table 17–4 for details.
editingAspect	Editing aspect. See Table 17–4 for details.
callback	Name of the Javascript callback function. This function is called whenever editor events are returned.

Table 17–4 shows the options for the rule-specific parameters.

Table 17–4 Rule-Specific Parameter Options

Rule Type	object	type	editingAspect
Group Eligibility Rule	Group identifier	choiceGroup	rule
Group's Choice Eligibility Rule	Group identifier	choiceGroup	choiceRule
Choice Eligibility Rule	Choice identifier	choice	rule
Filtering Rule	*<Omit this parameter>*	""	whole

The form inputs help to create an initial context and scope for the embedded Rule Editor.

For an example, see Defining the Rule Editor IFrame in the DC_Demo External Rules Helper.

17.2.5.2 Changing the Rule Editor Context and Scope

The embedded Rule Editor context and scope can also be dynamically changed with Javascript functions.

For an example, see Changing Rule Editor Context and Scope in the DC_Demo External Rules Helper.

17.2.5.3 Defining the Callback Function

The Javascript callback function must be created to respond to events returned by the embedded Rule Editor. The embedded Rule Editor will call the callback function with a single object parameter. This object will always have a **type** property that specifies the event type that is occurring. Each event type may use the **data** property to provide additional information.

The events that represent the current state of the embedded Rule Editor are the following:

- **editorReady**

  After the embedded Rule Editor has completed the required rule processing, it fires the **editorReady** event.

  There are three functions available as properties of the data object to stow away for calling in the future:

  – isValid(), which returns a boolean value

  – isModified(), which returns a boolean value

  – getXml(), which returns a string value

- **modified**

  The **modified** event is called upon every modification of the rule. It does not make use of the **data** property.

For an example, see Defining the Callback Function in the DC_Demo External Rules Helper.

17.3 Example of End to End Development Using Dynamic Choices and External Rules

DC_Demo is an Inline Service released with Oracle RTD that demonstrates the setup and use of dynamic choices and external rules.

The following section provides an overview of how DC_Demo utilizes dynamic choices external rules in Oracle RTD, and how this fits into the complete application development process.

As a summary, the main development procedures described in this section are the following:

- Using the standard external rule evaluation functions in dynamic choice eligibility evaluation
- Embedding the external rules editor in an Inline Service Web page
- Integrating with a dynamic choice content database
- Editing an external rule
- Reviewing dynamic choice reports in Decision Center

This section contains the following topics:

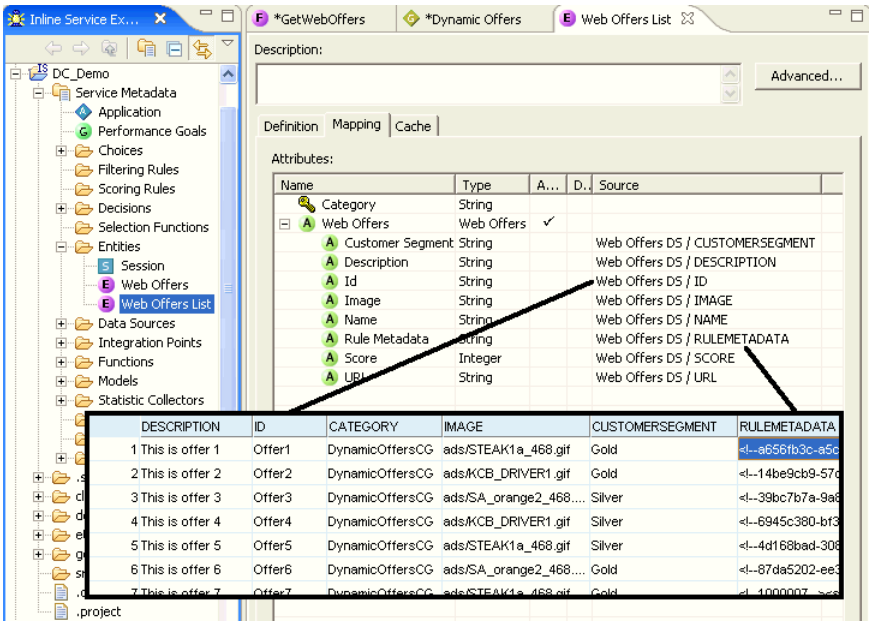
- Section 17.3.5, "Pushing External Rules To a Production Environment"
- Section 17.3.6, "Viewing Reports for Dynamic Choices"

17.3.1 Database Driven Dynamic Choices

Dynamic choices can be managed by a content management server and stored in a content database.

The Inline Service DC_Demo derives its dynamic choice Web offers from a table called **WEBOFFERS**. This table contains a column called RULEMETADATA which stores the external rule metadata used in choice eligibility evaluation. The table also contains a column called CATEGORY which specifies the ID of the parent dynamic choice.

The database columns are mapped to Oracle RTD entity object attributes through the data source **Web Offers DS**, as in the following image (*a third party tool was used to capture the WEBOFFERS rows*):

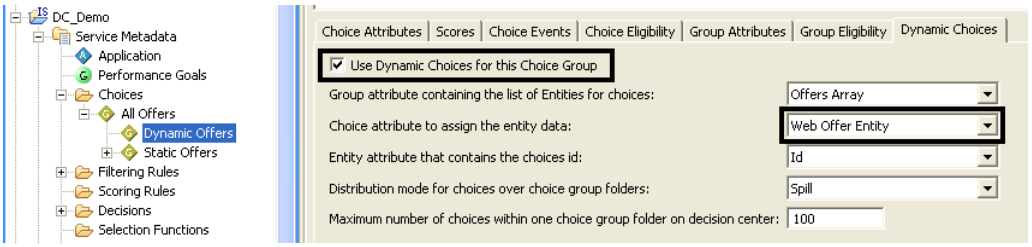


Dynamic choices are set up by creating two entity objects, as follows:

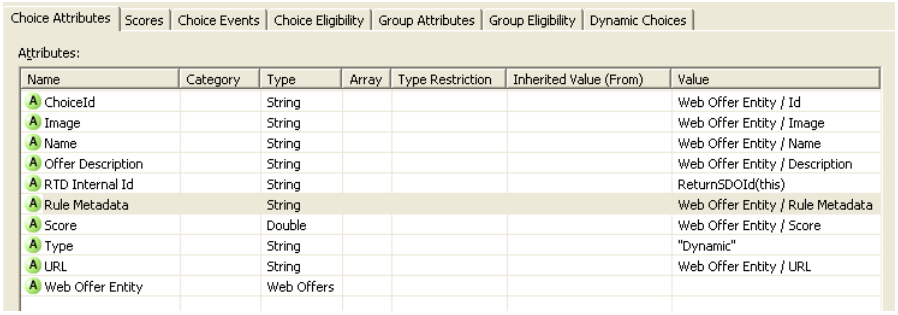
- The entity called **Web Offers** contains the attribute information for one dynamic choice.
- The entity **Web Offers List** contains a set of dynamic offers obtained by the database and is mapped to a datasource that describes the dynamic choice table information.

A choice group called **Dynamic Offers** is created whose dynamic choice configuration is enabled and set to use the Web Offers entity for assigning choice attribute data.

The following image shows dynamic choice definition for the Dynamic Offers choice group.



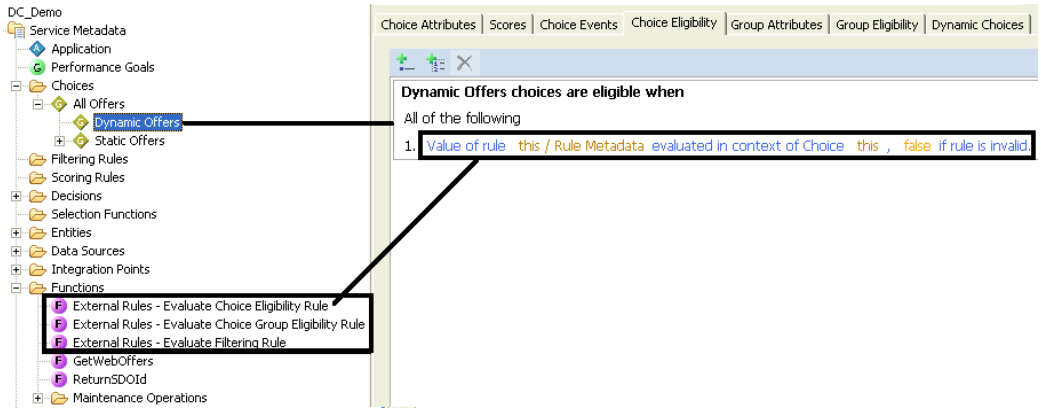
The following image shows dynamic choice attributes mapped to the Web Offers entity attributes. Note the choice attribute **Rule Metadata**, which is used to store the external rule metadata and will be used to evaluate choice eligibility.



For general information about setting up dynamic choices, see Section 17.1.5, "Overview of Setting up Dynamic Choices in Decision Studio."

17.3.2 Evaluating External Rules

Dynamic choice eligibility evaluation rules can reference external rules stored as rule metadata in a dynamic choice attribute by using the external rule evaluation functions provided in Decision Studio. The dynamic choice group Dynamic Offers uses the external rule function **External Rules - Evaluate Choice Eligibilty Rule** function as shown in the following image:



17.3.3 Embedding an External Rule Editor in a Third Party Interface

The Oracle RTD external Rule Editor provides a graphical user interface that can be used to create and edit external rules for dynamic choices.

For general information about embedding the Rule Editor in an external interface, see Section 17.2.5.1, "Defining the Rule Editor Widget."

As a summary, the form that sets up the Rule Editor Iframe must define the following form inputs:

- **app** - the name of the Inline Service, for example, DC_Demo.
- **url** - sdo/editor.jsp (the url of the editor jsp file)
- **object** - the default parent Inline Service choice type, for example, AllOffersCG.
- **type** - the default scope of the editor (values: choiceGroup, choice, "" - for filtering rules)
- **editingAspect** - the default editing aspect (values: choiceRule, rule, whole)
- **callback** - the Javascript callback function; function called whenever editor events are returned

The form inputs help to create the initial context and scope for the embedded rule editor.

Defining the Rule Editor IFrame in the DC_Demo External Rules Helper

The following is an example of how the embedded rule editor is integrated into the DC_Demo external Rule Editor helper:

```
<!--
  form attributes:

  name:   form name (i.e. editorViewForm)
  target:   form iframe target name (i.e. editorViewIFrame)
  method:   post (required)
  action:   /ui/workbench (editor servlet url; required)

-->

<form name="editorViewForm" target="editorViewIFrame"
  method="post" action="/ui/workbench">

  <iframe frameborder="0" name="editorViewIFrame"/>

  <!--
   form inputs:

   app:           inline service name (for example, DC_Demo)
   url:           embedded editor url (a constant)
   object:        parent dynamic choice group ID (for example, AllOffersCG)
   type:          rule evaluation scope or context (for example, choiceGroup)
   editingAspect: editor aspect view (for example, choiceRule)
   callback:      javascript callback function (for example, callbackFunction)
  -->

  <input type=hidden name=app value=DC_Demo>
  <input type=hidden name=url value=sdo/editor.jsp>
  <input type=hidden name=object value=AllOffersCG>
  <input type=hidden name=type value=choiceGroup>
  <input type=hidden name=editingAspect value=choiceRule>
  <input type=hidden name=callback value=callbackFunction>

</form>
```

Changing Rule Editor Context and Scope in the DC_Demo External Rules Helper

The embedded Rule Editor context and scope can also be dynamically changed with Javascript functions.

The following is an example of how DC_Demo dynamically changes the context and scope of the Rule Editor using defined Javascript functions to change the form input values:

```
<!--
  editorViewForm:                name of the form containing the rule editor
  editorViewForm.object:         ID of the choice or choice group context
  editorViewForm.type:           scope or context type (for example, choiceGroup)
  editorViewForm.editingAspect:  editor aspect view (for example, choiceRule)
-->

<script>
groupChoiceScope: function() {
  editorViewForm.object.value = "DynamicOfferCG";
  editorViewForm.type.value = "choiceGroup";
  editorViewForn.editingAspect.value = "choiceRule";
  loadRule();
}

</script>
```

Defining the Callback Function in the DC_Demo External Rules Helper

The Javascript callback function responds to events returned by the embedded Rule Editor. The events returned include **editorReady** and **modified** which represent the current state of the embedded Rule Editor.

The following is an example of DC_Demo's editor helper Javascript callback function **callbackFunction**, which obtains the **isValid** and **isModified** booleans and the rule metadata data from the editor after it fires an **editorReady** event:

```
<!--
  event.type:              the event type name (for example, editorReady)
  event.data.isValid:      the is valid rule return boolean
  event.data.isModified:   the is modified rule return boolean
  event.data.getXml:        returns the metadata of the rule in the editor
-->
<script>
callbackFunction: function(event) {
  switch(event.type) {
    case "editorReady":
      isValid = event.data.isValid;
      isModified = event.data.isModified;
      getXml = event.data.getXml;
      break;
    case "modified":
      log("is modified");
      break;
    default:
      throw "unexpected callback event type: " + event.type;
  }
}

</script>
```

17.3.4 DC_Demo External Rules Deployment Helper

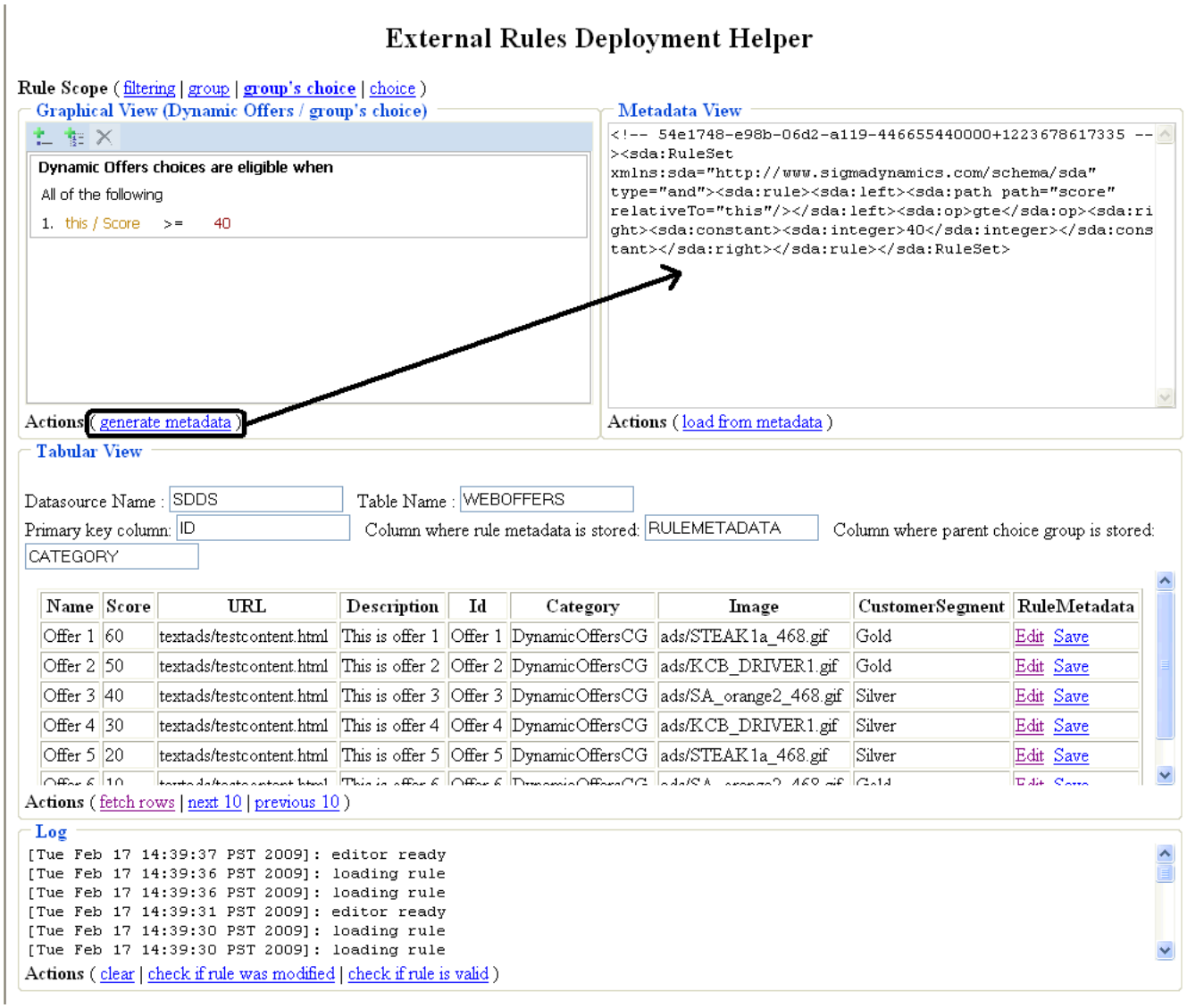
Oracle RTD supplies an external rules editor helper, in the Inline Service **DC_Demo**, in the form of two files, `external_rules_deployment_helper.html` and `database.jsp`, visible in Decision Studio under the folder path `dc > jsp`.

This editor helper interface is provided as an example of how to integrate the external rules editor widget into a third party interface. Through this interface, users can edit external rules for dynamic choices defined in the database table WEBOFFERS.

The DC_Demo editor helper is broken into four sections:

- The Graphical View contains the actual external rules editor and can be used to edit the rule.
- The Metadata View stores the metadata version of the external rule which can be saved back into a dynamic choice table row.
- The Tabular View is a database management section which allows user to search for, edit, and save dynamic choice external rules stored in the database table WEBOFFERS.
- The Log section displays actions performed in the editor helper.

The following image shows an example of the helper window, showing a rule that has been edited in the Graphical View, with the edited rule metadata visible in the Metadata View.



The Tabular View displays the rows of the database table WEBOFFERS, where the significant columns are:

- CATEGORY, which stores the Id of the dynamic choice group
- RULEMETADATA, which stores the external rule metadata

After editing the rule in the Graphical View rule editor, a user clicks the generate metadata link, and the generated metadata then appears in the Metadata View.

17.3.5 Pushing External Rules To a Production Environment

One of the main purposes of the external Rule Editor is to push updated dynamic choice external rules back into a production environment. Typically, a third party content database is used to store dynamic choices and their external rules.

In DC_Demo, dynamic choices are stored in a table called WEBOFFERS. This table contains a column called RULEMETADATA which used to store the external rule metadata. Another column called CATEGORY is used to store the ID of the parent dynamic choice group.

The DC_Demo external rule editor helper saves an external rule back to the database when a user selects a dynamic choice table row and clicks the Save link in the row.

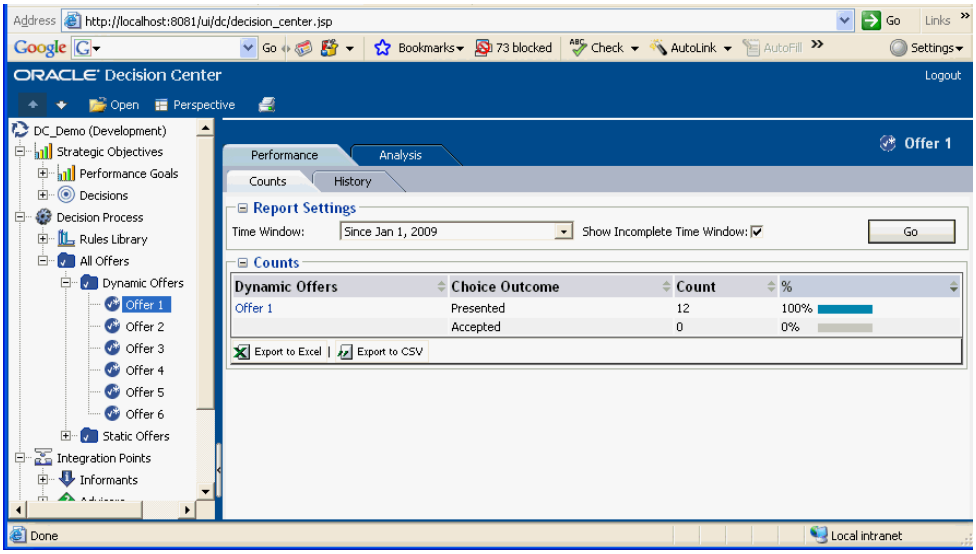
17.3.6 Viewing Reports for Dynamic Choices

Note: Decision Center does not by default display the eligibility rule of a Dynamic Choice, even when the rule was edited with an external Rule Editor.

Decision Center can be used to view reports for each dynamic choice defined in a content database, which were actually recommended and added to model learning data. This is done by logging into a Decision Center Inline Service and opening the Decision Process section in the Decision Center navigator.

From here, any defined dynamic choice groups will be listed and will contain all dynamic offers defined in database tables for each dynamic choice group, that were recommended and added to model learning data. Choices in the database tables that were not added to model learning data do *not* appear in Decision Center reports.

The following is an image of a Decision Center report, with the navigator tree showing the DC_Demo dynamic choices:



17.4 Externalized Performance Goal Weighting

In the Oracle RTD decisioning process, decisions can target segments of the population and weight the performance metrics attached to that decision for each segment.

You can set up your decision in two ways, depending on what kind of weights you want for your performance goals:

- Pre-defined weights, whose values are specified in the Inline Service
- Custom weights, whose values can be calculated or changed at run time

By selecting custom weights for performance goals in the Inline Service, end users can influence the decisioning process with on-the-spot decision process modifications, which effectively segment different population segments at run time.

For details of how to define and use custom performance goal weights in an Inline Service, see the following topics:

- Section 13.6, "Performance Goals"
- Section 13.11, "About Decisions"
- Section 13.11.1, "Segmenting Population and Weighting Goals"

A

Development Error Messages

This appendix shows the error messages that may occur during development.

> **Note:** For more general error messages, refer to Appendix B, "Exceptions" in *Oracle Real-Time Decisions Installation and Administration Guide*.

Table A–1 Development Error Messages

Error	Explanation
Internal error in code generator. See error log for details.	Check the Problems pane in Decision Studio for errors. If none are apparent, contact Oracle Support Services.
Cannot get *Inline_Service* from the database. Will mark it invalid	Check to make sure that the database server is running, that you have the proper drivers, and that you have connectivity to the database server.
Error loading Inline Service *Inline_Service_Name*. Will mark it as invalid in the database.	If you get an error from the server on loading your Inline Service, check to see if the logic in your Application Initialization logic and Session Initialization logic is correct.
Response for an asynchronous request to Advisor "*Advisor_name*" will not be sent	If you want a response from an Advisor, you must use the `invoke()` method, not `invokeAsync()`.
Invoke failed	Make sure that you have connectivity to Real-Time Decision Server. Check that your properties file is properly configured.
Internal error	Contact Oracle Support Services.
Internal error while generating code for *Inline_Service*.	Contact Oracle Support Services.
Error in Application Session Cleanup.	This error can be caused by incorrect logic in the Application and Session elements. Check to see if the logic in your Application Cleanup logic and Session Cleanup logic is correct.
Failed to load study "*study_name*"	This error can be caused by database connection issues. Check to make sure that the database server is running, that you have the proper drivers, and that you have connectivity to the database server.
Failed to save study "*study_name*".	This error can be caused by database connection issues. Check to make sure that the database server is running, that you have the proper drivers, and that you have connectivity to the database server.

Table A–1 (Cont.) Development Error Messages

Error	Explanation
Failed to load prediction model "*model_name*"	This error can be caused by database connection issues. Check to make sure that the database server is running, that you have the proper drivers, and that you have connectivity to the database server.
Error delivering response message: *error_details*	Your `invoke()` on the Integration Point may have timed out. Check the timeout setting in the properties file.
Product sdstudio could not be found.	This error message is informational and can be ignored.

B

Examples of Data Sources from Stored Procedures

This appendix shows examples of how to create data sources from Oracle, SQL Server, and DB2, and then how to create entities and session attributes from these data sources.

The examples are based on the CrossSellCustomers table. For details of setting up this table, see the topic "Populating the CrossSell Example Data" in *Oracle Real-Time Decisions Installation and Administration Guide*.

This appendix contains the following topics:

B.1 Creating a Data Source from Single Result Stored Procedures

To create a data source from single result stored procedures:

1. Create the stored procedure **Get_Single_CustomerInfo** in your Oracle, SQL Server, or DB2 database, using the appropriate commands:

   (A) Oracle

```
CREATE PROCEDURE GET_SINGLE_CUSTOMERINFO
(
   P_ID IN INTEGER,
   P_AGE OUT INTEGER,
   P_OCCUPATION OUT VARCHAR2,
   P_LASTSTATEMENTBALANCE OUT FLOAT
)
AS
BEGIN
   SELECT AGE, OCCUPATION, LASTSTATEMENTBALANCE INTO P_AGE, P_OCCUPATION,
          P_LASTSTATEMENTBALANCE
   FROM CROSSSELLCUSTOMERS
   WHERE CROSSSELLCUSTOMERS.ID = P_ID;
END;
```

   (B) SQL Server

```
CREATE PROCEDURE Get_Single_CustomerInfo
   @pId INTEGER,
   @pAge INTEGER OUTPUT,
   @pOccupation VARCHAR(20) OUTPUT,
```

```
   @pLastStatementBalance FLOAT OUTPUT
AS
   SELECT @pAge = Age,
         @pOccupation = Occupation,
          @pLastStatementBalance = LastStatementBalance
   FROM CrossSellCustomers
   WHERE Id = @pId;
GO
```

(C) DB2

```
CREATE PROCEDURE DB2ADMIN.GET_SINGLE_CUSTOMERINFO
(
   IN P_ID INTEGER,
   OUT P_AGE INTEGER,
   OUT P_OCCUPATION VARCHAR(20),
   OUT P_LASTSTATEMENTBALANCE REAL
)
LANGUAGE SQL
P1: BEGIN
   DECLARE CURSOR_ CURSOR WITH RETURN FOR
      SELECT AGE, OCCUPATION, LASTSTATEMENTBALANCE
      FROM DB2ADMIN.CROSSSELLCUSTOMERS AS CROSSSELLCUSTOMERS
      WHERE CROSSSELLCUSTOMERS.ID = P_ID;
   OPEN CURSOR_;
   FETCH CURSOR_ INTO P_AGE, P_OCCUPATION, P_LASTSTATEMENTBALANCE;
   CLOSE CURSOR_;
END P1
```

2. Create a JDBC data source for the stored procedure in the application server that you are using.

   For details of how to create data sources in the application servers, see "Configuring Data Access to Oracle Real-Time Decisions" in *Oracle Real-Time Decisions Installation and Administration Guide*.

3. In Decision Studio, create the stored procedure data source **DS_Single_Customer**, by importing the **Get_Single_CustomerInfo** stored procedure from your database.

   For the SQL Server stored procedure, change the direction of the parameters **pAge**, **pOccupation**, and **pLastStatementBalance** from **Input/Output** to **Output**.

4. In Decision Studio, create the entity **Ent_Single_Customer**, by importing the data source **DS_Single_Customer**.

5. Add the attribute **Id**, of data type Integer.

6. In the Mapping tab, in the Data Source Input Values area, set the Input Value for the Input Column **pId** to **Id**.

7. Open the **Session** entity, and add a new attribute **cust_sp**, setting the data type to **Ent_Single_Customer**.

B.2 Creating a Data Source from Stored Procedures with One Result Set

To create a data source from stored procedures with one result set:

1. Create the stored procedure **Get_OneSet_CustomerInfo** in your Oracle, SQL Server, or DB2 database, using the appropriate commands:

   (A) Oracle

```
CREATE PROCEDURE GET_ONESET_CUSTOMERINFO
(
   P_CREDITLINEAMOUNT IN INTEGER,
   CURSOR_ IN OUT TYPES.REF_CURSOR
)
AS
BEGIN
   OPEN CURSOR_ FOR
      SELECT * FROM CROSSSELLCUSTOMERS
      WHERE CREDITLINEAMOUNT >= P_CREDITLINEAMOUNT;
END;
```

(B) SQL Server

```
CREATE PROCEDURE Get_OneSet_CustomerInfo
   @pCreditLineAmount INTEGER
AS
   SET NOCOUNT ON;
   SELECT * FROM CrossSellCustomers
   WHERE CreditLineAmount >= @pCreditLineAmount;
GO
```

(C) DB2

```
CREATE PROCEDURE DB2ADMIN.GET_ONESET_CUSTOMERINFO
(
   IN P_CREDITLINEAMOUNT INTEGER
)
DYNAMIC RESULT SETS 1
LANGUAGE SQL
P1: BEGIN
   DECLARE CURSOR_ CURSOR WITH RETURN FOR
      SELECT * FROM DB2ADMIN.CROSSSELLCUSTOMERS AS CROSSSELLCUSTOMERS
      WHERE CROSSSELLCUSTOMERS.CREDITLINEAMOUNT >= P_CREDITLINEAMOUNT;
   OPEN CURSOR_;
END P1
```

2. Create a JDBC data source for the stored procedure in the application server that you are using.

   For details of how to create data sources in the application servers, see "Configuring Data Access to Oracle Real-Time Decisions" in *Oracle Real-Time Decisions Installation and Administration Guide*.

3. In Decision Studio, create the stored procedure data source **DS_OneSet_Customer**, by importing the **Get_OneSet_CustomerInfo** stored procedure from your database.

4. In the Results Set Details section, add a result set.

5. Check **Allow multiple rows**.

6. For the SQL Server stored procedure, add the following column names exactly as shown with the given data types:

   - Age [Integer]
   - Occupation [String]
   - LastStatementBalance [Double]

   For the Oracle and DB2 stored procedures, add the following column names exactly as shown with the given data types:

- AGE [Integer]
- OCCUPATION [String]
- LASTSTATEMENTBALANCE [Double]

7. In Decision Studio, create the entity **Ent_OneSet_Customer**, by importing the data source **DS_OneSet_Customer**.
8. Add the attribute **CreditLineAmount**, of data type Integer, and set its default value to 50000.

   This will limit results to around 30 rows.
9. Check the Array column for the attributes **Age**, **Occupation**, and **LastStatementBalance**.
10. In the Mapping tab, in the Data Source Input Values area, set the Input Value for the Input Column **pCreditLineAmount** to **CreditLineAmount**.
11. Open the **Session** entity, and add a new attribute **cust_oneset_sp**, setting the data type to **Ent_OneSet_Customer**.

B.3 Creating a Data Source from Stored Procedures with Two Result Sets

To create a data source from stored procedures with two result sets:

1. Create the stored procedure **Get_TwoSets_CustomerInfo** in your Oracle, SQL Server, or DB2 database, using the appropriate commands:

   (A) Oracle

```
CREATE PROCEDURE GET_TWOSETS_CUSTOMERINFO
(
   P_CREDITLINEAMOUNT IN INTEGER,
   CURSOR1_ IN OUT TYPES.REF_CURSOR,
   CURSOR2_ IN OUT TYPES.REF_CURSOR
)
AS
BEGIN
   OPEN CURSOR1_ FOR
      SELECT * FROM CROSSSELLCUSTOMERS
      WHERE CREDITLINEAMOUNT >= P_CREDITLINEAMOUNT;

  OPEN CURSOR2_ FOR
      SELECT * FROM CROSSSELLCUSTOMERS
      WHERE CARDTYPE = 'Platinum' AND CREDITLINEAMOUNT >= P_CREDITLINEAMOUNT;
END;
```

   (B) SQL Server

```
CREATE PROCEDURE Get_TwoSets_CustomerInfo
   @pCreditLineAmount INTEGER
AS
   SET NOCOUNT ON;
   SELECT * FROM CrossSellCustomers
   WHERE CreditLineAmount >= @pCreditLineAmount;
   SELECT * FROM CrossSellCustomers
   WHERE CreditLineAmount >= @pCreditLineAmount AND CardType = 'Platinum';
GO
```

   (C) DB2

```
CREATE PROCEDURE DB2ADMIN.GET_TWOSETS_CUSTOMERINFO
```

```
(
   IN P_CREDITLINEAMOUNT INTEGER
)
DYNAMIC RESULT SETS 2
LANGUAGE SQL
P1: BEGIN
   DECLARE CURSOR1_ CURSOR WITH RETURN FOR
      SELECT * FROM DB2ADMIN.CROSSSELLCUSTOMERS AS CROSSSELLCUSTOMERS
      WHERE CROSSSELLCUSTOMERS.CREDITLINEAMOUNT >= P_CREDITLINEAMOUNT;

  DECLARE CURSOR2_ CURSOR WITH RETURN FOR
     SELECT * FROM DB2ADMIN.CROSSSELLCUSTOMERS AS CROSSSELLCUSTOMERS
      WHERE CROSSSELLCUSTOMERS.CREDITLINEAMOUNT >= P_CREDITLINEAMOUNT
      AND CROSSSELLCUSTOMERS.CARDTYPE = 'Platinum';

   OPEN CURSOR1_;
   OPEN CURSOR2_;
END P1
```

2. Create a JDBC data source for the stored procedure in the application server that you are using.

   For details of how to create data sources in the application servers, see "Configuring Data Access to Oracle Real-Time Decisions" in *Oracle Real-Time Decisions Installation and Administration Guide*.

3. In Decision Studio, create the stored procedure data source **DS_TwoSets_Customer**, by importing the **Get_TwoSets_CustomerInfo** stored procedure from your database.

4. In the Results Set Details section, add a result set.

5. Check **Allow multiple rows**.

6. For the SQL Server stored procedure, add the following column names exactly as shown with the given data types:

   - Age [Integer]
   - Occupation [String]
   - LastStatementBalance [Double]

   For the Oracle and DB2 stored procedures, add the following column names exactly as shown with the given data types:

   - AGE [Integer]
   - OCCUPATION [String]
   - LASTSTATEMENTBALANCE [Double]

7. Repeat steps 4 through 6 for the second result set.

8. In Decision Studio, create the entity **Ent_TwoSets_Customer**, by importing the data source **DS_TwoSets_Customer**.

9. Add the attribute **CreditLineAmount**, of data type Integer, and set its default value to 50000.

   This will limit results to around 30 rows.

10. Check the Array column for the attributes **Age**, **Occupation**, and **LastStatementBalance**.

11. In the Mapping tab, in the Data Source Input Values area, set the Input Value for the Input Column **pCreditLineAmount** to **CreditLineAmount**.
12. Open the **Session** entity, and add a new attribute **cust_twosets_sp**, setting the data type to **Ent_TwoSets_Customer**.