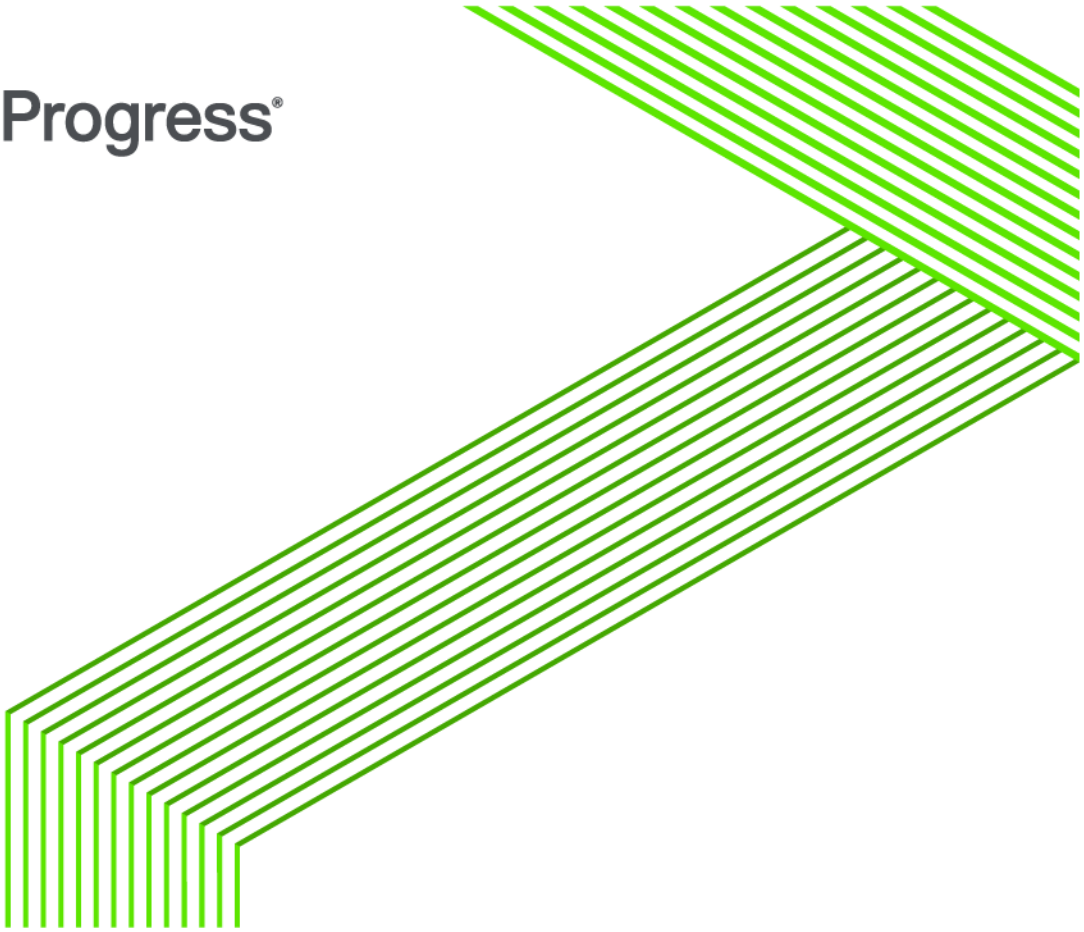
Progress®



Guide to Creating Corticon Extensions

Progress® Corticon®

Notices

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 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. This has been present in previous releases but has been made easier to use in Corticon 5.6.

Extensions are written as custom Java code that you package into one or more JAR files. In earlier releases, you had to modify Corticon's Java classpath and perform other configuration steps to use an extension in your Decision Services. In Corticon 5.6, you simply add extension jar files to your rule project and Corticon bundles them 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. These interfaces have not changed in Corticon 5.6. What is new is the addition of Java annotations to describe the extension. The use of annotations eliminates 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

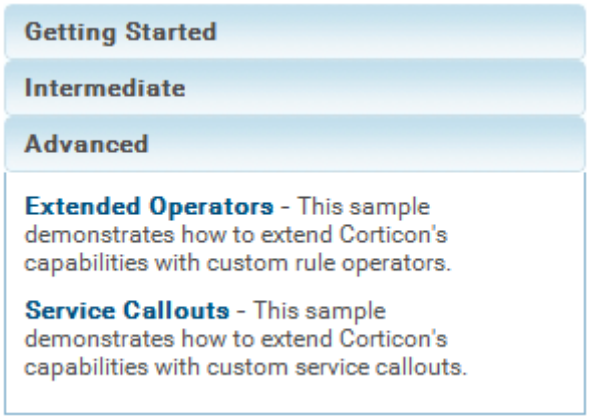
Using extensions when creating Decision Services

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.

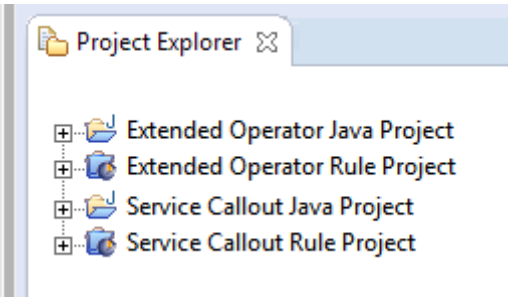
For now, assume that you just want to use the functionality in these extensions.

To use the packaged extension samples in my project

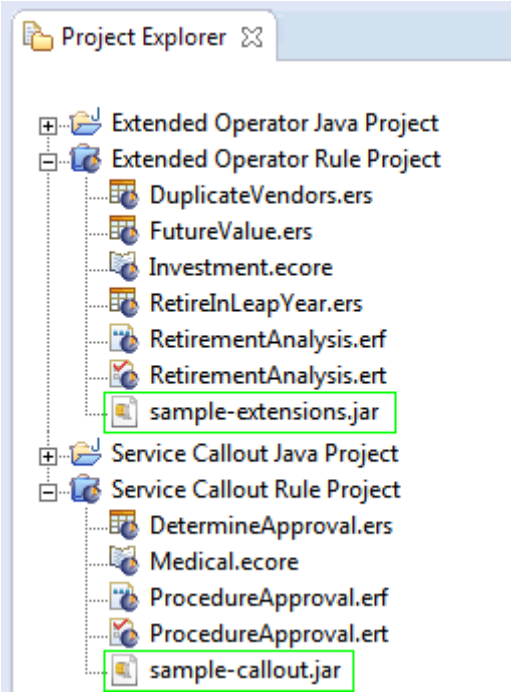
1. In Corticon Studio, choose **Help > Samples**. Locate the **Advanced** samples:



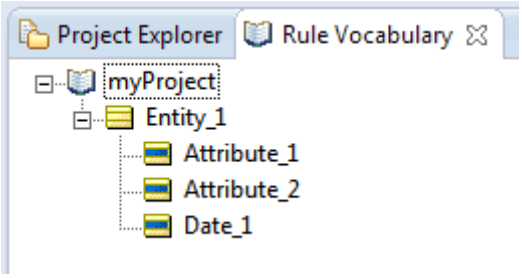
2. Select **Extended Operators**, click **Open**, and then click **OK**.
3. Then do the same to open the **Service Callouts** sample.
4. Your Studio's **Project Explorer** lists the two samples, each with its Java project and its Rules project:



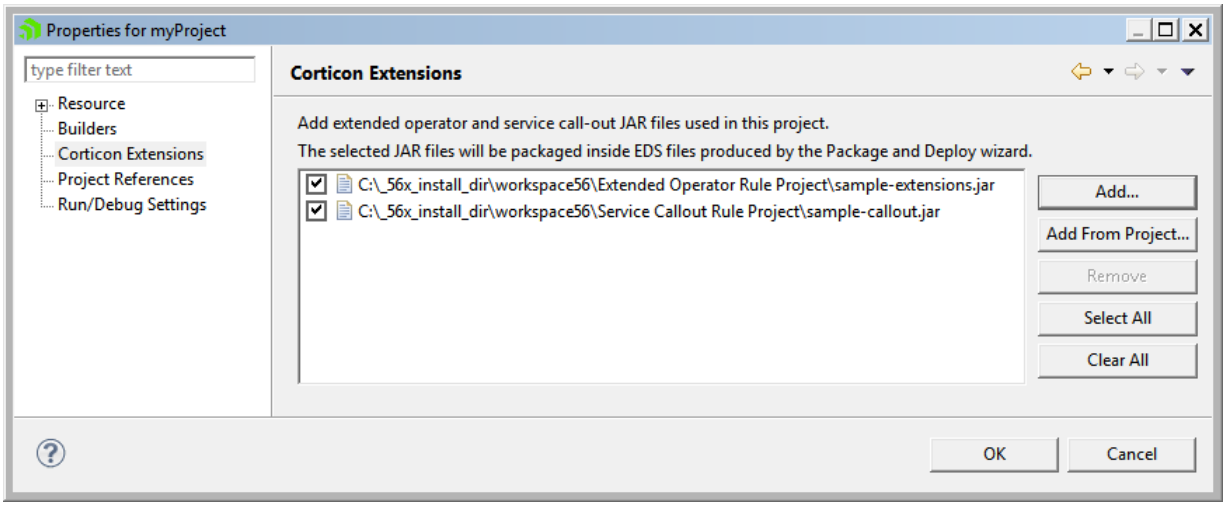
5. Expanding the two Rule Projects, you can see that each has a related samples JAR.



6. Create a new Rule Project named `myProject` and create a very simple Vocabulary that includes a date type:

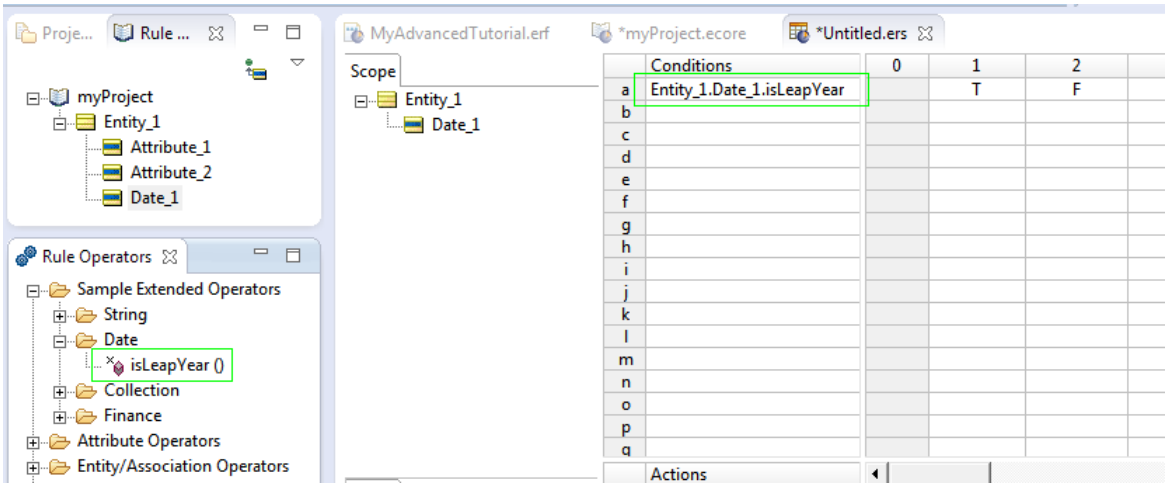


7. In thew Project Explorer, click on `myProject`, and then select **Properties**. Select **Corticon Extensions**. Click **Add** then navigate to each of the sample Rule Projects to select its extensions JAR, as shown:

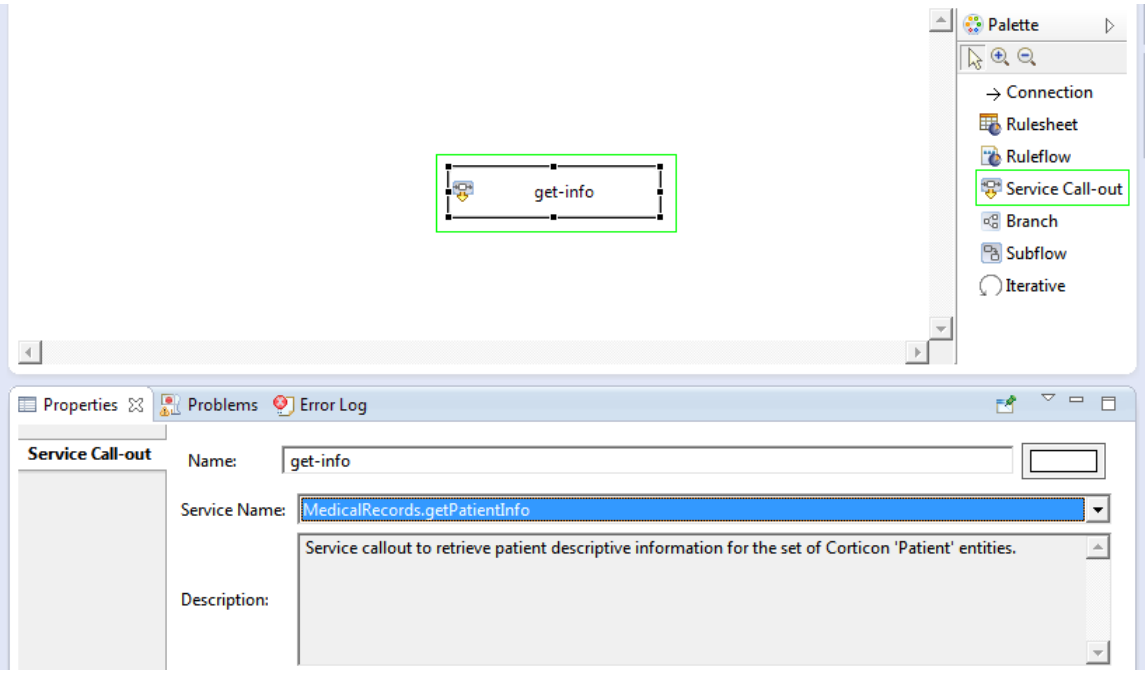


While it is good to copy extension JARs you want to use into your project, you can navigate to other locations to add them to your project

8. Because the JARs were referenced in their original locations, they are not listed in `myProject` folder. If your extensions requires other JARs, you can add them here.
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.

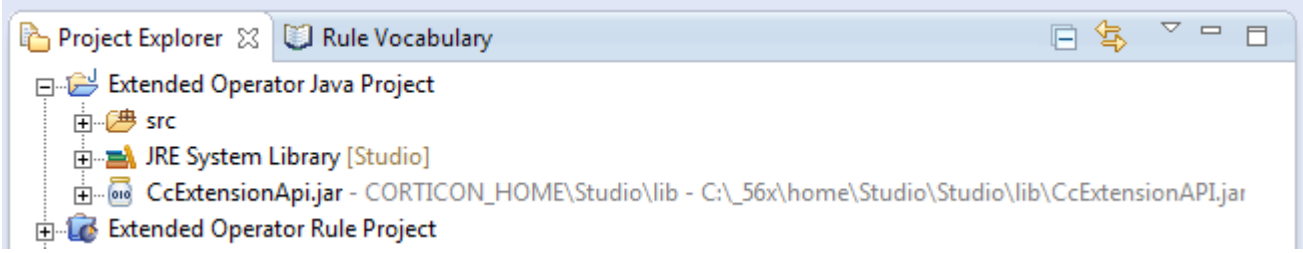
There are two sets of extension samples:

- **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.

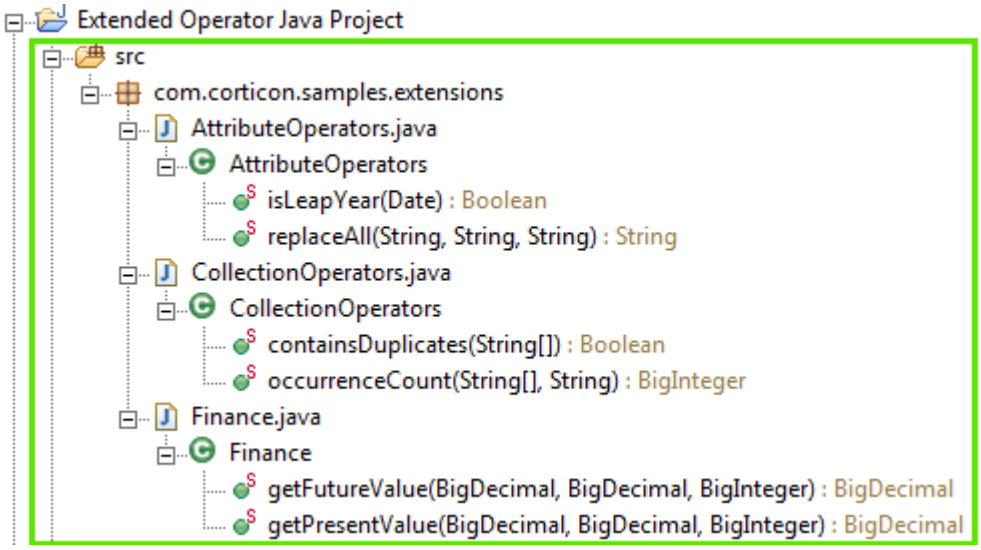
For details, see the following topics:

What's in the Extended Operators Sample Projects

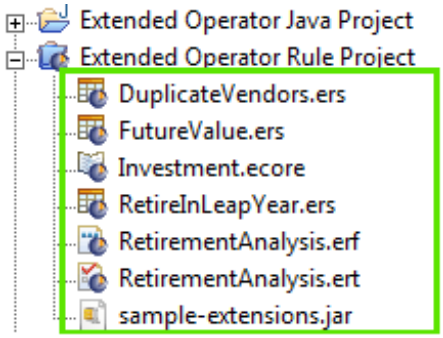
The **Extended Operator Java project** contains a `src` folder with the source code for the extended operators and has on its build path the standard Java system library and `CcExtensionApi.jar` with the Corticon extension API:



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 samples:

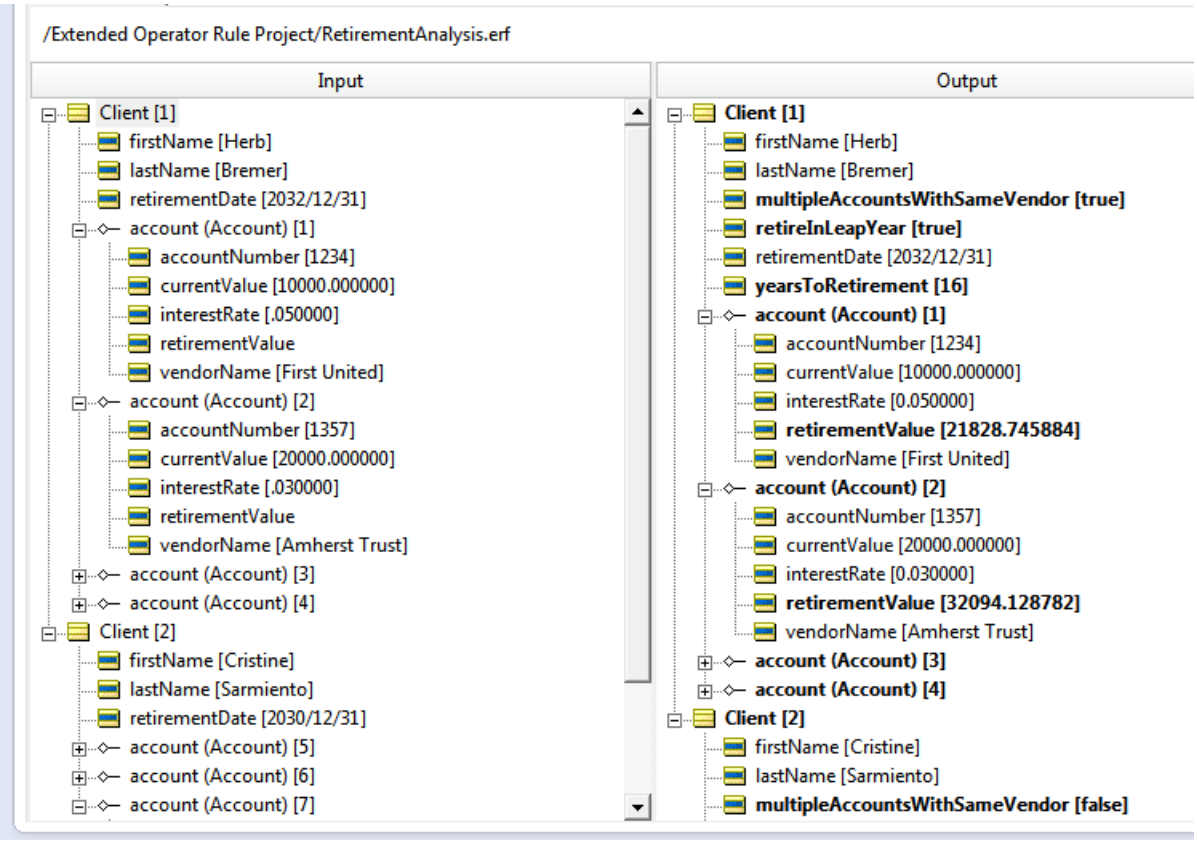


The **Extended Operator Rule project** uses the extended operators and the supporting vocabulary, Ruleflows, and Ruletests for the project. It also contains the JAR file of the extensions built from the Java project, `sample-extensions.jar`.

The sample project contains a Vocabulary, and three Rulesheets that demonstrate each of the new extended operators:

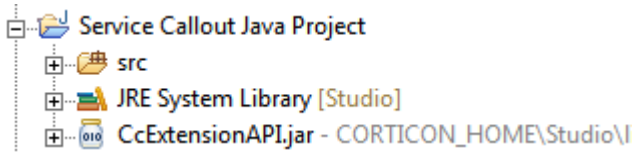
1. FutureValue - Determines years to retirement and the future value of the current investment with a constant interest rate.
2. RetireInLeapYear - Checks whether the retirement year is a leap year
3. DuplicateVendors - Checks to determine that there are not multiple accounts with the same vendor

The Ruleflow chains the three Rulesheets together for the Ruletest to see that the extended operators behave as expected.

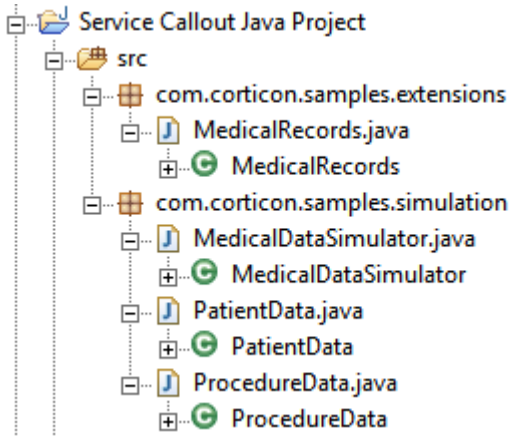


What's in the Service Callout Sample Projects

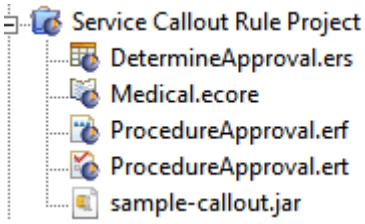
The **Service Callout Java project** contains a `src` folder with the source code for the service callouts and has on its build path the standard Java system library and `CcExtensionApi.jar` with the Corticon extension API:



The `src` folder contains the source code for four Java classes that implement service callouts:



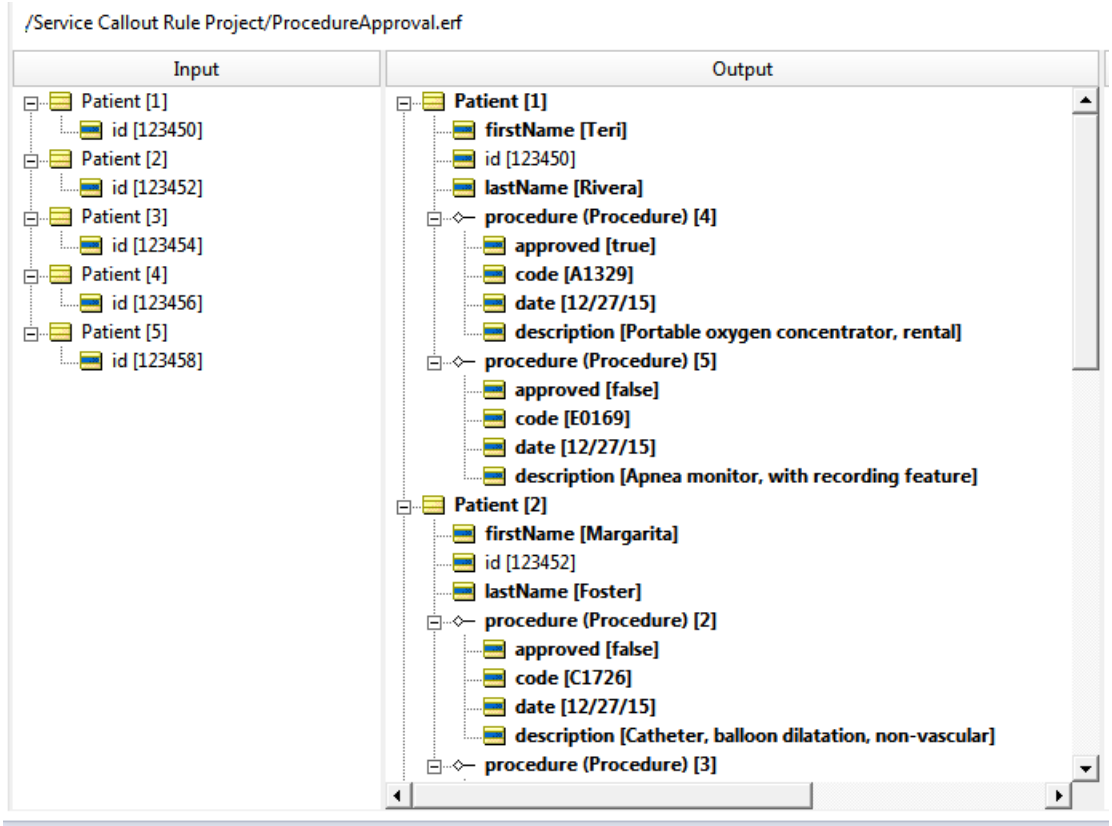
The Service Callout Rule Project is a set of rule assets that demonstrate the Service Callout Java samples:



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

4

Code conventions

For details, see the following topics:

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) {
  ...
 }
}
```

Imports and interfaces used in extensions

Imports

The `com.corticon.services.extension` interfaces for an extended operator loads the 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;
```

While the `com.corticon.services.extension` interfaces for a service callout loads only one annotation type:

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

And then add the interfaces that are required for the data types used in the extended operator class 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;
```

While the service callout adds the following:

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

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

5

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: 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 go ahead to Building the Java classes and JARs on page 31.

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 Building the Java classes and JARs on page 31.

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.

6

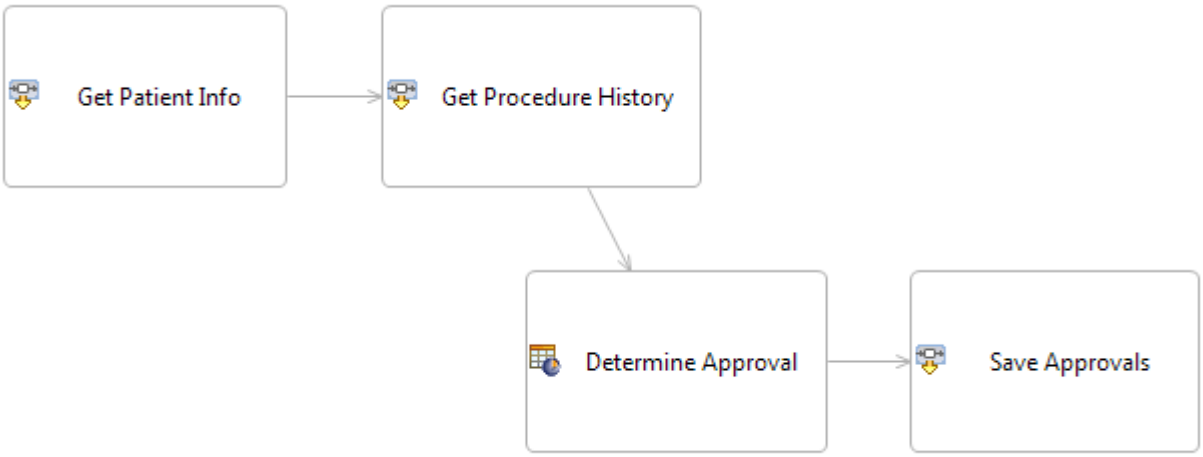
Creating custom service callouts

Service Call-out 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 Call-out to a Ruleflow diagram, you effectively instruct the system to transfer control to your extension class 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 Call-outs 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 Call-out 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." })
```

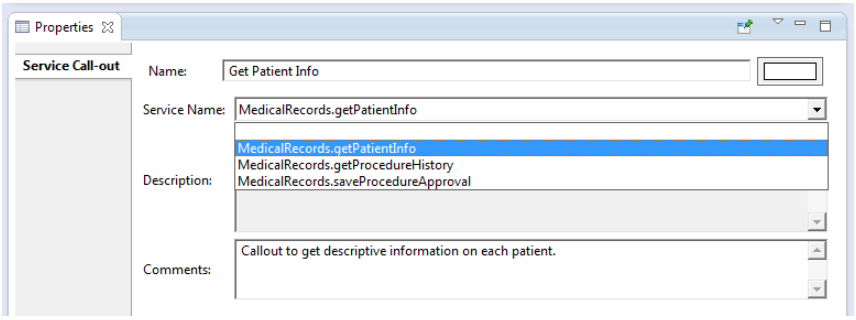
Service Call-out methods must be declared `public static`.

The system will recognize your Service Call-out 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 Call-out object is on a Ruleflow canvas:



The Service Call-out 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

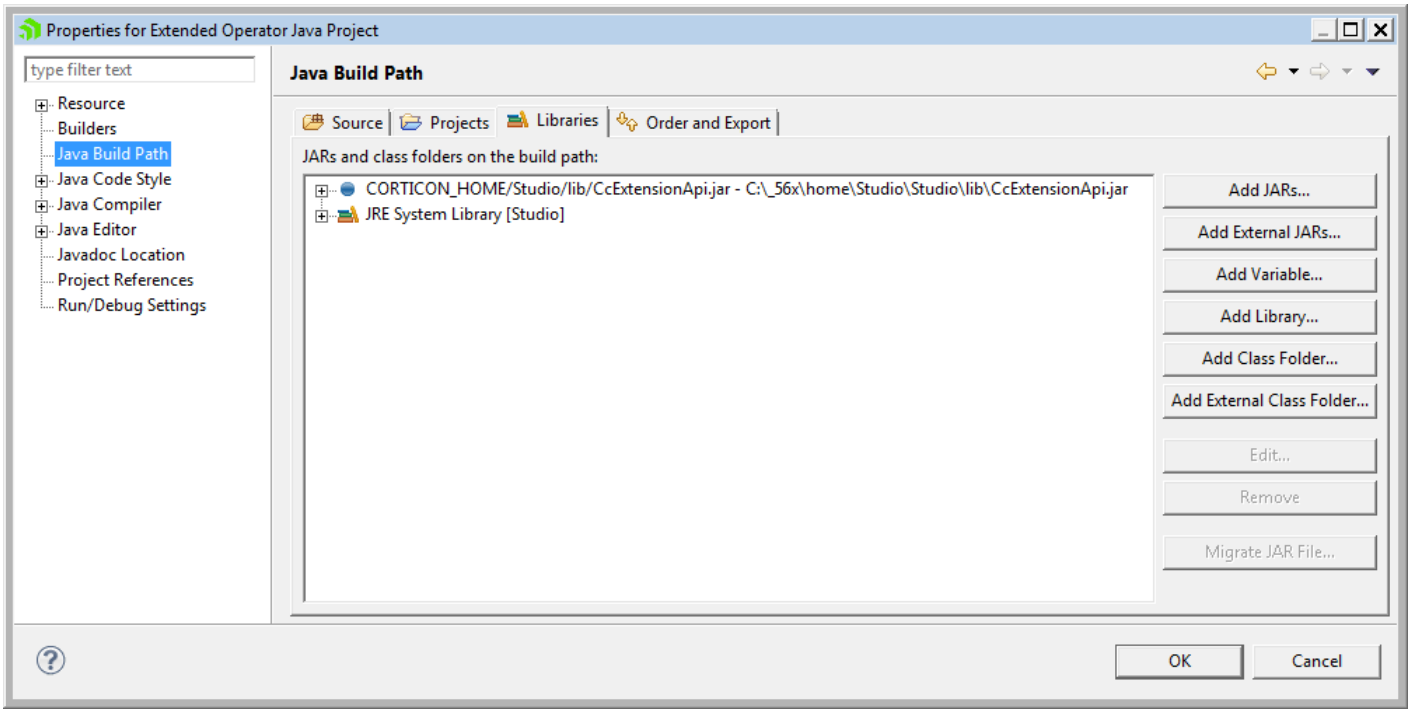
Refer to Service Callout Java sample project and the *API Javadocs* for more information.

7

Building the Java classes and JARs

Both the Extended Operator and Service Callout samples contain Java projects demonstrating how to create a Java extension. These are standard Java projects. Each has the CorticonCorticon JAR file that defines the API for Corticon extensions, `CcExtensionApi.jar` on its build path.

When developing extensions in Corticon Studio, add this JAR to the **Java Build Path** using the predefined Eclipse variable `CORTICON_HOME`. For example:



8

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.

A

Access to Corticon knowledge resources

Complete online documentation for the current release

Corticon online tutorials available in the **Corticon Learning Center**:

- Tutorial: Basic Rule Modeling in Corticon Studio
- Tutorial: Advanced Rule Modeling in Corticon Studio
- Modeling Progress Corticon Rules to Access a Database using EDC
- Connecting a Progress Corticon Decision Service to a Database using EDC
- Deploying a Progress Corticon Decision Service as a Web Service for Java
- Deploying a Progress Corticon Decision Service in process for Java
- Using Corticon Business Rules in a Progress OpenEdge Application

Corticon guides (PDF):

- What's New in Corticon
- Corticon Installation Guide
- Corticon Studio: Rule Modeling Guide
- Corticon Studio: Quick Reference Guide
- Corticon Studio: Rule Language Guide
- Corticon Studio: Extensions Guide
- Corticon Server: Integration and Deployment Guide

- Corticon Server: Web Console Guide
- Corticon Server: Deploying Web Services with Java
- Corticon Server: Deploying Web Services with .NET

Corticon JavaDoc API reference (HTML):

- Corticon Server API
- CorticonExtensions API

See also:

- Introducing the Progress® Application Server
- Corticon documentation for this release on the Progress download site: What's New Guide (PDF), Installation Guide (PDF), PDF download package, and the online Eclipse help installed with Corticon Studio.