Corticon
Extensions

Copyright

Table of Contents

1

Overview of Corticon extensions

When you are creating business rules, you sometimes need to perform operations that are not built natively into Corticon. For example, you may need to apply a complex mathematical formula or to retrieve data from an external web service. Corticon provides the ability to add custom extensions for just such purposes.

Extensions are written as custom Java code that you package into one or more JAR files. You simply add extension jar files to your rule project to have them bundled into the EDS file for your Decision Service. This ease of adding extensions makes it easier to develop extensions yourself or to use open source extensions that you download from the Corticon community. By bundling extensions with EDS files, the EDS file becomes *self-contained*. You can deploy it to a Corticon Server without modifying the server's classpath. It also allows you to have different Decision Services, or versions of Decisions Services, running that use different versions of an extension.

When developing an extension, it needs to implement one or more Java interfaces that Corticon has defined. The Corticon Extensions API provides for Java annotations to describe the extension, thereby eliminating the need for additional configuration files.

When developing a project in Corticon Studio you can add extensions to your project through the project's **Properties** dialog box, so that they are available for development and running rule tests. The **Package and Deploy** wizard in Corticon Studio will include any extensions used by the project into the EDS file it generates or deploys. If you want to script the building of your EDS files, you can also use the Corticon ANT scripts to package extensions into EDS files.

For compatibility with previous releases you can still place extension JAR files on the Corticon classpath so that they are available to all Decision Services.

Extensions can be created in the Java development environment included in Corticon Studio, or you can use another IDE.

There are two types of Corticon extensions:

- **Extended operators** - Operators are used when defining conditions and actions in a Rulesheet. While Corticon has a large built-in set of operators, you can expand this set by adding custom operators. Operators can operate on individual attributes, collections or sequences. Examples include:

- Financial functions, such as net present value, and loan amortization

- Statistical functions, such as standard deviation, and permutations

- Engineering functions, such as pi, sine, and cosine

- **Service callouts** - Callouts can be used in a ruleflow to retrieve, modify, or store data that is being processed by the rules. The most common use is to access data in a database or external web service. For example, if your Ruleflow needs to look up an applicant's credit rating, the service callout can have a step in the Ruleflow processing that calls out to a trusted realtime ratings provider, and then adds the response back into the decision processing.

2

Using extensions when creating Decision Services

You might want your project to include extensions that are already packaged and ready to use. The sample extensions bundled with Corticon Studio provide sample Rule projects with their samples already packaged into JARs.

For now, assume that you just want to use the functionality in these extensions.

To use the packaged extension samples in my project:

1. In Corticon Studio, choose **Help > Samples**. Locate the **Advanced** samples:
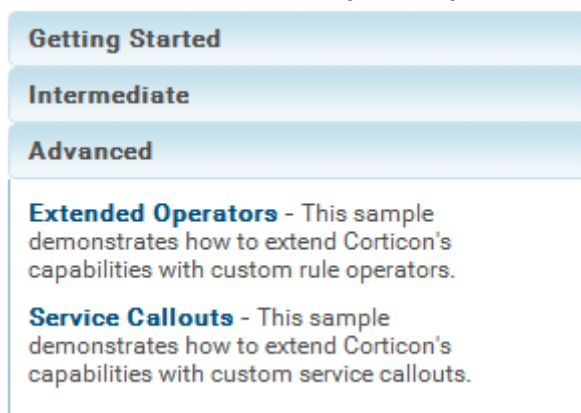
2. Select **Extended Operators**, click **Open**, and then click **OK**.

3. Then do the same to open the **Service Callouts** sample.

4. Your Studio's **Project Explorer** lists the two samples, each with its Java project and its Rules project:
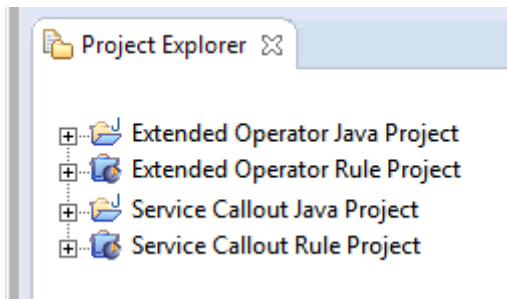
5. Expanding the two Rule Projects, you can see that each has a related samples JAR.
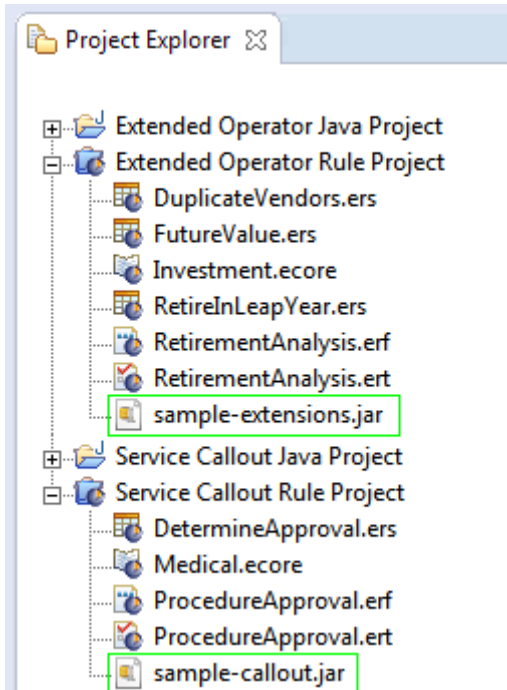
6. Create a new Rule Project named `myProject` and create a very simple Vocabulary that includes a date type:
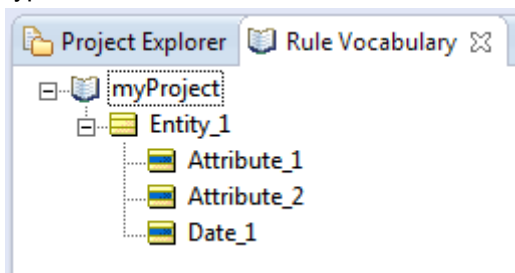
7. Copy the JAR files highlighted in step 5, and then paste them in to `myProject`.

Note: While you could reference the JARs in their projects, copying them into your project insures that when you export that Project, those JARs are included so the references don't break.

8. In the Project Explorer, click on `myProject`, and then select **Properties**. Select **Corticon Extensions**. Click **Add** then navigate to each of those JARs, as shown:
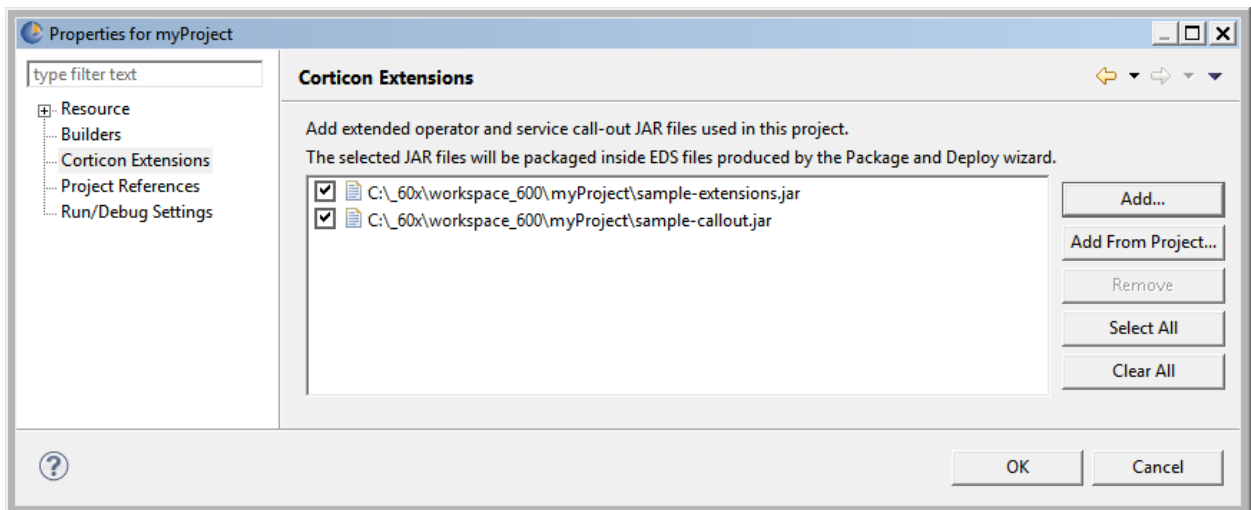
Note: This feature does not support JARs nested within a JAR that you add to a project.

---

9. Create a Rulesheet in `myProject`. Note that the Rule Operators tab adds in the extended operators that are in the JAR, so that you can readily use one as a valid operator in the Rulesheet, as shown:
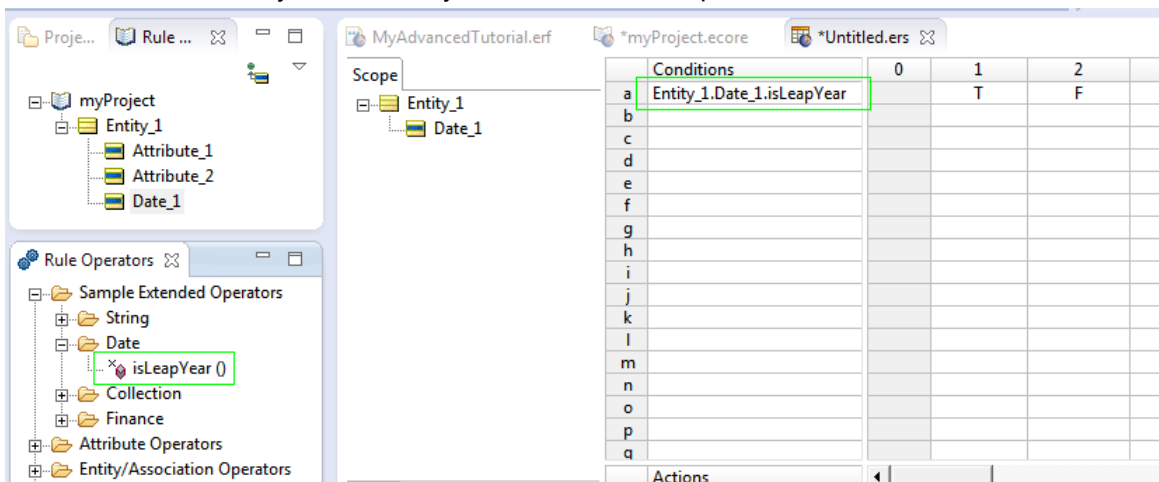
10. Create a Ruleflow in `myProject`. Click **Service Call-out** on the palette, click on the canvas, and then name it. On the **Properties** tab, click the Service Name dropdown to see the service callouts that are packaged in the sample JAR you added to the project, as shown when `getPatientInfo` was selected:
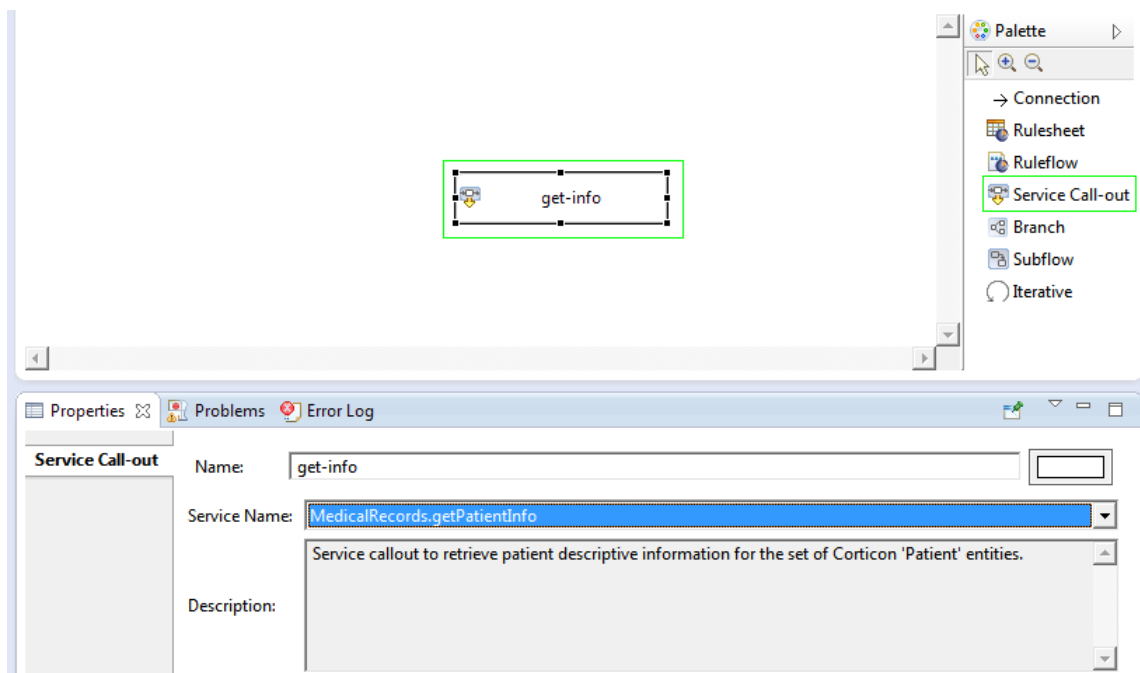
Because the two extension JARs are properties of the project, they are embedded in Decision Services that you package and deploy from Studio.

The next section looks at the source code in each of their Java projects to gain insight into how extensions are created and prepared for use.

3

Reviewing what is in the sample extensions

Both the Extended Operator and Service Callout samples contain Java projects that show how you can create the sample extension JAR yourself, and a Corticon Rules project that demonstrates those extensions.

There are two sets of extension samples:

- **Extended Operator Sample Projects** - The Extended Operators sample contains the source code for several extended operators and a rule project that uses them. The rule project uses extended operators for determining the future value or an investment, if a collection contains duplicate strings, and if a given date is a leap year.

- **Service Callout Sample Projects** - The Service Callout sample contains the source code for several service callouts and a rule project that uses them. The rule project uses service callouts to retrieve and update patient medical data in a pseudo external service. Accessing web services or other external datasources is a common use case for service callouts.

For details, see the following topics:
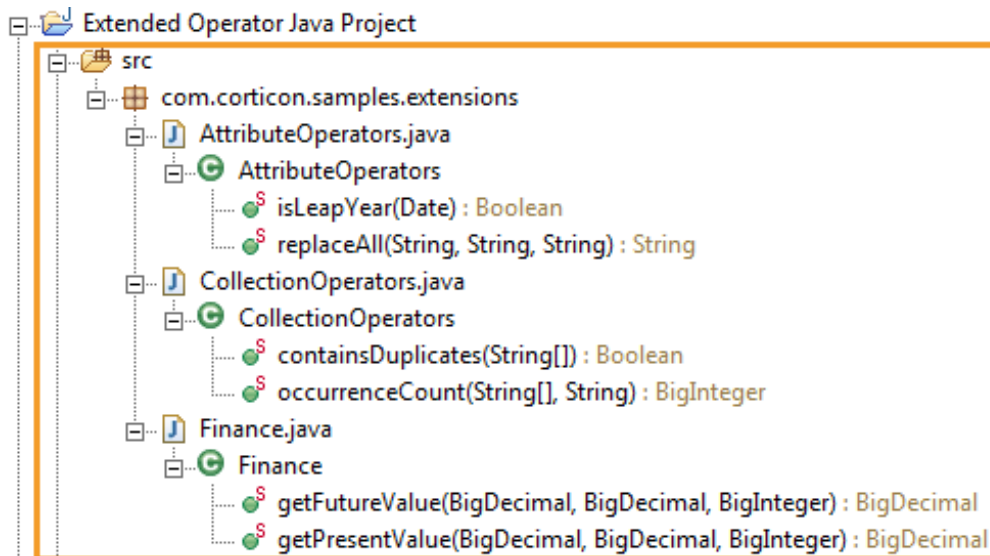
- What is in the Extended Operators Sample Projects
- What is in the Service Callout Sample Projects

What is in the Extended Operators Sample Projects

The **Extended Operator Java project** contains a `src` folder with the source code for the extended operators and has on its build path the standard Java system library and `CcExtensionApi.jar` with the Corticon extension API:
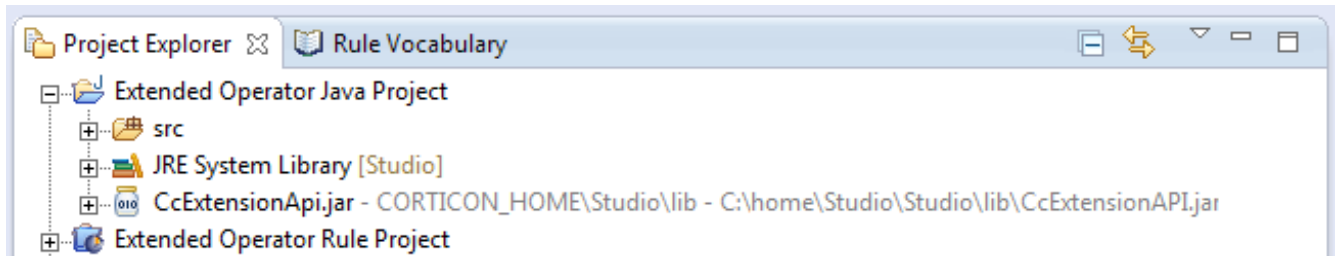
The `src` folder contains the source code for three Java classes that implement extended operators:

The **Extended Operator Rule project** is a set of rule assets that demonstrate the extended operator Java sample It uses the extended operators and the supporting vocabulary, Ruleflows, and Ruletests for the project, plus the JAR file of the extensions built in the Java project, `sample-extensions.jar`:
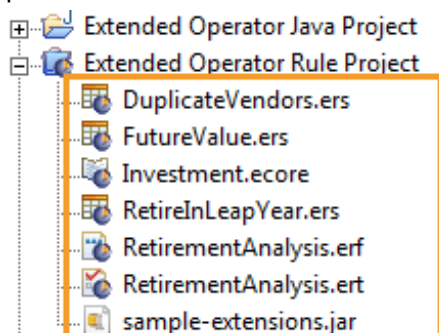
The sample project contains a Vocabulary, and three Rulesheets that demonstrate each of the new extended operators:

1. FutureValue - Determines years to retirement and the future value of the current investment with a constant interest rate.

2. RetireInLeapYear - Checks whether the retirement year is a leap year

3. DuplicateVendors - Checks to determine that there are not multiple accounts with the same vendor

The Ruleflow chains the three Rulesheets together for the Ruletest to see that the extended operators behave as expected.
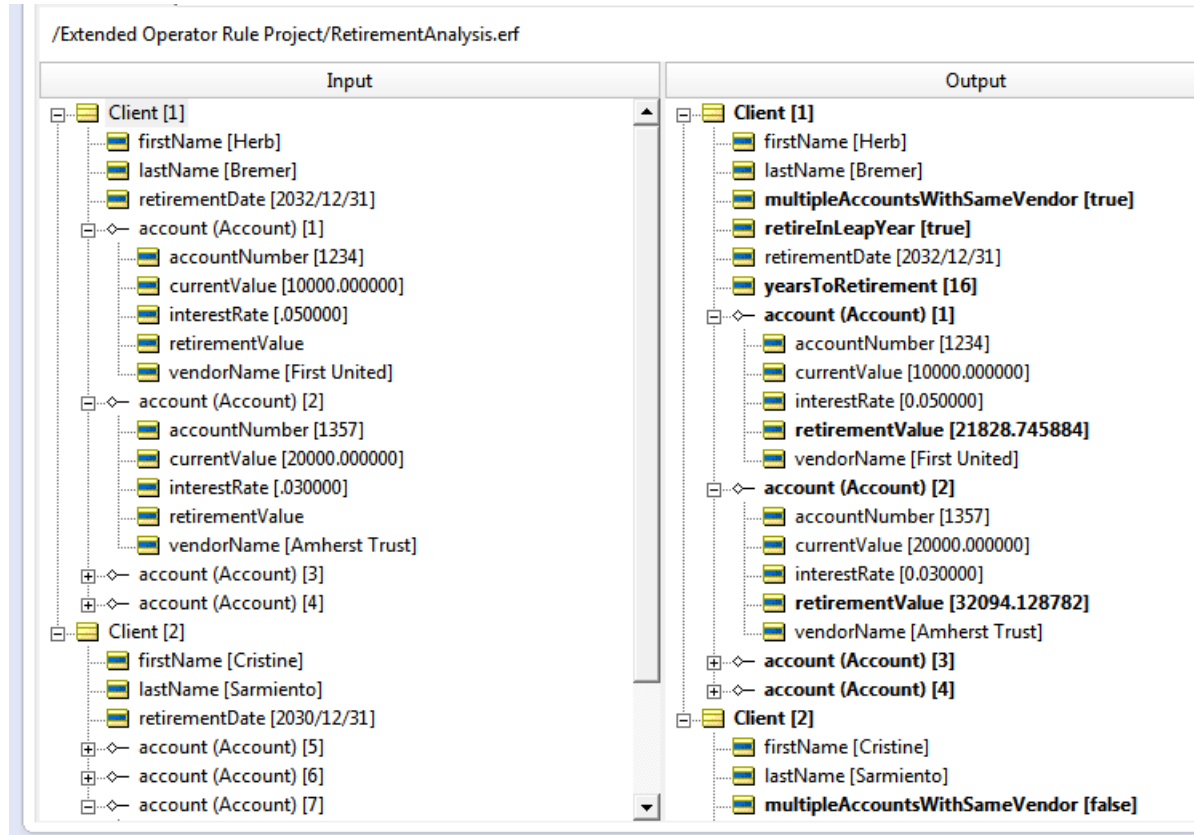
What is in the Service Callout Sample Projects

A Corticon Studio installation includes two Service Callout sample projects that you can immediately bring in to your workspace, and then run.

Service Callout Java and Rule Projects

The **Service Callout Java Project** contains a `src` folder with the source code for the service callouts and has on its build path the standard Java system library and `CcExtensionApi.jar` with the Corticon extension API:
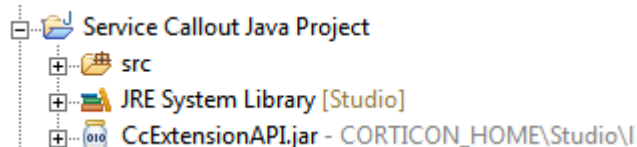
The `src` folder contains the source code for four Java classes that implement service callouts:

- Service Callout Java Project
  - src
    - com.corticon.samples.extensions
      - MedicalRecords.java
        - MedicalRecords
    - com.corticon.samples.simulation
      - MedicalDataSimulator.java
        - MedicalDataSimulator
      - PatientData.java
        - PatientData
      - ProcedureData.java
        - ProcedureData
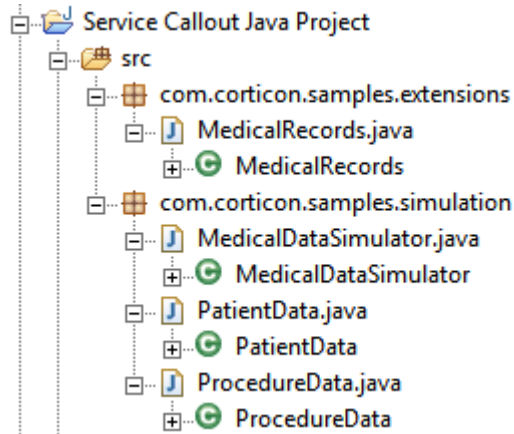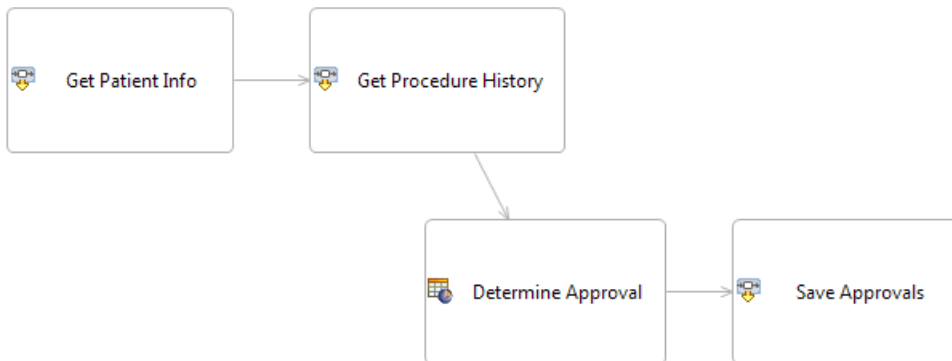
The **Service Callout Rule Project** is a set of rule assets that demonstrate the Service Callout Java samples:

- Service Callout Rule Project
  - DetermineApproval.ers
  - Medical.ecore
  - ProcedureApproval.erf
  - ProcedureApproval.ert
  - sample-callout.jar

The project uses the service callouts and the supporting vocabulary, Rulesheet, Ruleflow, and Ruletest for the project. It also contains the JAR file of the extensions built from the Java project, `sample-callouts.jar`.

The Ruleflow uses three service callouts and a Rulesheet to perform its functions:

The Ruletest shows that the service callouts behave as expected:

/Service Callout Rule Project/ProcedureApproval.erf

Input	Output

Input:
- Patient [1]
  - id [123450]
- Patient [2]
  - id [123452]
- Patient [3]
  - id [123454]
- Patient [4]
  - id [123456]
- Patient [5]
  - id [123458]

Output:
- Patient [1]
  - firstName [Teri]
  - id [123450]
  - lastName [Rivera]
  - procedure (Procedure) [4]
    - approved [true]
    - code [A1329]
    - date [12/27/15]
    - description [Portable oxygen concentrator, rental]
  - procedure (Procedure) [5]
    - approved [false]
    - code [E0169]
    - date [12/27/15]
    - description [Apnea monitor, with recording feature]
- Patient [2]
  - firstName [Margarita]
  - id [123452]
  - lastName [Foster]
  - procedure (Procedure) [2]
    - approved [false]
    - code [C1726]
    - date [12/27/15]
    - description [Catheter, balloon dilatation, non-vascular]
  - procedure (Procedure) [3]

The **Service Callouts** sample is accessed from the Studio's **Help > Samples** in the **Advanced** section.

Weather Callout Java and Rule Projects
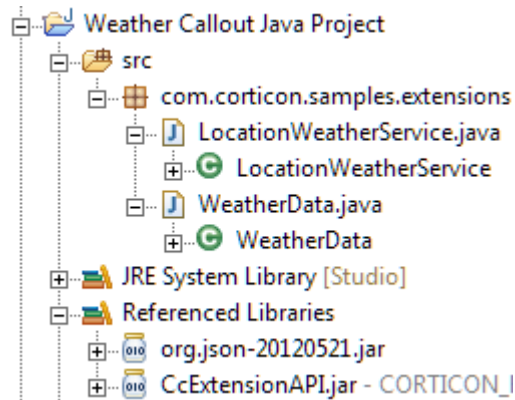
Corticon Studio includes a sample project that shows how service callouts can be reused when they can be parameterized so that each instance can have a different configuration.

The **Weather Callout Java Project** is a Java project that includes the source code for a service callout to call the REST API on OpenWeatherMap.org to retrieve weather data for individual cities. The **Weather Callout Rule Project** uses the callout in the Java project to retrieve data for cities specified in `Location` entities. The service callout is specific to the vocabulary in that it looks for `Location` entities with specific attributes. To retrieve live weather data for a city you need to create an account on http://openweathermap.org/, and then generate an API key that you provide as a property on the callout. By default, the keyword 'demo' is used, and sample weather data is generated.
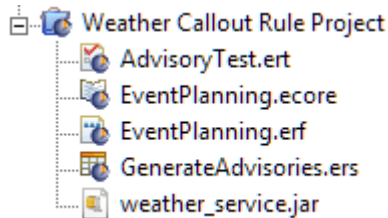
The **Weather Callout Java project** contains a `src` folder with the source code for the weather callout and has on its build path the standard Java system library, `org.json-20120521.jar`, and `CcExtensionApi.jar`.

- Weather Callout Java Project
  - src
  - JRE System Library [Studio]
  - Referenced Libraries
    - org.json-20120521.jar
    - CcExtensionAPI.jar - CORTICON
  - org.json-20120521.jar

The `src` folder contains the source code for two Java classes.

The **Weather Callout Rule Project** is a set of rule assets that demonstrate the Weather Callout Java samples.
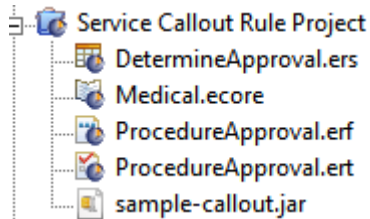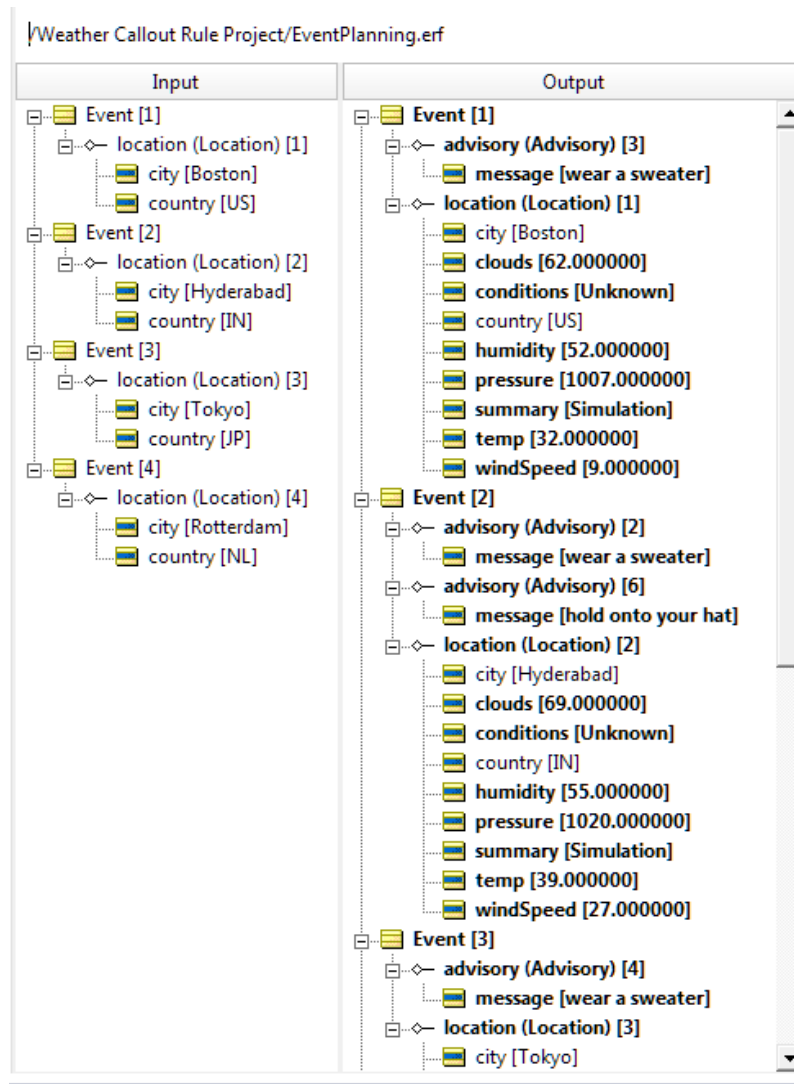

The project uses the service callout and the supporting vocabulary, Rulesheet, Ruleflow, and Ruletest for the project. It also contains the JAR file of the extensions built from the Java project, `weather_service.jar`.

The Ruleflow uses a service callout and a Rulesheet to perform its functions.
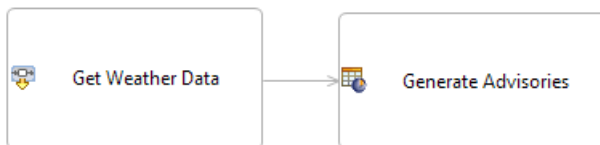
The Ruletest shows that the weather callout behaves as expected.

```
/Weather Callout Rule Project/EventPlanning.erf
```

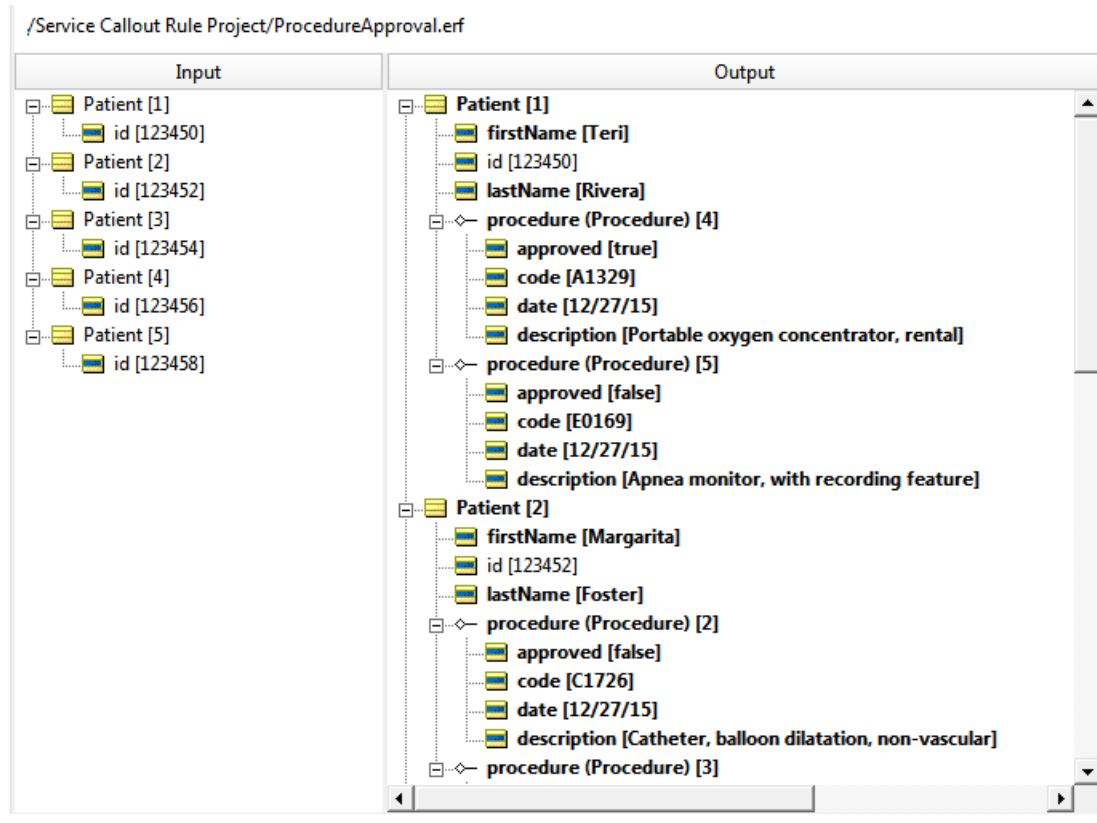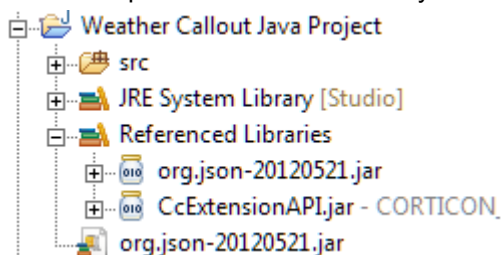| Input	Output
Event [1]	Event [1]
location (Location) [1]	advisory (Advisory) [3]
city [Boston]	message [wear a sweater]
country [US]	location (Location) [1]
Event [2]	city [Boston]
location (Location) [2]	clouds [62.000000]
city [Hyderabad]	conditions [Unknown]
country [IN]	country [US]
Event [3]	humidity [52.000000]
location (Location) [3]	pressure [1007.000000]
city [Tokyo]	summary [Simulation]
country [JP]	temp [32.000000]
Event [4]	windSpeed [9.000000]
location (Location) [4]	Event [2]
city [Rotterdam]	advisory (Advisory) [2]
country [NL]	message [wear a sweater]
	advisory (Advisory) [6]
	message [hold onto your hat]
	location (Location) [2]
	city [Hyderabad]
	clouds [69.000000]
	conditions [Unknown]
	country [IN]
	humidity [55.000000]
	pressure [1020.000000]
	summary [Simulation]
	temp [39.000000]
	windSpeed [27.000000]
	Event [3]
	advisory (Advisory) [4]
	message [wear a sweater]
	location (Location) [3]
	city [Tokyo]

The **Weather Callout** sample is accessed from the Studio's **Help > Samples** in the **Advanced** section.

4

Code conventions

For details, see the following topics:

- Using annotations
- Accessing HTTP Headers in Extended Operators and Service Callouts
- Imports and interfaces used in extensions
- Serializing Response and RuleMessages in Corticon extensions

Using annotations

Corticon extensions use Java annotations to get information about extensions. Corticon supports four types of annotations:

- **Class annotations**:
  - `@TopLevelFolder` - The name of the top level folder that will contain the extended operator. The operator tree supports two levels of folders; a top level folder and an operator folder. All operators defined in the class will be under a subfolder of the top level folder defined for the class..

- **Method annotations**
  - `@Description` - The text that describes the operator in the Rule operator tooltip or the description of the service callout in the Ruleflow properties. Note that this is typically the only annotation type used withh service callouts.

- @OperatorFolder - The name of the subfolder, under top level folder, for the operator defined by the method.

- **Parameter annotations**

  - @ArgumentName - The name of the argument shown in the function signature part of the tool tip.

Localizing Annotations

Annotations offer a format that allows localization. In its basic format, you can just enter @Annotation("text").

You can choose to add a list of one or more locales with corresponding strings for each locale. For example:

```
@Annotation(lang={"en","fr"}, values={"text","texte"})
```

Creating multi-line annnotations

Some annotations provide the help that the user sees when they hover over an operator in the **Rule Operator** tab. Often the description can be improved by adding line returns

You can embed line returns in your description by using the "\n text" + convention, as shown:

```
@Description(lang = { "en" }, values = { "
    "\n Replace all occurrences of a substring" +
    "\n within a string" +
    "\n with another string."
    })
```

The following excerpt from AttributeOperators.java highlights usage of Corticon extension annotations:

```
...
@TopLevelFolder("Sample Extended Operators")
public class AttributeOperators implements ICcDecimalExtension,
  ICcStringExtension, ICcDateTimeExtension {

 @OperatorFolder(lang = { "en" }, values = { "String" })
 @Description(lang = { "en" }, values = { "Replace all occurences of a substring with a
 string with another string." })
 public static String replaceAll(String s,
    @ArgumentName(lang = { "en" }, values = { "searchString" }) String searchString,
   @ArgumentName(lang = { "en" }, values = { "replacement" }) String replacement) {
  ...
 }
}
```

Accessing HTTP Headers in Extended Operators and Service Callouts

Access to Vocabulary Metadata through the ICcDataObjectManager

When deployed as a web service, the Corticon server can retrieve all HTTP header name/value pairs from the HTTP session. These will be placed in a class implementing the ICcServerHttpInfo interface. This class is available from the ICcDataObjectManager class passed to all extensions.

An example of a service callout accessing the HTTP headers is as follows:

```
public static void getHttpHeaders(ICcDataObjectManager dataObjMgr)
{
    ICcServerHttpInfo ccServerHttpInfo = dataObjMgr.getCcServerHttpInfo();
    Map<String, String> httpHeaders = ccServerHttpInfo.getHttpHeaders();
    for (Map.Entry<String, String> entry : httpHeaders.entrySet())
    {
        String key = entry.getKey();
        String value = entry.getValue();
        .......
    }
}
```

One use of this mechanism is to pass a user security token from the client making the request to the decision service where an extension can use the token to access an external data source as the client. This is useful when Corticon is behind a SaaS or other multi-tenant service and a decision service needs to access external data as a specific tenant. Corticon simply makes the HTTP headers available, their interpretation is dependent on the extension that uses them.

See the Corticon extension JavaDoc for more details.

Imports and interfaces used in extensions

Annotations

Extended Operators - Extended operators load four annotation types:

```
import com.corticon.services.extensions.ArgumentName;
import com.corticon.services.extensions.Description;
import com.corticon.services.extensions.OperatorFolder;
import com.corticon.services.extensions.TopLevelFolder;
```

Service Callouts – Service callouts load one annotation type:

```
import com.corticon.services.extensions.Description;
```

Interfaces

Extended Operators – The interfaces added are those required for the data types used in the extended operator class, selected from the following:

```
import com.corticon.services.extensions.ICcCollectionExtension;
import com.corticon.services.extensions.ICcDateTimeExtension;
import com.corticon.services.extensions.ICcDecimalExtension;
import com.corticon.services.extensions.ICcIntegerExtension;
import com.corticon.services.extensions.ICcSequenceExtension;
import com.corticon.services.extensions.ICcStandAloneExtension;
import com.corticon.services.extensions.ICcStringExtension;
```

Note: While extended operators are limited to returning a value, the standalone extended operator type can access the interfaces for `ICcDataObject`:

```
import com.corticon.services.extensions.ICcStandAloneExtension;
import com.corticon.services.dataobject.ICcDataObject;
import com.corticon.services.dataobject.ICcDataObjectManager;
```

When these interfaces are loaded, extended standalone operator methods can define `ICcDataObjectManger` as their first parameter. This provides flexibility in integrating Corticon with databases and other external services to perform complex actions, such as retrieving a set of records from a web service, and then adding them as associations on an entity. The `ICcDataObjectManager` parameter is not allowed in rule syntax.

Service Callouts – A service callout adds the following interface:

```
import com.corticon.services.extensions.ICcServiceCalloutExtension;
```

Extended operator data type mappings

The mapping of parameter and return types for Extended Operators are as follows:

Java Type	Corticon Type
java.math.BigInteger	Integer
java.math.BigDecimal	Decimal
java.lang.Boolean	Boolean
java.util.Date	Date, Time or DateTime
java.lang.String	String

Serializing Response and RuleMessages in Corticon extensions

Service Callouts and Extended Operators can retrieve a serialized representation of the Response and Rule Messages resulting from the execution of a decision service. Extensions can retrieve this data to perform custom logging or alerting which includes the Response or Rule Messages. Use the following methods on the `ICcDataObjectManger()` interface:

```
 /**
 * Returns the current response with regards to the execution.
 * Whether the payload is XML, JSON, or Business Objects,
 * the payload will be converted to a String and returned.
 * @return String
 */
String getCurrentResponse();

 /**
 * Returns the current Rule Messages with regards to the execution.
 * Whether the payload is XML, JSON, or Business Objects,
 * the payload will be converted to a String and returned.
 * @return String
```

```
 */
String getCurrentRuleMessages();
```

The String that is returned conforms to the type of the execution: XML, JSON, or Business Objects.

EXAMPLES

XML

XML payload into the Decision Service yields the following Strings. This String can be import into a JDOM Document if the user chooses to do so.

String getCurrentResponse():

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonResponse xmlns="urn:Corticon"  decisionServiceName="myDecisionService">
    <WorkDocuments>
       <Customer id="Customer_id_1">
          <id>1</id>
          <status>APPROVED</status>
       </Customer>
    </WorkDocuments>
</CorticonResponse>
```

String getCurrentRuleMessages():

```
<?xml version="1.0" encoding="UTF-8"?>
<Messages xmlns="urn:Corticon" version="0.0">
   <Message>
      <text>[Text associated with RuleMessage]</text>
      <severity>[INFO, WARNING, VIOLATION]</severity>
      <entityReference>[Entity reference that this RuleMessage is
assigned]</entityReference>
   </Message>
   <Message>
      ...
   </Message>
</Messages>
```

JSON

JSON payload into the Decision Service yields the following Strings. This String can be imported into a JSONObject as apppropriate.

String getCurrentResponse():

```
{
    "Objects": [{
        "id": 1,
        "status": "APPROVED",
        "__metadata": {
            "#type": "Customer",
            "#id": "Customer_id_1"
        }
    }],
    "__metadataRoot": {
    }
}
```

```
String getCurrentRuleMessages():
```

```
 {
 "Messages": {
       "version": "0.0",
       "Message": [
           {
              "severity": "<INFO, WARNING, VIOLATION>",
              "text": "<Text associated with RuleMessage>",
              "entityReference": "<Entity reference that this RuleMessage is assigned>",
              "__metadata": {"#type": "#RuleMessage"}
           },
           {
              "severity": "<INFO, WARNING, VIOLATION>",
              "text": "<Text associated with RuleMessage>",
              "entityReference": "<Entity reference that this RuleMessage is assigned>",
              "__metadata": {"#type": "#RuleMessage"}
           }
       ]
    },
    "__metadata": {"#type": "#RuleMessages"}
 }
```

Business Objects

For Business Object payloads, the Server will call the `toString()` method on the BOs that have been added to the `java.util.Collection` or `java.util.Set`, which is passed into the `execute()` method.

```
String getCurrentResponse()
```

Note: Meaningful results will only be seen if the `toString()` method is overridden in each BO .

For RuleMessages, the Server calls `ICRuleMessages.toString()` on the object. Corticon has control of this format because it overrides the `toString()` method.

```
String getCurrentRuleMessages():
```

```
 ...
 [Message[<hashcode of CcRuleMessage object>]
     entityReference[<Entity reference that this RuleMessage is assigned>]
     severity[<INFO, WARNING, VIOLATION>]
     text[<Text associated with RuleMessage>],
 Message[hashcode of CcRuleMessage object]
     entityReference[<Entity reference that this RuleMessage is assigned>]
     severity[<INFO, WARNING, VIOLATION>]
     text[<Text associated with RuleMessage>]
  ]
```

5

Using DataDirect drivers

If you need custom database access beyond that provided by Corticon's Enterprise Data Connector (EDC) or Advanced Data Connector (ADC) you can now use the DataDirect® drivers bundled with Corticon in your extensions and wrappers. Corticon provides a new factory method for getting a connection to a database. It connects to a database using a DataDirect driver and returns a standard `java.sql.Connection` object. You work with this Connection object the same as you would any Connection in Java.

The `ICcDataObjectManager` class now provides a method to retrieve an instance of `IDatabaseDriverManager`. This new class provides the `getConnection(…)` method that can be used to create a database connection using a bundled DataDirect driver. See the Corticon JavaDoc for details on these classes and methods.

Get a connection

The class `CcDatabaseConnectionFactory` opens a connection to the database and processes queries. It contains a method that returns a `java.sql.Connection` interface to open a connection using the DataDirect driver and returning it. The `getConnection()` method returns a `java.sql.Connection` from these signatures:

```
getConnection(String dataSourceId, String driverId, String connectionString, String
username, String password)
```

- String: `dataSourceId` - The JNDI dataSource Id. This value is used to lookup an existing connection.

- String: `driverID` - The Id (as outlined in `DatabaseDefinitions.xml`) for the DataDirect driver.

- String: `ConnectionString` - Driver connection string, in the same format as EDC connection in Corticon Studio. For example, `jdbc:progress:openedge://hostname:5566;databaseName=corticon`

- String: `username` - username for logging into the database.

- String: `password` - password for logging into the database.

```
getConnection(String dataSourceId, String driverId, String host, int Port, String
databaseName, String username, String password)
```

- String: `dataSourceId` - The JNDI dataSource Id. This value is used to lookup an existing connection.

- String: `driverID` - The Id (as outlined in `DatabaseDefinitions.xml`) for the DataDirect driver.

- String: `host` - The hostname of the database server for connection.

- Int : `Port` - Port number of the database server for connection.

- String: `databaseName` - the name of the database for connection on the server .

- String: `username` - username for logging into the database.

- String: `password` - password for logging into the database.

```
getConnection(String dataSourceId)
```

- String: `dataSourceId` - The JNDI dataSource Id. This value is used to lookup an existing connection.

```
getConnection(Properties connectionProperties)
```

- Properties: *properties* - Object passed with each of the above items as fields. This also lets you specify additional connection parameters. Constants for the properties fields will be supplied.

Close a connection

When you have completed processing requests, you can either:

- Close the connection using the `close()` method in the `Connection` interface. In some cases, you want to immediately release a connection's database and JDBC resources instead of waiting for them to be automatically released; the `close()` method provides this immediate release.

- Leave the connection up when connection pooling is being used.

- Leave the connection open until the JVM exits or the class gets garbage collected.

6

How to create extensions

For details, see the following topics:

- Import the Corticon APIs into the Java project

- Creating custom extended operators

- Creating custom service callouts

- Add the compiled Java classes to the Rules Project

Import the Corticon APIs into the Java project

The Extended Operator and Service Callout samples contain Java projects demonstrating how to create a Java extension. These are standard Java projects. Each references the Corticon JAR file that defines its API, `CcExtensionApi.jar`, located in a Studio installation at `[CORTICON_HOME]/Studio/lib/`. The JAR is added to the project from its Corticon installed location to the project's build path using the predefined Eclipse variable `CORTICON_HOME`. For example:
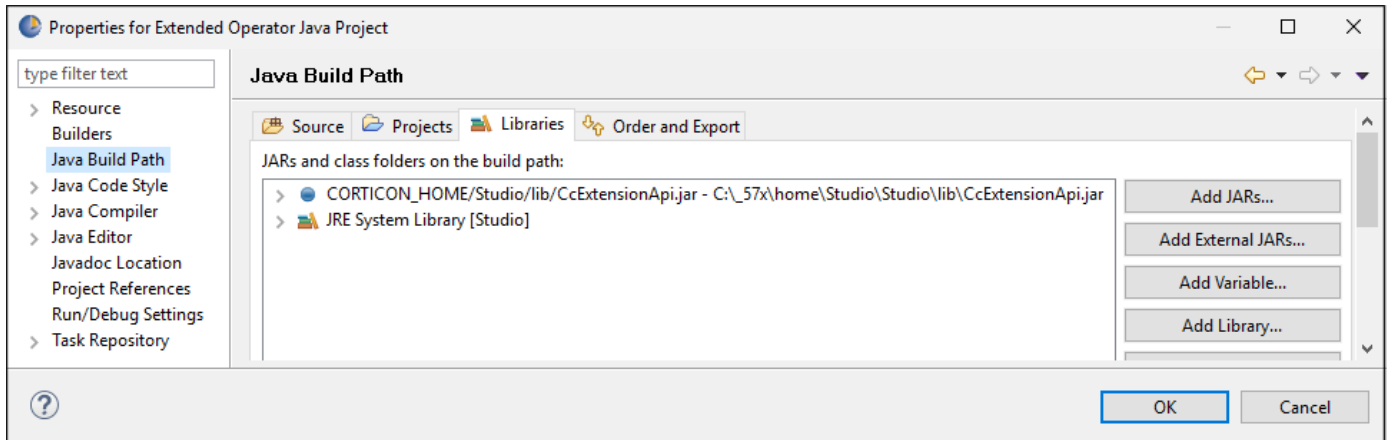
In your Java project, import the Corticon APIs as described, then create your Java source files. Build the Java project by right-clicking on the project name, and then choosing **Export**. In the Export Dialog, choose **Java > Jar file**. Enter a destination location for the JAR file, then choose appropriate options, and then click **Finish**.

Creating custom extended operators

Note: Corticon Studio is built on Eclipse which provides a Java development environment you can use for creating Corticon extensions that you can use in current and future versions of Corticon. If you want to create extensions in a separate IDE, you must use Java 1.7.0_05 or higher.

Note: Compatibility of extensions created in an earlier release, any extension operators and service callouts that are in `extended.core.jar` are shared across all Rule Projects. As a result, such extensions are always in the **Rule Operator** tab in every editor. Then, you can add your extended operators and service callouts to specific Rule Projects using the new mechanism.

Note: You might want to simply copy the sample project **Extended Operator Java Project**, and then tweak a sample such as `AttributeOperators.java` by renaming the `TopLevelFolder` to `mySampleExtendedOperators` for your first run. You can then build the Java project.

For many developers, the quickest way to learn is by example. You might want to compare the three Java source files in the **Extended Operator Java Project** to see what is common and what changes. In this example, the `AttributeOperators.java` is presented.

1. Specify the imports and interfaces you will need.

```
package com.corticon.samples.extensions;

import java.util.Calendar;
import java.util.Date;

import com.corticon.services.extensions.ArgumentName;
import com.corticon.services.extensions.Description;
import com.corticon.services.extensions.ICcDateTimeExtension;
import com.corticon.services.extensions.ICcStringExtension;
import com.corticon.services.extensions.OperatorFolder;
import com.corticon.services.extensions.TopLevelFolder;
```

This class imports the Corticon `ICcStringExtension` and `ICcDateTimeExtension` interfaces because it will implement extended operators for String and DateTime attributes. The other Corticon imports are for the annotations which will be used to describe the extensions.

2. Enter your comments that describe the class.

```
/**
 * This class provides sample Corticon stand-alone extended operators.
 * Extended operators are a means to add custom features to Corticon for
 * use in Corticon rules.
 *
 * The samples in this class provide simple operators for calculating the
 * present and future value of an investment for a number of years at a given
 * interest rate.
 */
```

A general description of this source file is always good coding practice. It has no use outside of the source file.

3. Specify the `TopLevelFolder` name.

```
@TopLevelFolder("Sample Extended Operators")
```

The `TopLevelFolder` annotation identifies the folder that will group the extended operators on the **Rule Operators** tab in Corticon Studio. You can name the folder to fit your needs, such as "My Operators", or "Financial Operators".

4. Specify the class and its implementations.

```
public class AttributeOperators implements ICcStringExtension, ICcDateTimeExtension {
```

5. Name your operator folder, and use the locale parameters if appropriate.

```
@OperatorFolder(lang = { "en" }, values = { "Date" })
```

The `OperatorFolder` defines the subfolder that will list an individual operator within the `TopLevelFolder` on the **Rule Operators** tab in Corticon Studio. You can organize and name folders to fit your needs.

6. Add your description of the operator, and use the locale parameters if appropriate..

```
@Description(lang = { "en" }, values = { "Returns true if the date is in a leap year."
})
```

The `Description` annotation describes the specific operator. The hover help reveals what is passed, what is returned, and description text for the locale.

7. Write your actual implementation of the extended operator. It is always `public static`.

```
public static Boolean isLeapYear(Date d) {
 if (d == null)
  return null;

 Calendar c = Calendar.getInstance();
 c.setTime(d);

 int year = c.get(Calendar.YEAR);
 if ((year % 400 == 0) || ((year % 4 == 0) && (year % 100 != 0)))
  return true;
 else
  return false;
}
```

The example takes a date and returns a boolean. It is up to you to determine that this produces the result you want, and that you can verify it across a range of values and error conditions.

8. The sample includes another operator definition using the same structure, this one for the **String** `OperatorFolder`. You can similarly change this section, or just cut the whole section out.

```
/**
 * Replaces all occurrences of a substring in a string with another string.
 *
 * @param s A string.
 * @param searchString The substring to look for in s.
 * @param replacement The string to replace it with.
 * @return The original string with all instances of searchString replace by
 * replacement.
 */
@OperatorFolder(lang = { "en" }, values = { "String" })
@Description(lang = { "en" }, values = { "Replace all occurrences of a substring within
a string with another string." })
public static String replaceAll(String s,
  @ArgumentName(lang = { "en" }, values = { "searchString" }) String searchString,
  @ArgumentName(lang = { "en" }, values = { "replacement" }) String replacement) {
 if (s == null)
  return null;

 if (searchString == null)
  return s;

 String r = s.replaceAll(searchString, replacement);
 return r;
 }
 }
```

9. Save your work, and then go ahead to build the Java project by right-clicking on the project name, and then choosing **Export**. In the Export Dialog, choose **Java > Jar file**. Enter a destination location for the JAR file, then choose appropriate options, and click **Finish**.

10. In order for the extension to get referenced by a Rules project, and then packaged with a Decision Service, you must Add the compiled Java classes to the Rules Project.

11. Create and run Ruletests that evaluate the range of possible values that could be presented to the extension. Be sure to include blanks and nulls so that you get complete coverage.

Creating custom service callouts

Service Callout (SCO) extensions are user-written functions that can be invoked in a Ruleflow.

In a Ruleflow, the flow of control moves from Rulesheet to Rulesheet, with all Rulesheets operating on a common pool of facts. This common pool of facts is retained in the rule execution engine's working memory for the duration of the transaction. Connection arrows on the diagram specify the flow of control. Each Rulesheet in the flow may update the fact pool.

When you add a Service Callout (SCO) to a Ruleflow diagram, you effectively instruct the system to transfer control to your extension class at a specific point in the flow. Your extension can directly update the fact pool, and your updated facts are available to subsequent Rulesheets.

Consider the sample:

The rule flow uses two service callouts (`Get Patient Info` and then `Get Procedure History`), then uses the data in the `Determine Approval` Rulesheet, and finally passes control to Service Callout extension class `Save Approvals`.

Your Service Callouts use the Progress Corticon Extension API to retrieve and update facts. The package `com.corticon.services.dataobject` contains two Java interfaces:

Interface	Purpose
`com.corticon.services.dataobject.ICcDataObjectManager`	Provides access to the entire fact pool. Allows you to create, retrieve and remove entity instances.
`com.corticon.services.dataobject.ICcDataObject`	Provides access to a single entity instance. Allows you to update the entity instance, including setting attributes and creating and removing associations.

Your Service Callout class must implement marker interface `ICcServiceCalloutExtension`.

Here is the source code for the service callout `MedicalRecords.java`:

```
/*
 * Copyright (c) 2016 by Progress Software Corporation. All rights reserved.
 */

package com.corticon.samples.extensions;

import java.util.ArrayList;
import java.util.Set;

import com.corticon.services.dataobject.ICcDataObject;
import com.corticon.services.dataobject.ICcDataObjectManager;
import com.corticon.services.extensions.Description;
import com.corticon.services.extensions.ICcServiceCalloutExtension;
import com.corticon.samples.simulation.*;

/**
 * This class provides sample Corticon service callouts. Callouts provide
 * a means to add custom features to Corticon for use in Corticon ruleflows.
 * Callouts are for integrating with external systems such as webservices.
 * In a callout you can create and delete instances of Corticon entities,
```

```
         * set their properties and define associations.
         *
         * The samples in this class provide a simulation of retrieving patient
         * medical data given a patient id. The class MedicalDataSimulator provides
         * a static set of patient and procedure data. In a real implementation
         * this could be a class which gets data from a webservice, database, or
         * other source.
         *
         */
 public class MedicalRecords implements ICcServiceCalloutExtension {

  private static MedicalDataSimulator md = new MedicalDataSimulator();

  /**
   * Service callout to retrieve patient descriptive information for
   * the set of Corticon "Patient" entities.
   *
   * This callout demonstrates how to iterate over a set of Corticon
   * entities and set attributes on them.
   */
  @Description(lang = { "en" }, values = { "Service callout to retrieve patient descriptive
  information for the set of Corticon 'Patient' entities." })
```

Service Callout methods must be declared `public static`.

The system will recognize your Service Callout method if the method signature takes one parameter and that parameter is an instance of `ICcDataObjectManager`.

```
  public static void getPatientInfo(ICcDataObjectManager dataObjMgr) {
   // Get the set of "Patient" entities.
   Set<ICcDataObject> patients = dataObjMgr.getEntitiesByName("Patient");
   if (patients != null) {
    // Process each patient in the set.
    for (ICcDataObject patient : patients) {
     // Get the "id" attribute of the current patient.
     Long id = (Long) patient.getAttributeValue("id");

     if (id != null) {
      // Get the simulated information for this patient.
      PatientData pd = md.getPatientData(id.intValue());

      if (pd != null) {
       // Set patient information as entity attributes.
       patient.setAttributeValue("firstName",pd.getFirstName());
       patient.setAttributeValue("lastName", pd.getLastName());
      }
     }
    }
   }
  }

  /**
   * Service callout to retrieve history of medical procedures for the
   * set of Corticon "Patient" entities.
   *
   * This callout demonstrates how to create new Corticon entities and
   * associate them with existing Corticon entities.
   */
  @Description(lang = { "en" }, values = { "Service callout to retrieve history of medical
  procedures for the set of Corticon 'Patient' entities." })
  public static void getProcedureHistory(ICcDataObjectManager dataObjMgr) {
   // Get the set of "Patient" entities.
   Set<ICcDataObject> patients = dataObjMgr.getEntitiesByName("Patient");
   if (patients != null) {
    // Process each patient in the set.
    for (ICcDataObject patient : patients) {
     // Get the "id" attribute of the current patient.
```

```
            Long id = (Long) patient.getAttributeValue("id");

            if (id != null) {
             // Get the simulated information for this patient.
             PatientData pd = md.getPatientData(id.intValue());

             // Get the medical procedure history for this patient.
             ArrayList<ProcedureData> td = pd.getProcedureRecords();

             // Add procedures to the patient entity
             for (ProcedureData r : td) {
              // Create a new "Procedure" entity.
              ICcDataObject p = dataObjMgr.createEntity("Procedure");

              // Set attributes on this entity.
              p.setAttributeValue("code", r.getCode());
              p.setAttributeValue("description", r.getDescription());
              p.setAttributeValue("date", r.getDate());

              // Associate it with the current patient.
              patient.addAssociation("procedure", p);
             }
            }
          }
        }
    }

    /**
     * Service callout to save the approval state for each medical procedure
     * for a set of Corticon "Patient" entities.
     *
     * This callout demonstrates how to iterate over a set of entities and
     * associations to get the value of attributes.
     */
    @Description(lang = { "en" }, values = { "Service callout to save the approval state
 for each medical procedure for a set of Corticon 'Patient' entities." })
    public static void saveProcedureApproval(ICcDataObjectManager dataObjMgr) {
     // Get the set of "Patient" entities.
     Set<ICcDataObject> patients = dataObjMgr.getEntitiesByName("Patient");
     if (patients != null) {
      // Process each patient in the set.
      for (ICcDataObject patient : patients) {
       // Get the "id" attribute of the current patient.
       Long id = (Long) patient.getAttributeValue("id");

       // Process each "Procedure" association on the patient.
       Set<ICcDataObject> procedures = patient.getAssociations("procedure");
       for (ICcDataObject procedure: procedures) {

        // Get the value of the "approved" and "code" attributes on the procedure.
        Boolean approved = (Boolean) procedure.getAttributeValue("approved");
        String code = (String) procedure.getAttributeValue("code");

        // Update the procedure record.
        md.setProcedureApproval(id.intValue(), code, approved);
       }
      }
     }
    }
```
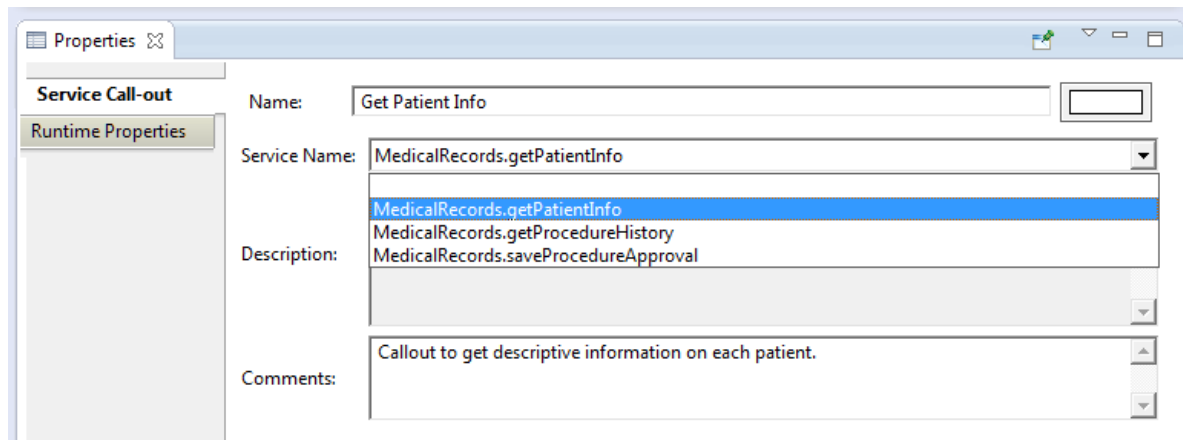
Recognized classes and methods are displayed in the Ruleflow Properties View/Service Name drop-down list when a Service Callout object is on a Ruleflow canvas:

The Service Callout API provides your extension class complete access to the fact pool, allowing you to:

- Find entities in several ways

- Read and update entity attributes

- Create and remove entity instances

- Create and remove associations

Refer to Service Callout Java sample project and the *API Javadocs* for more information.

Access to Vocabulary Metadata

Access to Vocabulary Metadata through the `ICcDataObjectManager`

Extended operators have access to the `ICcDataObjectManager`. This class has long been available to Service Callouts and provides access to metadata such as the Corticon Vocabulary and the entities being processed. To be passed an instance of `ICcDataObjectManager`, the extension class must define method signatures which take `ICcDataObjectManager` as a parameter. See the Corticon JavaDoc for more details.

The method:

```
ICcDataObjectManager.getVocabularyMetadata()
```

Return type:

```
com.corticon.services.metadata.IVocabularyMetadata
```

For details about the `IVocabularyMetadata` object, see the topic *"Accessing the Vocabulary metadata of a Decision Service" in the Deployment Guide.*

Serializing Response and RuleMessages in Corticon extensions

Service Callouts and Extended Operators can retrieve a serialized representation of the Response and Rule Messages resulting from the execution of a decision service. Extensions can retrieve this data to perform custom logging or alerting which includes the Response or Rule Messages. Use the following methods on the `ICcDataObjectManger()` interface:

```
  /**
 * Returns the current response with regards to the execution.
```

```
* Whether the payload is XML, JSON, or Business Objects,
* the payload will be converted to a String and returned.
* @return String
*/
String getCurrentResponse();

 /**
* Returns the current Rule Messages with regards to the execution.
* Whether the payload is XML, JSON, or Business Objects,
* the payload will be converted to a String and returned.
* @return String
 */
String getCurrentRuleMessages();
```

The String that is returned conforms to the type of the execution: XML, JSON, or Business Objects.

EXAMPLES

XML

XML payload into the Decision Service yields the following Strings. This String can be import into a JDOM Document if the user chooses to do so.

```
String getCurrentResponse():
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<CorticonResponse xmlns="urn:Corticon"  decisionServiceName="myDecisionService">
    <WorkDocuments>
       <Customer id="Customer_id_1">
          <id>1</id>
          <status>APPROVED</status>
       </Customer>
    </WorkDocuments>
</CorticonResponse>
```

```
String getCurrentRuleMessages():
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Messages xmlns="urn:Corticon" version="0.0">
   <Message>
      <text>[Text associated with RuleMessage]</text>
      <severity>[INFO, WARNING, VIOLATION]</severity>
      <entityReference>[Entity reference that this RuleMessage is
assigned]</entityReference>
   </Message>
   <Message>
      ...
   </Message>
</Messages>
```

JSON

JSON payload into the Decision Service yields the following Strings. This String can be imported into a `JSONObject` as apppropriate.

```
String getCurrentResponse():
```

```json
{
    "Objects": [{
```

```
            "id": 1,
            "status": "APPROVED",
            "__metadata": {
                "#type": "Customer",
                "#id": "Customer_id_1"
            }
        }],
        "__metadataRoot": {
        }
    }
```

String getCurrentRuleMessages():

```
 {
 "Messages": {
        "version": "0.0",
        "Message": [
            {
                "severity": "<INFO, WARNING, VIOLATION>",
                "text": "<Text associated with RuleMessage>",
                "entityReference": "<Entity reference that this RuleMessage is assigned>",
                "__metadata": {"#type": "#RuleMessage"}
            },
            {
                "severity": "<INFO, WARNING, VIOLATION>",
                "text": "<Text associated with RuleMessage>",
                "entityReference": "<Entity reference that this RuleMessage is assigned>",
                "__metadata": {"#type": "#RuleMessage"}
            }
        ]
    },
    "__metadata": {"#type": "#RuleMessages"}
 }
```

Business Objects

For Business Object payloads, the Server will call the `toString()` method on the BOs that have been added to the `java.util.Collection` or `java.util.Set`, which is passed into the `execute()` method.

String getCurrentResponse()

Note: Meaningful results will only be seen if the `toString()` method is overridden in each BO .

For RuleMessages, the Server calls `ICRuleMessages.toString()` on the object. Corticon has control of this format because it overrides the `toString()` method.

String getCurrentRuleMessages():

```
 ...
 [Message[<hashcode of CcRuleMessage object>]
     entityReference[<Entity reference that this RuleMessage is assigned>]
     severity[<INFO, WARNING, VIOLATION>]
     text[<Text associated with RuleMessage>],
 Message[hashcode of CcRuleMessage object]
     entityReference[<Entity reference that this RuleMessage is assigned>]
     severity[<INFO, WARNING, VIOLATION>]
     text[<Text associated with RuleMessage>]
  ]
```
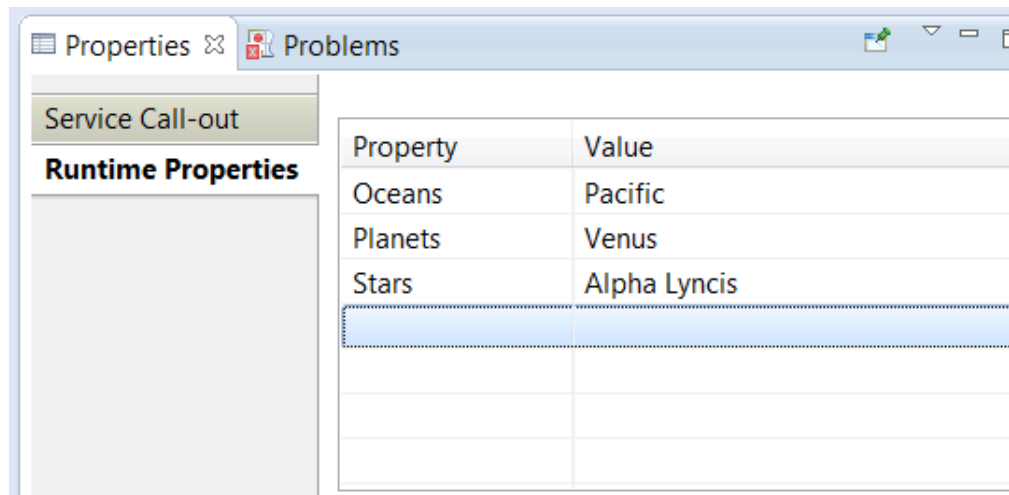
Specifying properties on a service callout instance

You can specify properties on a Service Callout (SCO) that can be set per instance . That means that a SCO that retrieves data from a web service could use multiple instances of it in a Ruleflow where each instance has different parameters. The nature of the parameters is unrestricted; they are simple name/value pairs that a SCO can interpret as needed.

Overview of Service Callout (SCO) parameters

When a SCO is added to a Ruleflow canvas, its **Properties (View) > Runtime Properties** let you set name/value parameter pairs on this SCO instance. These name/value pairs will be passed to the SCO when the SCO is executed. For example:

Properties ⊠ Problems		
Service Call-out	Property	Value
Runtime Properties	Oceans	Pacific
	Planets	Venus
	Stars	Alpha Lyncis

To enable this functionality, the SCO's method must need to accept a `java.util.Properties` object in its method signature:

```
public static void processCreditReport(ICcDataObjectManager aDataObjectManager,
                Properties apropServiceCalloutProperties)
```

If the method does not accept a `Properties` object (as is the case for SCO's created before 5.6.1), the original method will still be called, providing both backward compatibility as well as an alternative approach to using parameters in SCOs.

```
public static void processCreditReport(ICcDataObjectManager aDataObjectManager)
```

If the SCO has implemented both methods, the method with the `Properties` object will be called during execution. If this method does not exist, then the alternative applies.

Selecting the Runtime Properties for a SCO

Defined Property Names and Values

Often you will want to constrain the Property Names and their respective Values to ensure that only valid combinations are selected by the user from a drop-down list box. The Service Callout (SCO) must implement a specific Interface and the following methods for the Ruleflow to list the possible Property Names and their respective Values.

Interface:

```
com.corticon.services.extensions.ICcPropertyProvider
```
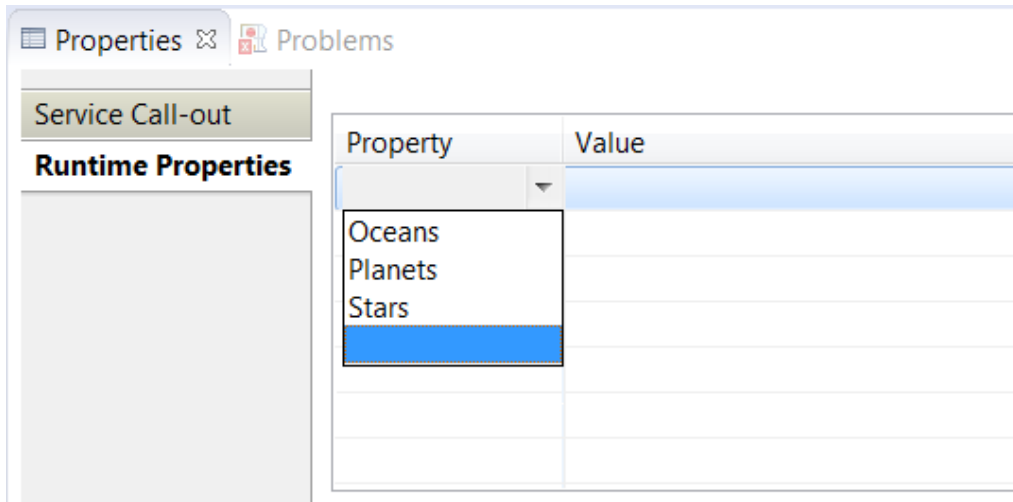
Static Methods:

```
public List<String> getPropertyNameOptions()throws Exception;
```
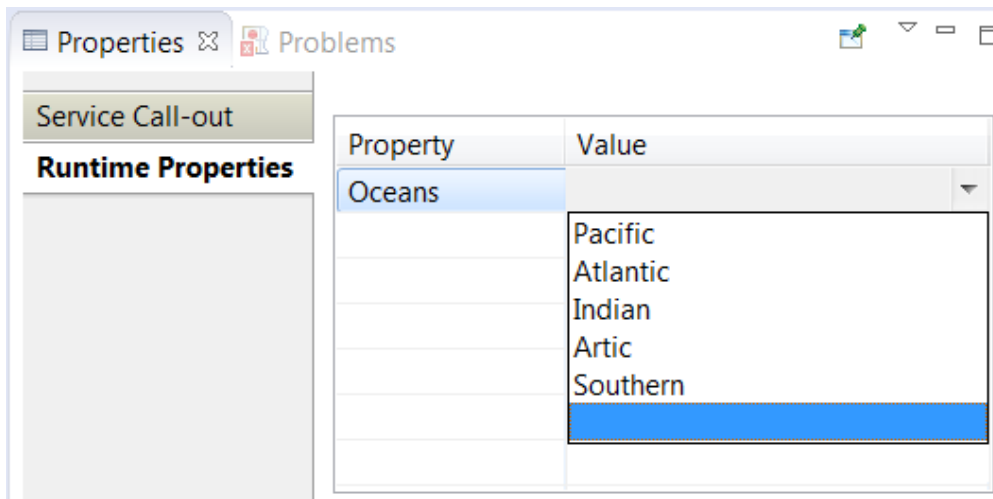
```
public List<String> getProperyValueOptions(String astrPropertyName)throws Exception;
```

Example:

The user drops down the list and then chooses a property name:

The Ruleflow calls back to the SCO to get the possible Values for that Property name, and then lists the values in a drop-down list where the user selects the value:
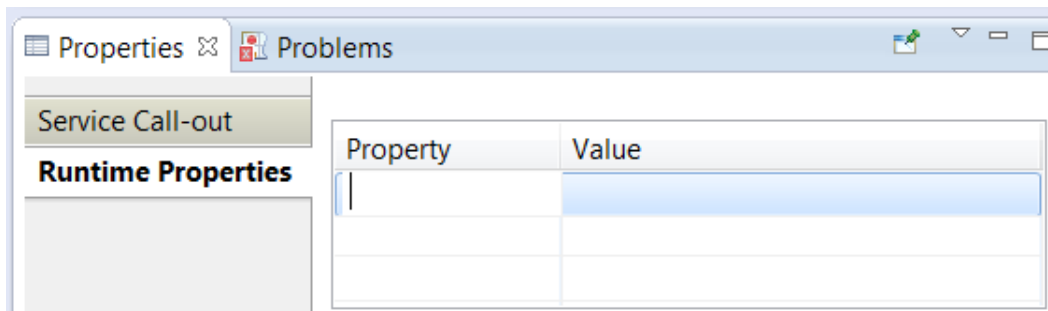

Note: This technique does not allow additional name/value pairs to be entered.
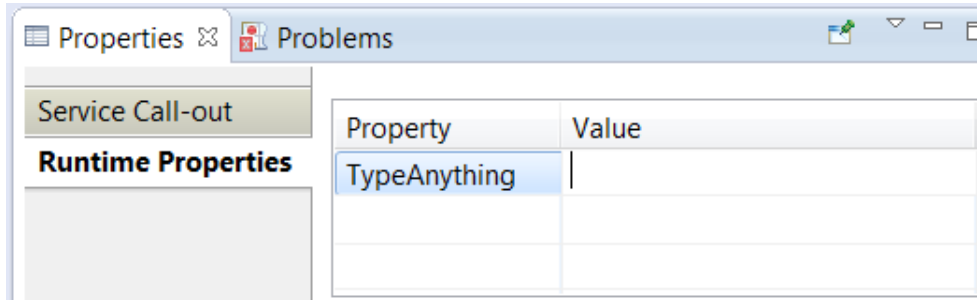
No defined Property Names and Values

Undefined Property Names and Values occur when:

• The SCO does not implement the `ICcPropertyProvider` interface

• The interface and methods are implemented, but the methods return a null or empty list

Under these conditions, the Property Name and Property Value cells in the **Properties (View) > Runtime Properties** are Text Boxes where names and values can be typed on many lines:

There are no values defined for a free-form property name so a value must be typed in:

Note: Property Name and Value lists work independently - While `getPropertyNameOptions()` might return a `List<String>` with values so that the cell on the current **Property** line offers a drop-down list, the selected property might find that its `getPropertyValueOptions(..)` returns a null or empty list. In that case, the **Value** cell is provided as a text box for your free-form entry. However, each property name and value pair must have non-blank entries to complete valid service callout runtime properties.

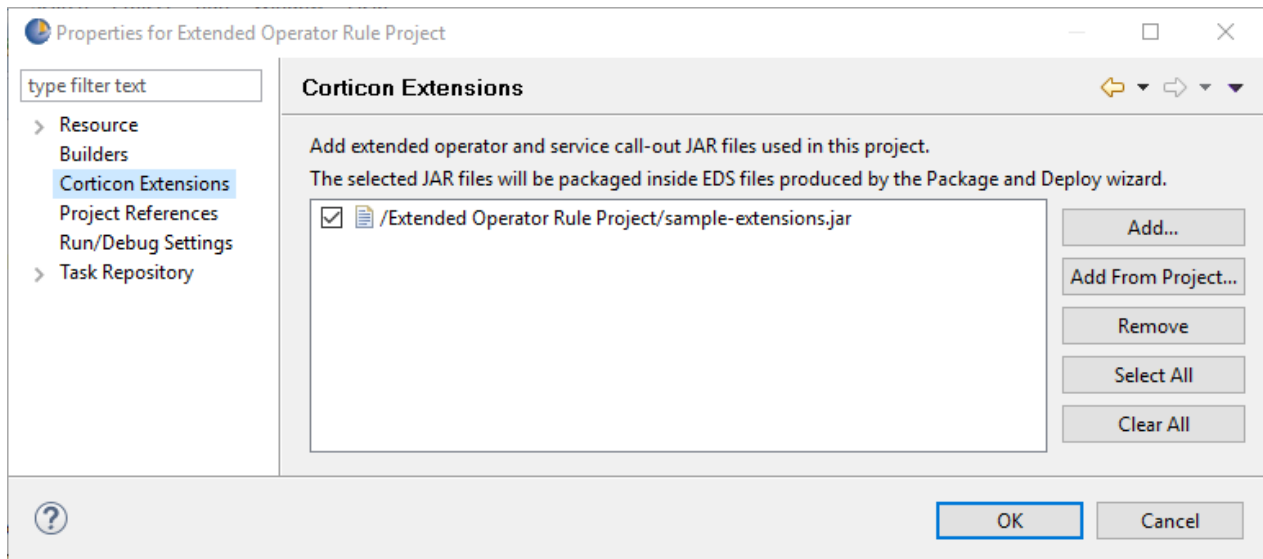### Add the extension to the Rule Project, and then test it.

In order for the extension to get referenced by a Rules project, and then packaged with a Decision Service, you must Add the compiled Java classes to the Rules Project.

Create and run Ruletests that evaluate the range of possible values that could be presented to the extension. Be sure to include blanks and nulls so that you get complete coverage.

Add the compiled Java classes to the Rules Project

To add the Java project's compiled classes to a rules project:

1. Right-click on a project name in the Studio's Project Explorer that requires additional JARs, and then choose **Properties**.

2. Click **Corticon Extensions**.

3. Navigate in the panel to locate and list all JAR files used by the project, as illustrated:

All the listed JARs will be added to compiled EDS as *dependent* JARs, but only the ones that are checked will also be *included* in the compiled EDS file.

4. Click **OK** to save the project properties.

When you use the Studio feature of **Package and Deploy > Save for later Deployment**, the JAR dependencies and inclusions will be added into the **.eds** file.

7

Deploying Decision Services with extensions

Once you have added extension JARs to your project, several deployment tools provide mechanisms to package the extension JARs into deployment. When you compile a project Ruleflow into an EDS file, the extension JARs are encapsulated within the encrypted `.eds`. That insures that regardless how you relocate or update a Decision Service, the extension JARs that are associated with it are consistent.

Deployment from Studio

The three standard techniques in Studio that package and deploy Decision Services all incorporate the extension JARs that were associated with the project:

- Deploying directly to a server
- Deploying to a server through a Web Console application
- Packaging the EDS file locally for access by other tools or the Web Console to complete the deployment.

Deployment using the Server's command line interface

When you use the tool `CorticonManagement` at a server's `[CORTICON_HOME]\Server\bin` location, the `compile` command provides parameters that will declare dependent JARs and then include them. Both parameters take comma-separated values and both parameters are required to achieve the packaging into EDS file.

```
-dj,--dependentjars dependentJar   add jar files required for this decision service
-ij,--includedjars includedJar     add jar files to include in the generated eds file
```

A complete command might look like this:

```
corticonManagement
   --compile
   --input C:\myProject\myRuleflow.erf
   --output C:\myProject\Output
   --service MyDS
   --dependentjars C:\myProject\myExtensions.jar,C:\myProject\myCallouts.jar
   --includedjars C:\myProject\myExtensions.jar,C:\myProject\myCallouts.jar
```

With only required options specified, the result is `C:\myProject\Output\MyDS.eds`

Additions to Ant macro compile arguments

If you want to use Ant macros for the `corticonManagement` command line utilities that are in the file
`[CORTICON_HOME]\Server\lib\corticonAntMacros.xml`, you can set the required extension JARs in
the arguments for the `compile` macro so that you can use them in the call:

```
<attribute name="input" default=""/>
   <attribute name="output" default="" />
   <attribute name="service" default="" />
   <attribute name="version" default="false" />
   <attribute name="edc" default="false" />
   <attribute name="failonerror" default="false" />
   <attribute name="dependentjars" default="" />
   <attribute name="includedjars" default="" />
```

Example of a call to the compile macro:

```
<corticon-compile
   input="${project.home}/Order.erf"
   output="${project.home}"
   service="OrderProcessing"
   dependentjars="${project.home}/myExtensions.jar,${project.home}/myCallouts.jar"
   includedjars="${project.home}/myExtensions.jar,${project.home}/myCallouts.jar"
/>
```

Note: Deploymemt to a Corticon .NET Server - Once a project that includes extension JARs is packaged
into a Decision Service, it deploys and performs as expected on Corticon .NET Server.