PROGRESS® CORTICON®

Corticon Server:
Integration & Deployment Guide

Notices

Table of Contents

Preface

For details, see the following topics:

- Progress Corticon documentation
- Overview of Progress Corticon

Progress Corticon documentation

The following documentation, as well as a *What's New in Corticon* document, is included with this Progress Corticon release:

Corticon Tutorials	
Corticon Studio Tutorial: Basic Rule Modeling	Introduces modeling, analyzing, and testing rules and decisions in Corticon Studio. Recommended for evaluators and users getting started. *See also the PowerPoint-as-PDF version of this document that is accessed from the Studio for Analysts'* ***Help*** *menu.*
Corticon Studio Tutorial: Advanced Rule Modeling	Provides a deeper look into Corticon Studio's capabilities by defining and testing vocabularies, scope, collections, messages, filters, conditions, transient data, and calculations in multiple rulesheets that are assembled into a Ruleflow. *See also the PowerPoint-as-PDF version of this document that is accessed from the Studio for Analysts'* ***Help*** *menu.*
Corticon Tutorial: Using Enterprise Data Connector (EDC)	Introduces Corticon's direct database access with a detailed walkthrough from development in Studio to deployment on Server. Uses Microsoft SQL Server to demonstrate database read-only and read-update functions.

Corticon Studio Documentation: Defining and Modeling Business Rules	
Corticon Studio: Installation Guide	Step-by-step procedures for installing Corticon Studio and Corticon Studio for Analysts on computers running Microsoft Windows. Also shows how use other supported Eclipse installations for integrated development. Shows how to enable internationalization on Windows.
Corticon Studio: Rule Modeling Guide	Presents the concepts and purposes the Corticon Vocabulary, then shows how to work with it in Rulesheets by using scope, filters, conditions, collections, and calculations. Discusses chaining, looping, dependencies, filters and preconditions in rules. Presents the Enterprise Data Connector from a rules viewpoint, and then shows how database queries work. Provides information on versioning, natural language, reporting, and localizing. Provides troubleshooting and many *Test Yourself* exercises.
Corticon Studio: Quick Reference Guide	Reference guide to the Corticon Studio user interface and its mechanics, including descriptions of all menu options, buttons, and actions.
Corticon Studio: Rule Language Guide	Reference information for all operators available in the Corticon Studio Vocabulary. A Rulesheet example is provided for many of the operators. Includes special syntax issues, handling arithmetic and character precedence issues.
Corticon Studio: Extensions Guide	Detailed technical information about the Corticon extension framework for extended operators and service call-outs. Describes several types of operator extensions, and how to create a custom extension plug-in.
Corticon Server Documentation: Deploying Rules as Decision Services	
Corticon Server: Deploying Web Services with Java	Details installing the Corticon Server as a Web Services Server, and then and deploying and exposing Decision Services as Web Services on Tomcat and other Java-based servers. Presents the features and functions of the browser-based Server Console.
Corticon Server: Deploying Web Services with .NET	Details installing the Corticon Server as a Web Services Server and deploying and exposing decisions as Web Services with .NET. Provides installation and configuration information for the .NET Framework and Internet Information Services (IIS) on various supported Windows platforms.
Corticon Server: Integration & Deployment Guide	An in-depth, technical description of Corticon Server deployment methods, including preparation and deployment of Decision Services and Service Contracts through the Deployment Console tool. Discusses relational database concepts and implementation of the Enterprise Data Connector. Goes deep into the server to discuss state, persistence, and invocations by version or effective date. Includes samples, server monitoring techniques, and recommendations for performance tuning.

Overview of Progress Corticon

Progress® Corticon® is the Business Rules Management System with the patented "no-coding" rules engine that automates sophisticated decision processes.

Progress Corticon products

Progress Corticon distinguishes its development toolsets from its server deployment environments.

- **Corticon Studios** are the Windows-based development environment for creating and testing business rules:
  - **Corticon Studio for Analysts.** is a standalone application, a lightweight installation that focuses exclusively on Corticon.
  - **Corticon Studio** is the *Corticon Designer* perspective in the **Progress Developer Studio** (PDS), an industry-standard Eclipse 3.7.1 and Java 7 development environment. The PDS enables integrated applications with other products such as Progress OpenEdge and Progress Apama.

  The functionality of the two Studios is virtually identical, and the documentation is appropriate to either product. Documentation of features that are only in the *Corticon Designer* (such as on integrated application development and Java compilation) will note that requirement. Refer to the *Corticon Studio: Installation Guide* to access, prepare, and install each of the Corticon Studio packages.

  Studio Licensing - Corticon embeds a time-delimited evaluation license that enables development of both rule modeling and Enterprise Data Connector (EDC) projects, as well as testing of the projects in an embedded Axis test server. You must obtain studio development licenses from your Progress representative.

- **Corticon Servers** implement web services for business rules defined in Corticon Studios:
  - **Corticon Server for deploying web services with Java** is supported on various application servers, and client web browsers. After installation on a supported Windows platform, that server installation's deployment artifacts can be redeployed on various UNIX and Linux web service platforms as Corticon Decision Services. The guide *Corticon Server: Deploying web services with Java* provides details on the full set of platforms and web service software that it supports, as well as installation instructions in a tutorial format for typical usage.
  - **Corticon Server for deploying web services with .NET** facilitates deployment of Corticon Decision Services on Windows .NET Framework 4.0 and Microsoft Internet Information Services (IIS). The guide *Corticon Server: Deploying web services with .NET* provides details on the platforms and web service software that it supports, as well as installation instructions in a tutorial format for typical usage.

  Server Licensing - Corticon embeds a time-delimited evaluation license that enables evaluation and testing of rule modeling projects on supported platform configurations. You must obtain server deployment licenses and server licenses that enable the Enterprise Data Connector (EDC) from your Progress representative.

1

Introduction to Corticon Server deployment

The Corticon Server installation and deployment process involves the sequence of activities illustrated in the following diagram. Use this diagram as a map to this manual – each box below corresponds to a following chapter.

For details, see the following topics:

- Choose the deployment architecture

Choose the deployment architecture

Corticon Decision Services are intended to function as part of a service-oriented architecture. Each Decision Service automates a discrete decision-making activity – an activity defined by business rules and managed by business analysts.

Important: A Corticon *Ruleflow* deployed to the Corticon Server and available to process transactions is referred to as a "Decision Service." *Rulesheet*s are not directly deployable to Corticon Server. They must be "packaged" as *Ruleflow*s in order to be deployed and executed on Corticon Server.

The application architect must consider how these Decision Services will be used ("consumed") by external applications, clients, processes or components. Which applications need to consume Decision Services and how will they invoke them? Your choice of installation and deployment architecture impacts subsequent steps, including installation of Corticon Server, and integration and invocation of the individual Decision Services deployed to Corticon Server.

The primary available options are described in the following table, and addressed in detail below:

Table 1: Table: Corticon Server Installation Options

Installation Option	Description	Appropriate If:
1 - Web Services	Corticon Server is deployed with a Servlet interface, causing individual *Ruleflow*s to act as Web Services. Invocations to Corticon Server are made using standard SOAP requests, and data is transferred within the SOAP request as an XML "payload".	• Currently using Web Services. • Need to expose Decision Services to the Internet or other distributed architecture. • Using Microsoft .NET or other legacy systems which do not support Java method calls (invocations).
2 - Java Services with XML Payloads	Corticon Server is deployed with an Enterprise Java Bean (EJB) interface and integrated with architectures that can make Java method calls and transfer XML payloads.	• Prefer to use XML for best flexibility in data payload. • Prefer JMS or RMI method calls for high performance and/or tighter coupling to client applications.
3 - Java Services with Java Object Payloads	Corticon Server is deployed with an Enterprise Java Bean (EJB) interface and integrated with architectures that can make Java method calls and transfer Java object payloads.	• Prefer Java objects for data payload. • Prefer JMS or RMI method calls for high performance. • Willing to accept decreased portability
4 - In-process Java Classes ("POJO")	Corticon Server is deployed into a client-managed JVM as Java classes	• Require lightest-weight, smallest-footprint install. • Prefer direct, in-process method calls for lowest messaging overhead and fastest performance

Table 2: Table: Corticon Server Communication Options

Server Installed As…	Call Server With…	Send Data As…
Java Servlet	• SOAP: RPC or Document-style	• XML String (RPC-style) • XML Document (Document-style)
Java Session EJB	• Corticon Server API via JMS • Corticon Server API via RMI	• XML String or JDOM • collection or map of Java Business Objects
Java Classes	• in-process Java methods from the Corticon Server API	• XML String or JDOM • collection or map of Java Business Objects

Installation option 1: Web services

Web Services is the most common deployment choice. By using the standards of Web Services (including XML, SOAP, HTTP, WSDL, and XSD), this choice offers the greatest degree of flexibility and reusability.

Corticon Server may be installed as a Web Service using a Java Servlet running in a J2EE web or application server's Servlet container. You can use a Web Services server with IBM WebSphere, Oracle/BEA WebLogic, Apache Tomcat or other containers that support multi-threading Web Services (see *Installing Corticon Server*).

The Web Services option is the easiest to configure and integrate into diverse consuming applications. Refer to *Corticon Server: Deploying Web Services with Java* and *Corticon Server: Deploying Web Services with .NET*.

When deploying *Corticon* Decision Services into a Web Services server, the *Deployment Console* (or Deployment Console API) is used to generate WSDL files for each Decision Service (see Deploying Corticon Ruleflows). These WSDL files can then be used to integrate the Decision Services into Consuming applications as standard Web Services (see Integrating Decision Services). *Corticon* users can also build their own infrastructure that publishes the WSDL files to UDDI directories for dynamic discovery and binding.

Installation option 2: Java services with XML message payloads

You are not restricted to Web Services and SOAP as the technical application architecture. *Corticon Server* is, at its core, a set of Java classes. You can deploy *Corticon Server* as:

- A J2EE Stateless Session bean (EJB).
- A set of In-process Java classes on the server or client-side.

This approach avoids the overhead of SOAP messaging, but requires that consuming applications speak Java, in other words, be able to invoke the *Corticon Server* API via JMS or RMI. The payload of the call is the same XML representation as in the Web Services deployment method, minus the SOAP wrapper. Using XML offers good decoupling of consuming application from Decision Service and greater degrees of flexibility.

Installation option 3: Java services with java object payloads

In cases where it is not appropriate to send a string containing the XML payload (or receive a string back as a response) as is required by Option 2, *Corticon* offers an additional way to pass the payload:

- As Java objects (by reference) conforming to the JavaBeans specification. Each Java object corresponds to an entity in the *Corticon* Decision Service Vocabulary. *Corticon Server* uses introspection to identify the entity's attributes (as JavaBean properties).

This option offers the best performance, as payloads do not need transformation from objects to/from XML. That being said, it is also the least portable because it requires Java objects and a tight relationship between those objects and the *Corticon* Vocabulary to exist. In addition, it suffers in flexibility because changes to the Vocabulary require changes to the Java object model.

Note: External name mapping and extended attributes, as discussed below and in the *Corticon* product documentation, offer some help in coping with these constraints.

Installation option 4: In-process Java classes with Java object or XML payloads

The installation option with lightest weight and smallest footprint is the In-process Java option.

With this option, no interface or wrapper class is used to forward calls from the client application to *Corticon Server* (`CcServer.jar`). Instead, the client must use the *Corticon Server* Java API to initialize the *Corticon Server* classes, load any Decision Services, and execute them. In addition, the client application must start and manage the JVM in which the *Corticon Server* classes are loaded.

JVM and thread management are normally functions of the Servlet or EJB container in a web or application server – if you choose to take responsibility for these activities in your client code then you do not need a container, at least as far as *Corticon Server* is concerned. Installing *Corticon Server* without a web or application server reduces the overall application footprint and permits more compact installations, but by eliminating the helpful functions of the container, it places more of the deployment burden on you.

2

Types of Corticon Servers

Corticon Server is provided in two installation sets: Corticon Server for Java, and Corticon Server for .NET.

Corticon Servers implement web services for business rules defined in Corticon Studios.

- The **Corticon Server for deploying web services with Java** is supported on various application servers, databases, and client web browsers. After installation on a supported Windows platform, that server installation's deployment artifacts can be redeployed on various UNIX and Linux web service platforms. The guide *Corticon Server: Deploying Web Services with Java* provides details on the full set of platforms and web service software that it supports, as well as installation instructions in a tutorial format for typical usage. See *Deploying Web Service with Java* for information about its files and API tools.
- The **Corticon Server for deploying web services with .NET** facilitates deployment on Windows .NET framework 4.0 and Microsoft Internet Information Services (IIS) that are packaged in the supported Windows operating systems. The guide *Corticon Server: Deploying Web Services with .NET* provides details on the full set of platforms and web service software that it supports, as well as installation instructions in a tutorial format for typical usage. See *Deploying Web Service with .NET* for information about its files and API tools.

3

Preparing Studio files for deployment

For details, see the following topics:

Mapping the Vocabulary

Part of the integration process involves mapping our Vocabulary terms (which are used by the rules in our *Rulesheet*s) to the structure of the data that will be sent to the deployed *Ruleflow*s in runtime. This ensures that when the Decision Service is invoked, the data included in the invocation will be understood, translated, and processed correctly.

To map your Vocabulary, Corticon Studio must be set to **Integration & Deployment** mode. To select this mode, choose the Studio menu item **Window** > **Preferences**. Expand **Progress Corticon**, and then click on **Rule Modeling**. Select the radio button **Integration & Deployment**.

Note: If you have your license file available, make it accessible, and then browse to choose its location in this panel.

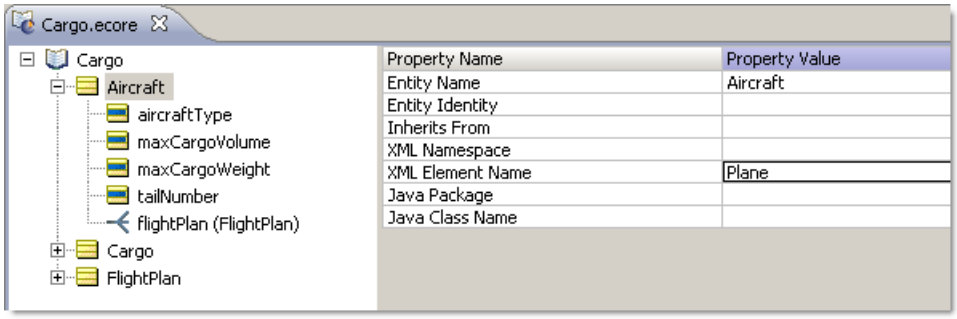
XML mapping

If you have chosen to use Option 1 or 2 in the table Corticon Server Installation Options – in other words, the data payload of your call will be in the form of an XML document – then your Vocabulary may need to be configured to match the naming convention of the elements in your XML payload.

Entity Mapping

Vocabulary entities correspond to XML complex elements (complexTypes). If the `complexType` matches exactly (spelling, case, special characters, *everything*), then no mapping is necessary. However, if the `complexType` name differs in any way from the Vocabulary entity name, then the `complexType` name must be entered into the **XML Class Name** property, as shown below.

Figure 1: Mapping a Vocabulary Entity to an XML complexType



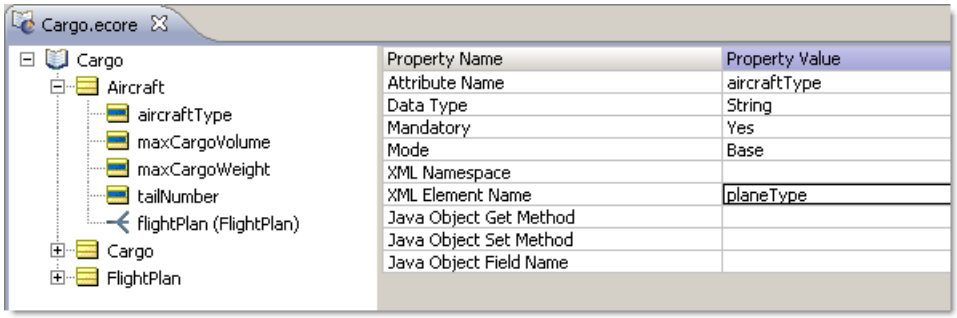
In the example shown in this figure, the Vocabulary entity name (`Aircraft`) does not *exactly* match the name of the external XML Class (`Plane`), so the mapping entry is required. If the two names were identical, then no mapping entry would be necessary.

If XML Namespaces vary within the document, then use the **XML Namespace** field to enter the full namespace of the XML Element Name. If no XML Namespace value is entered, then it is assumed that all XML Elements use the same namespace.

Attribute Mapping

Vocabulary attributes correspond to XML simple elements. If the element name matches exactly (spelling, case, spaces, and non-alphanumeric characters), then no mapping is necessary. However, if the element name differs in *any* way from the Vocabulary attribute name, then the element name must be entered into the **XML Property Name** property, as shown in the following figure.

Figure 2: Mapping a Vocabulary Attribute to an XML SimpleType

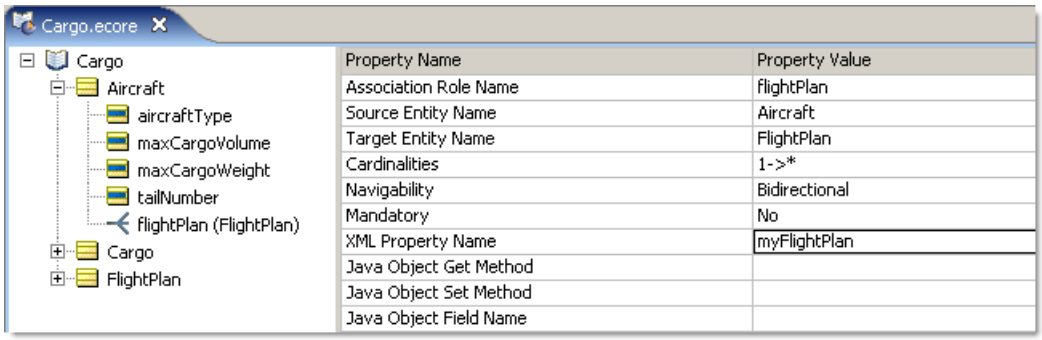


If XML Namespaces vary within the document, then use the **XML Namespace** field to enter the full namespace of the XML Element Name. If no XML Namespace value is entered, then it is assumed that all XML Elements use the same namespace.

Association Mapping

Vocabulary associations correspond to references between XML complex elements. If the element name matches exactly (spelling, case, special characters, *everything*), then no mapping is necessary. However, if the element name differs in any way from the Vocabulary association name, then the element name must be entered into the **XML Property Name** property, as shown below.

Figure 3: Mapping a Vocabulary Association to an XML ComplexType



XML Namespace Mapping – Changes in BRMS Version 5.2

In BRMS versions prior to 5.2, *Corticon Server* assumed that incoming XML requests were loosely compliant with the XSD/WSDL generated for a particular Decision Service (by the Deployment Console, for example). The *Corticon* XSD/WSDLs generated all had a `targetNamespace` of `urn:Corticon`.

The problem with this approach is that most SOA systems, particularly those that care about XML validation, require the `targetNamespace` to be unique, ideally globally unique. Two properties in `CcDeployment.properties` (`com.corticon.xml.addDefaultNamespace` and `com.corticon.schemagenerator.addDefaultNamespace`) allow you to "turn off" `targetNamespace` usage in the XSD/WSDL altogether. But prior to version 5.2, you could not selectively assign specific namespaces to specific elements (either Entities or Attributes) within the Vocabulary.

Starting in version 5.2, the XML Schema `targetNamespace` is controlled by a new `CcDeployment.properties` property named `ensureUniqueTargetNamespace` so that you can specify the expected behavior of *`Corticon`*'s XSD/WSDL generation subsystem. By default, the SOAP envelope `targetNamespace` will be set to a concatenation of the following strings:

- the WSDL's service soap address location +
- forward slash character (`/`) +
- the decision service name.

You can also switch back to old (prior to version 5.2) behavior for backward compatibility purposes by setting `ensureUniqueTargetNamespace=false`.

The following screenshots juxtapose old XSD/WSDLs with new XSD/WSDLs:

Figure 4: Old XSD with Generic Namespace (version 5.1 and earlier)

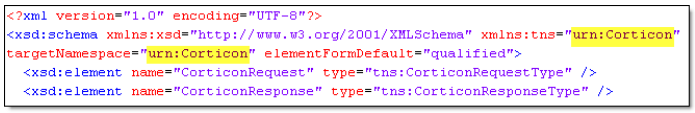


Figure 5: New XSD with Unique Namespace (version 5.2 and later)

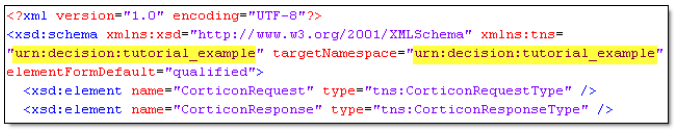


Figure 6: Old WSDL with Generic Namespace (version 5.1 and earlier)

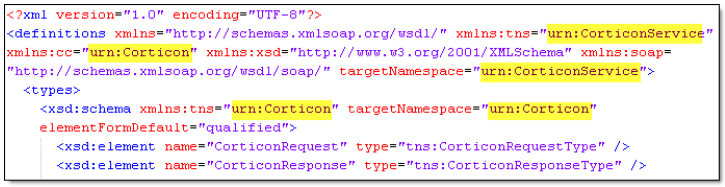
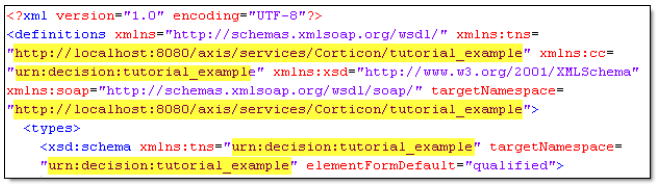


Figure 7: New WSDL with Unique Namespace (version 5.2 and later)



Namespace mapping changes in version 5.2 also affected some APIs. See the *JavaDoc* for full details.

Java object mapping

If you have chosen to use Option 3 in Corticon Server Installation Options – in other words, the data payload of your call will be in the form of a map or collection of Java objects – then your Vocabulary may need to be configured to match the method names within those objects.

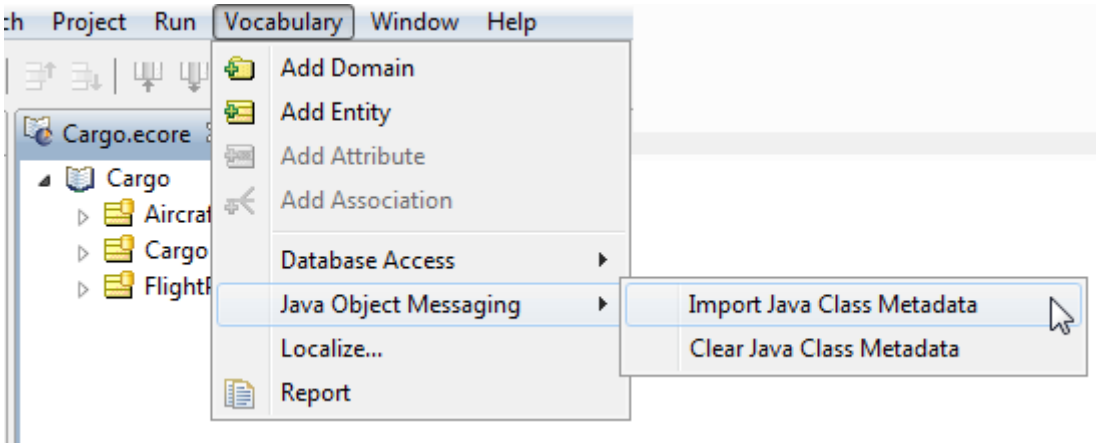
Corticon Studio can import a package of classes and automatically match the object structure with the Vocabulary structure. In other words, it will try to determine which objects match which Vocabulary entities, which properties match which Vocabulary attributes, and which object references match which Vocabulary associations.

To perform this matching, Corticon Studio assumes your objects are JavaBean compliant, meaning they contain public get and set methods to expose those properties used in the Vocabulary. Without this JavaBean compliance, the automatic mapper may fail to fully map the package, and you will need to complete it manually.

To import package metadata:

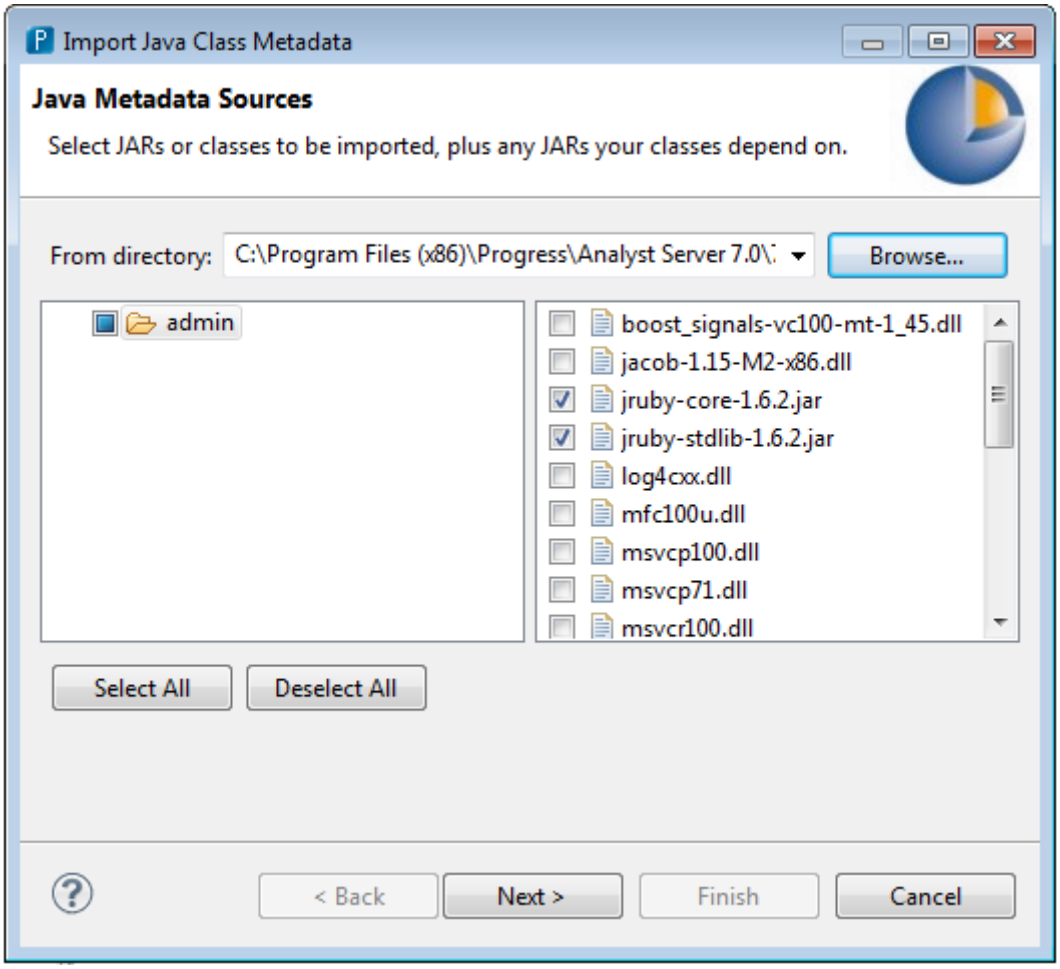
1. Open your Vocabulary in Corticon Studio's **Vocabulary Edit** mode.
2. From the menubar, select **Vocabulary** > **Java Object Messaging** > **Import Java Class Metadata**, as shown in the following figure:

Figure 8: Importing Java Class Metadata for Mapping



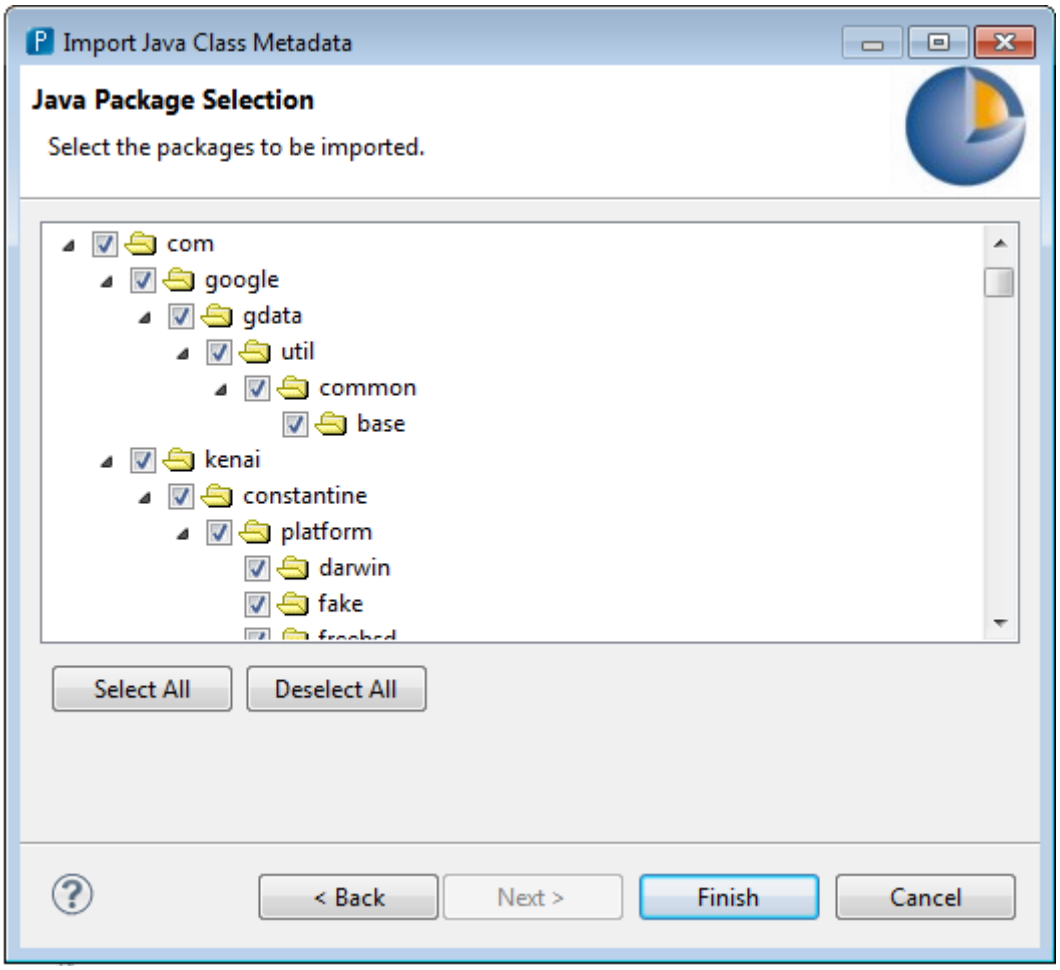
3. Use the **Browse** button to select the location of your Java Business Objects. They should be compiled `class` files or Java archives (`.jar` files).

Figure 9: Browsing to your Java Class files



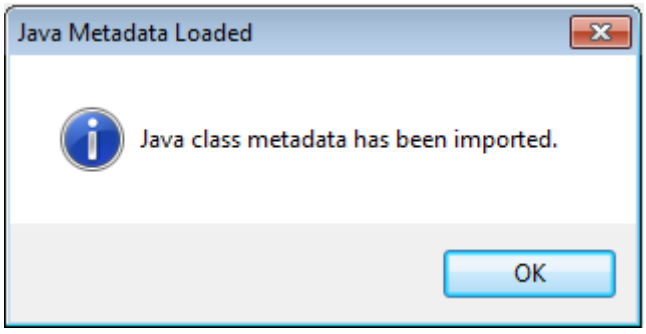
4. Select the package containing the Java business objects as shown:

Figure 10: Importing Java Class Metadata for Mapping



5. When the import succeeds, you see the following message:

Figure 11: Java Class Metadata Import Success Message



Now that the import is complete, we will examine our Vocabulary to see what happened.

Entity

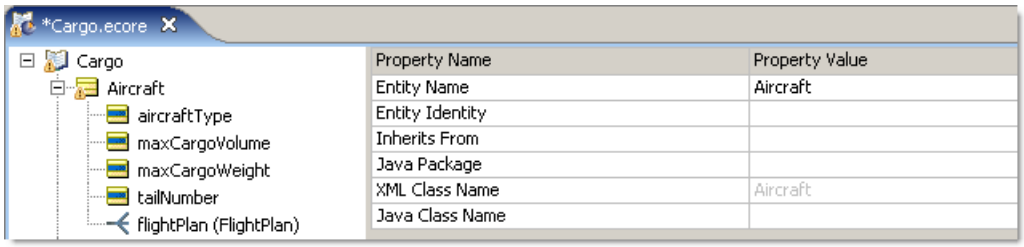
Let's take a look at a sample class that we might have wanted to map to the `Aircraft` entity.

Figure 12: First Portion of MyAircraft Class

```
1  package com.corticon.bo.tutorial;
2
3  import java.math.BigDecimal;
4  import java.util.Vector;
5
6  public class MyAircraft
7  {
8     //  Pulic Attribute Instance Variables
9     public  String     istrAircraftType = null;
10
11    //  Private Attribute Instance Variables
12    private BigDecimal ibdMaxCargoVolume = null;
13    private Float      ifMaxCargoWeight = null;
14    private String     istrTailNumber = null;
15
16    //  Private Association Instance Variables
17    private Vector ivectFlightPlan = new Vector();
18
19    //-------------------------------------------------------
20    //  Zero Argument Constructor
21    //-------------------------------------------------------
22    public MyAircraft() {}
```

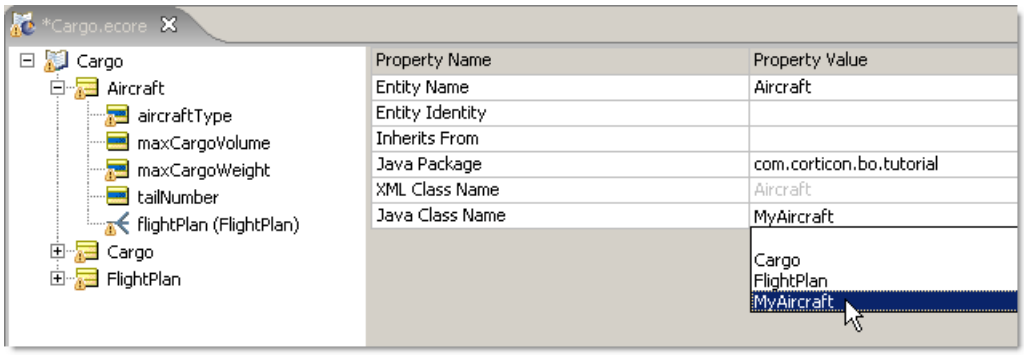
We can see in line 6 of this figure that this class is not actually named `Aircraft` – it is named `MyAircraft`. The automatic mapper attempts to locate a class by the same name as each entity. In the case of `Aircraft`, it looks for a class named `Aircraft`. Not finding one, it leaves the field empty, as shown in the following figure.

Figure 13: Default Map of Class to Entity



Because no `Aircraft` class exists in the package, we need to manually map this entity, using the **Java Package** and **Java Class Name** drop-downs, as shown in the following figure. The metadata import process populates the drop-downs for us. Be sure to select the package name from the **Java Package** drop-down so the mapper knows where to look.

Figure 14: Manually Mapping MyAircraft Class to Aircraft Entity



Attribute

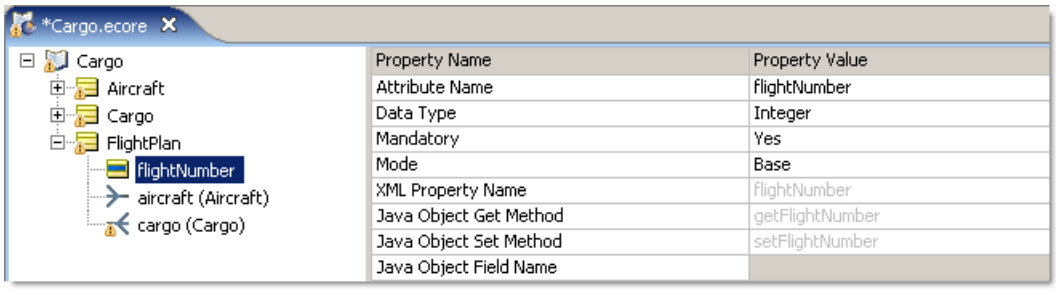
When attempting to map attributes, the mapper looks for class properties which are exposed using public get and set methods by the same name. For example, if mapping attribute `flightNumber`, the mapper looks for public `getFlightNumber` and `setFlightNumber` methods in the mapped class.

Figure 15: Second Portion of MyAircraft Class

```
23
24    //-------------------------------------------------------
25    //  Attribute Getter / Setters
26    //-------------------------------------------------------
27    public BigDecimal getMaxCargoVolume() {
28       return ibdMaxCargoVolume;
29    }
30    public void setMaxCargoVolume(BigDecimal abdValue) {
31       ibdMaxCargoVolume = abdValue;
32    }
33
34    public Float getMyMaxCargoWeight() {
35       return ifMaxCargoWeight;
36    }
37    public void setMyMaxCargoWeight(Float afValue) {
38       ifMaxCargoWeight = afValue;
39    }
40
41    public String getTailNumber() {
42       return istrTailNumber;
43    }
44    public void setTailNumber(String astrValue) {
45       istrTailNumber = astrValue;
46    }
47
```

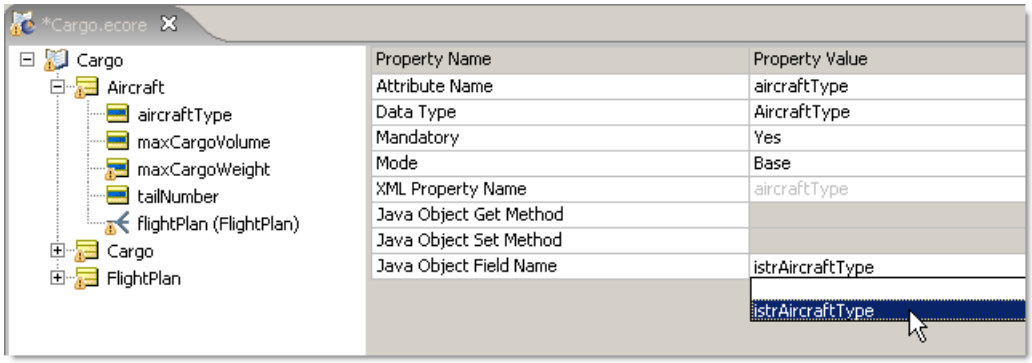
In the case of attribute `tailNumber`, the mapper finds get and set methods that conform to this naming convention, so the method names are inserted into the fields in gray type, as shown in the following figure.

Figure 16: Auto-Mapped Attribute Method Names



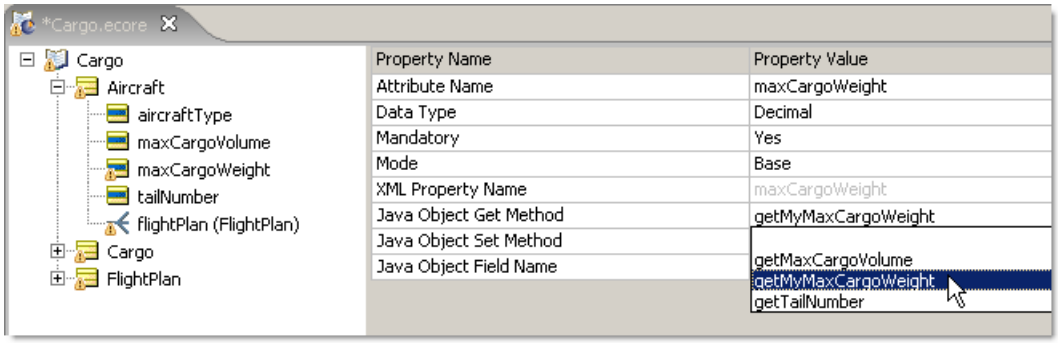
In those cases where the mapper cannot locate the corresponding methods, you will need to select them manually. Notice in the `MyAircraft` class shown in First Portion of MyAircraft Class, no get and set methods exist for `istrAircraftType` since it is a public instance variable. Therefore, we need to select it from the **Java Object Field Name** drop-down, as shown in the following figure.

Figure 17: Manually Mapped Public Instance Variable Name



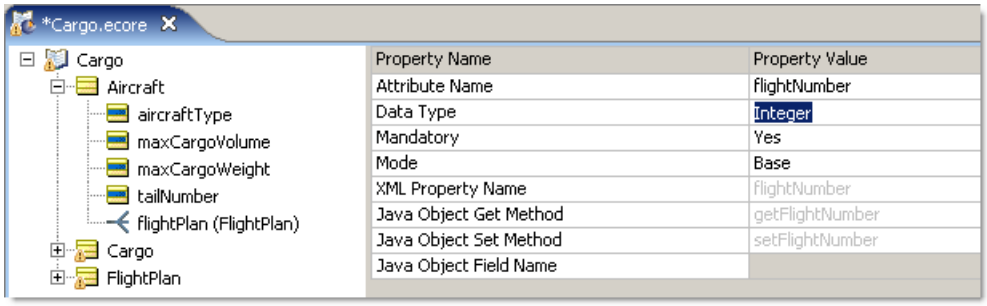
When a class property contains get and set methods, but their names do not conform to the naming convention assumed by the auto-mapper, we must select the method names from the **Java Object Get Method** and **Set Method** drop-downs, as shown in the following figure.

Figure 18: Manually Mapped Property Get and Set Method Names



If you worked with Java Object Messaging and mapping in versions of Corticon Studio prior to 5.2, then you may recall that external data types also required manual mapping. Starting in version `5.2`, a property's data type is detected by the auto-mapper, so there's no need to manually enter it. This is shown by the `flightNumber` attribute in the following figure.

Figure 19: Auto-Mapped Property Despite Different Data Type



Note: First Portion of MyAircraft Class shows that this property uses a primitive data type int, and it is automatically mapped anyway.

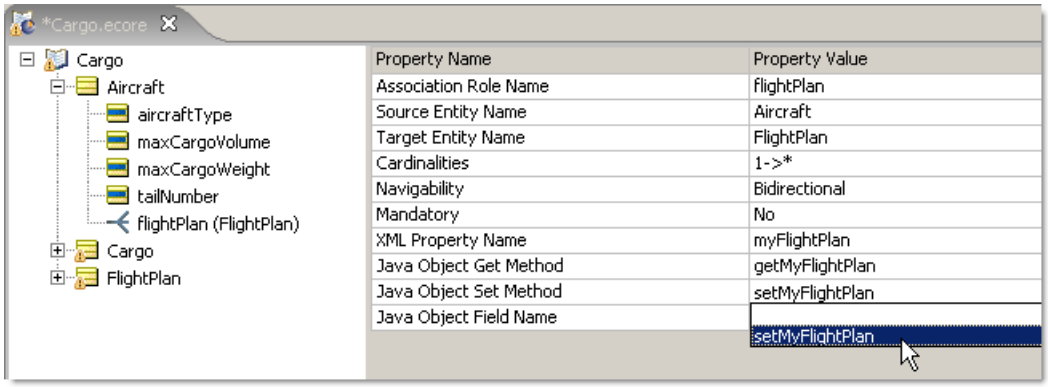
Association

The mapper looks for get and set methods for associations the same way that it does for attributes. In the case of the `Aircraft.flightPlan` association, shown in Third Portion of MyAircraft Class, below, these methods do not conform to the naming convention expected by the mapper. So once again, we must manually select the appropriate method names from the **Java Object Get Method** and **Set Method** drop-downs, as shown in Manually Mapped Association Get and Set Method Names, below.

Figure 20: Third Portion of MyAircraft Class

```
48      //-------------------------------------------------------
49      //  Association Getter / Setters
50      //-------------------------------------------------------
51      public Vector getMyFlightPlan() {
52         return ivectFlightPlan;
53      }
54      public void setMyFlightPlan(Vector avectValue) {
55         ivectFlightPlan = avectValue;
56      }
57   }
58
```

Figure 21: Manually Mapped Association Get and Set Method Names



Java generics

Support for type-casted collections is included in Corticon Studio. If your Java Business Objects include type-casted collections (introduced in Java 5), then Corticon will ensure these constraints are interpreted correctly in association processing.

Java enumerations

Enumerations are custom Java objects you define and use inside of your Business Objects. They are used to define a preset "value set" for a particular type.

For simplicity, let's assume that a Java Enumeration has a Name and multiple Labels (or Types). Here is a common example of a Java Enumeration:

```
public enum Day
{
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY;
}
```

The `Day` enumeration has 5 different Labels {`Day.MONDAY`, `Day.TUESDAY`, `Day.WEDNESDAY`, `Day.THURSDAY`, and `Day.FRIDAY`}. All of these labels are all considered "of type `Day`". So if a method signature accepts a `Day` type, it will accept all 5 of these defined labels.

For example:

```
public class Person
{
   private Day iPayDay = null;

   public Day getPayDay() {return iPayDay;}
   public void setPayDay(Day aValue) {iPayDay = aValue;}
   }
```

And here is an example of a call to the `setPayDay(Day)` method:

```
lPerson.setPayDay(Day.MONDAY);
```

Because `Day.MONDAY` is of type `Day`, the setting of the value is complete.

Prior to Version *5.2*, business rules could only set basic Data Types into Business Objects. Basic data types included String, Long, long, Integer, int, and Boolean. But starting in *5.2*, business rule execution can also set your business object's Enumeration values.

Corticon performs this by matching Labels in your business object's enumerations with the Custom Data Type (CDT) labels defined in your Vocabulary.

From our example:

Java Enumeration Label Names for enum `Day`:

`MONDAY`

`TUESDAY`

`WEDNESDAY`

`THURSDAY`

`FRIDAY`

Now, the Vocabulary must have these same Labels defined in the CDT that is assigned to the attribute.

Figure 22: Vocabulary CDT Labels must match Business Object Enumeration Labels (Types)

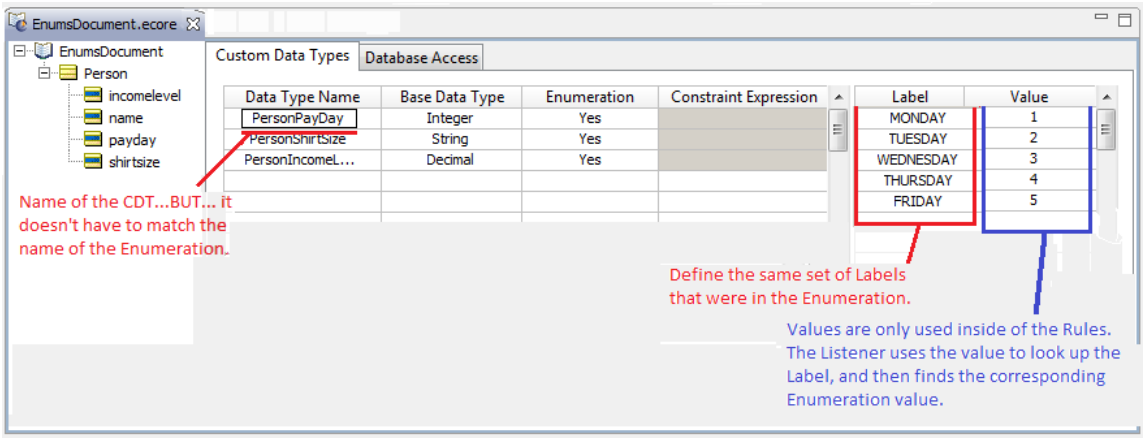
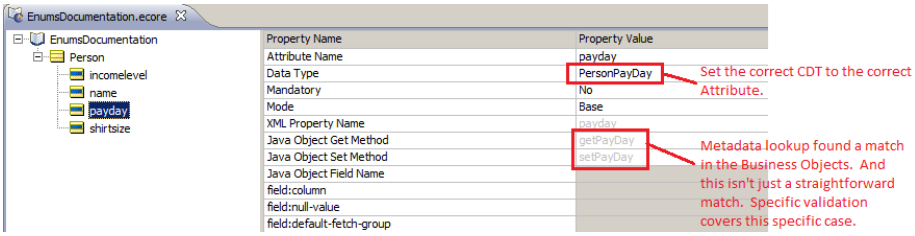


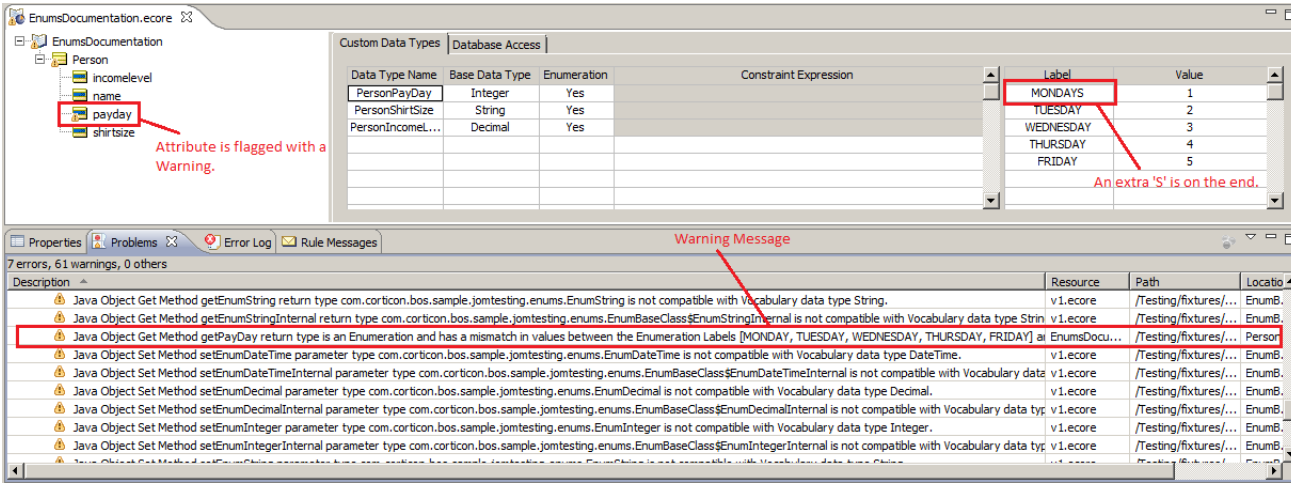
Figure 23: Vocabulary Mapper found correct Metadata based on matching enumeration labels



The key to metadata matching is the Labels – as long as your BO enumeration labels match the Vocabulary's CDT Labels, it should work fine. So what happens if the Labels do NOT match?

Extra validation has been added to the Vocabulary to help identify this problem. The example below shows a Label mismatch:

Figure 24: Vocabulary CDT Label / Object Enumeration Label Mismatch



Notice the Vocabulary attribute is "flagged" with the small orange warning icon (shown in the upper left of the figure above). The associated warning message states:

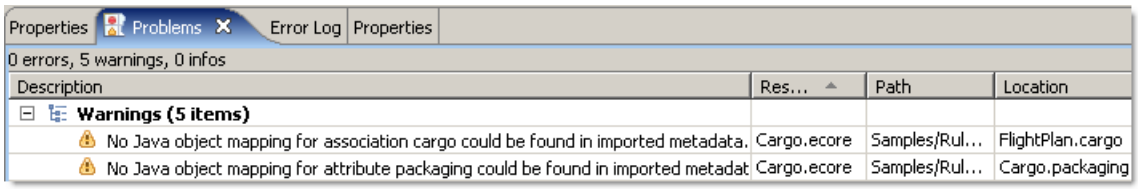
```
Java Object Get Method getPayDay return type is an Enumeration and has
a mismatch in values between the Enumeration Labels [MONDAY, TUESDAY,
WEDNESDAY, THURSDAY, FRIDAY] and Custom Datatype Labels [MONDAYS, TUESDAY,
WEDNESDAY, THURSDAY, FRIDAY].
```

Verifying java object mapping

You may have noticed that in several of the screenshots above, small orange triangle "warning" icons appear next to the Vocabulary nodes whose mappings have not yet been selected. Each warning will have a corresponding message entered in the **Problems** window, which is usually located towards the bottom of the Corticon Studio window. If you do not see it,

- Use **Window** > **>** > **Show View** > **>** > **Problems** to display it.

Figure 25: Problem window showing list of current mapping problems



When all mappings have been made (either automatically or manually), these warning icons will disappear.

Listeners

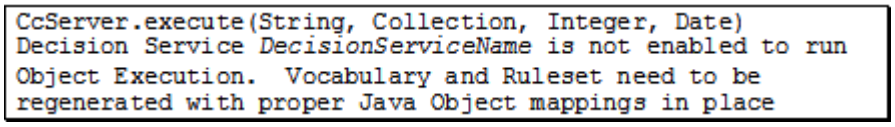
During runtime, when an attribute's value is updated by rules, the update is communicated back to the business object by way of "Listener" classes. Listener classes are compiled at deployment time when Corticon Server detects a *Ruleflow* using Java Object Messaging. Once compiled, these Listener classes are also added to the `.eds` file, which is the compiled, executable version of the `.erf`. This process ensures that Corticon Server "knows" how to properly update the objects it receives during an invocation. Because the update process uses compiled Listener classes instead of Java Reflection, the update process occurs very quickly in runtime.

Even though Java Object metadata was imported into Corticon Studio for purposes of mapping the Vocabulary, and those mappings were included in the *Rulesheet* and *Ruleflow* assets, the Listener classes, like the rest of the `.erf`, is not compiled until deployment time. As a result, the same Java business object classes must also always be available to Corticon Server during runtime.

Corticon Server assumes it will find these classes on your application server's classpath. If it cannot find them, Listener class compilation will fail, and your deployed *Ruleflow* will be unable to process transactions using Java business objects as payload data.

If a *Ruleflow* (.erf) is deployed to Corticon Server *without* compiled Listeners, then it will accept only invocations with XML payloads, and reject invocations with Java object payloads. When invoked with Java object payloads, Corticon Server will return an exception, as shown in the following figure:

Figure 26: Server Error Message When Listeners Not Present



4

Deploying Corticon Ruleflows

When a *Ruleflow* has been built and tested in Corticon Studio, it must be prepared for deployment. Deploying a *Ruleflow* requires at least one and, often two, steps:

Note: *Rulesheet*s may be tested in Corticon Studio using *Ruletest*s, however they must be packaged as *Ruleflow*s in order to be deployed to Corticon Server. *Rulesheet*s cannot be deployed directly to Corticon Server.

For details, see the following topics:

- Ruleflow deployment
- Publish and Download Wizards
- Testing the deployed Corticon Decision Service
- Deploying uncompiled vs. pre-compiled decision services

Ruleflow deployment

Once a *Ruleflow* has been deployed to the Corticon Server, we stop calling it a *Ruleflow* and start calling it a Decision Service. The process of "deploying" a *Ruleflow* is really an act of instructing a running Corticon Server instance to load a *Ruleflow* and prepare to execute it in response to a call from an external client application. There are 2 ways to inform the Corticon Server which *Ruleflow*s it should load and how to configure certain execution parameters for each. These methods are discussed in the following topics.

Ruleflow deployment using Deployment Descriptor files

Before a Decision Service can be consumed by a client application, Corticon Server must first be initialized by the application architecture startup sequence. Then, Corticon Server loads one or more Deployment Descriptor files by calling the `ICcServer` interface's `loadFromCdd` method. This causes Corticon Server to read the Deployment Descriptor file(s), load each listed *`Ruleflow`*, and set other execution and configuration parameters accordingly.

Deployment Descriptor files, which have the filename suffix `.cdd`, tell Corticon Server the following:

- The name(s) and directory location(s) of the *`Ruleflow`*(s) to load upon startup.
- The reload option each Decision Service will use.
- The type of request message style to expect from consumers of each Decision Service.
- The unique name ("Decision Service Name") of each *`Ruleflow`*.
- The concurrency properties of each Decision Service.

Deployment Descriptor files are easily created and managed using the Corticon Deployment Console tool. The Deployment Console is included by default in all Corticon Server installations.

Functions of the Deployment Console tool

The Deployment Console has two functions:

- **Create Deployment Descriptor files**. These files, carrying the file suffix `.cdd`, are XML documents which instruct Corticon Server which *`Ruleflow`*s to load, and how to configure certain parameters and settings for each *`Ruleflow`*. Creating and using Deployment Descriptor files will make up the main topics within this chapter.
- **Create XML service contract documents**. These documents are used for *`Ruleflow`* integration and will be discussed in the Integration chapter of this manual.

Using the Deployment Console tool's Decision Services

The Corticon Deployment Console is started, as follows for each of the server types:

- **Java Server:** On the Windows Start menu, choose **Programs** > **Progress** > **Corticon n.n** > **Deployment Console** to launch the script file `\Server\deployConsole.bat`.
- **.NET Server:** On the Windows Start menu, choose **Programs** > **Progress** > **Corticon n.n** > **Corticon Server .NET > Deployment Console** to launch the executable file `Server .NET\samples\bin\DeploymentConsole.exe`.

The Deployment Console is divided into two sections. Because the Deployment Console is a rather wide window, its columns are shown as two screen captures in the following figures. The **red** identifiers are the topics listed below.

Figure 27: Left Portion of Deployment Console, with Deployment Descriptor File Settings Numbered

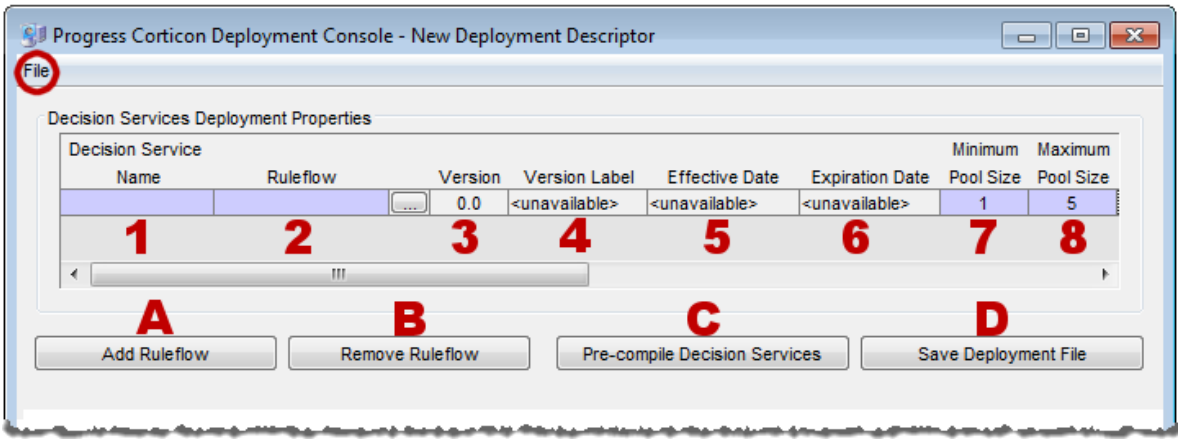
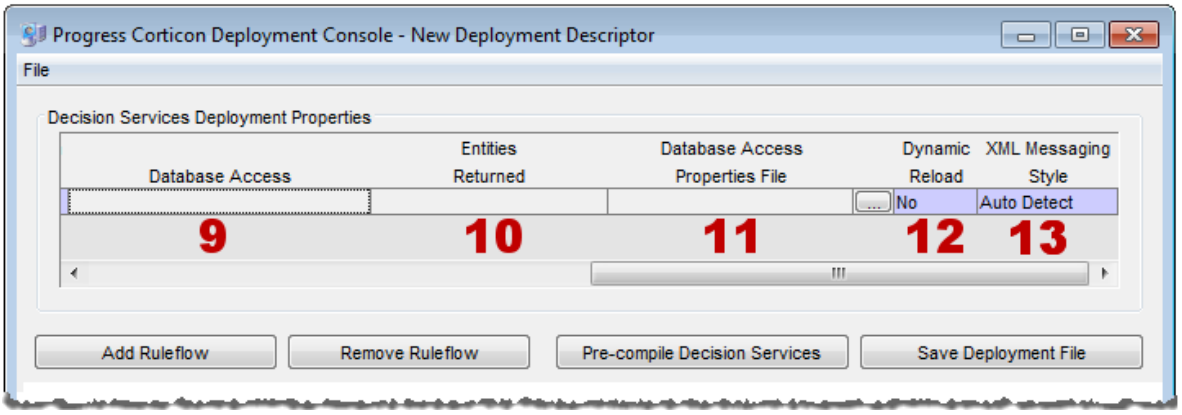


Figure 28: Right Portion of Deployment Console, with Deployment Descriptor File Settings Numbered



The name of the open Deployment Descriptor file is displayed in the Deployment Console's title bar.

The **File** menu, circled in the top figure, enables management of Deployment Descriptor files:

- To save the current file, choose (**File** > **Save**).
- To open an existing `.cdd`, choose (**File** > **Open**).
- To save a `.cdd` under a different name, choose (**File** > **Save As**).

The marked steps below correspond to the Deployment Console columns for each line in the Deployment Descriptor.

1. **Decision Service Name** - A unique identifier or label for the Decision Service. It is used when invoking the Decision Service, either via an API call or a SOAP request message. See Invoking Corticon Server for usage details.
2. **Ruleflow** - All *`Ruleflow`*s listed in this section are part of this Deployment Descriptor file. Deployment properties are specified on each *`Ruleflow`*. Each row represents one *`Ruleflow`*. Use the ... button to navigate to a *`Ruleflow`* file and select it for inclusion in this

Deployment Descriptor file. Note that *Ruleflow absolute* pathnames are shown in this section, but *relative* pathnames are included in the actual .cdd file.

The term "deploy", as we use it here, means to "inform" the Corticon Server that you intend to load the *Ruleflow* and make it available as a Decision Service. It does **not** require actual physical movement of the .erf file from a design-time location to a runtime location, although you may do that if you choose – just be sure the file's path is up-to-date in the Deployment Descriptor file. But movement isn't required – you can save your .erf file to any location in a file system, and also deploy it from the same place *as long as the running Corticon Server can access the path*.

3. **Version** - the version number assigned to the *Ruleflow* in the **Ruleflow** > **Properties** window of Corticon Studio. Note that this entry is editable only in Corticon Studio and not in the Deployment Console. A discussion of how Corticon Server processes this information is found in the topics *"Decision Service Versioning and Effective Dating" of the Integration and Deployment Guide.* Also see the *Quick Reference Guide* for a brief description of the *Ruleflow* Properties window and the *Rule Modeling Guide* for details on using the *Ruleflow* versioning feature. It is displayed in the Deployment Console simply as a convenience to the *Ruleflow* deployer.
4. **Version Label** - the version label assigned to the *Ruleflow* in the **Ruleflow** > **Properties** window of Corticon Studio. Note that this entry is editable only in Corticon Studio and not in the Deployment Console. See the ***Quick Reference Guide*** for a brief description of the *Ruleflow* Properties window and the purpose of the *Ruleflow* versioning feature.
5. **Effective Date** - The effective date assigned to the *Ruleflow* in the **Ruleflow** > **Properties** window of Corticon Studio. Note that this entry is editable only in Corticon Studio and not in the Deployment Console. A discussion of how Corticon Server processes this information is found in the topics *"Decision Service Versioning and Effective Dating" of the Integration and Deployment Guide.* Also see the *Quick Reference Guide* for a brief description of the *Ruleflow* Properties window and the purpose of the *Ruleflow* effective dating feature.
6. **Expiration Date** - The expiration date assigned to the *Ruleflow* in the **Ruleflow** > **Properties** window of Corticon Studio . Note that this entry is editable only in Corticon Studio and not in the Deployment Console. A discussion of how Corticon Server processes this information is found in the topics *"Decision Service Versioning and Effective Dating" of the Integration and Deployment Guide.* Also see the *Quick Reference Guide* for a brief description of the *Ruleflow* Properties window and the purpose of the *Ruleflow* expiration dating feature.
7. **Minimum Pool Size** - The minimum number of instances or 'copies' created for a Decision Service when it is loaded by Corticon Server. Instances of a Decision Service are known as **Reactors** - These Reactors are placed in a pool, where they wait for assignment by Corticon Server to an incoming request, or they expire due to inactivity. The larger the pool size, the greater the concurrency (but greater the memory usage). The default value is **1**, which means that even under no load (no incoming requests) Corticon Server will always maintain one Reactor in the pool for this Decision Service.
8. **Maximum Pool Size** - The maximum number of Reactors Corticon Server can put into the pool for this Decision Service. Therefore, the number of Reactors that can execute concurrently is determined by the max pool size. If additional requests for the Decision Service arrive when all Reactors are busy, the new requests queue until Corticon Server can allocate a Reactor to the new transaction (usually right after a Reactor is finished with its current transaction). The more Reactors in the pool, the greater the concurrency (and the greater the memory usage). See Performance and Tuning chapter for more guidance on Pool configuration. The default value is **5**.

Note: Functions 9, 10, and 11 are active only if your Corticon license enables EDC, and you have registered its location in tool.

Note: If you are evaluating Corticon, your license requires that you set the parameter to `1`.

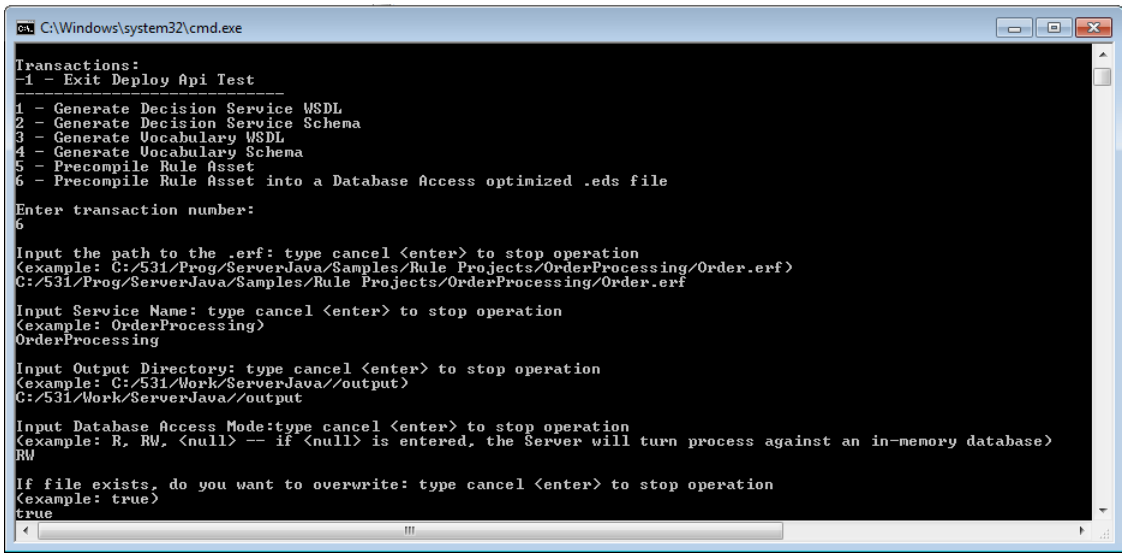
9. **Database Access** - Controls whether the deployed Rule Set has direct access to a database, and if so, whether it will be read-only or read-write access.
10. **Entities Returned** - Determines whether the Corticon Server response message should include all data used by the rules including data retrieved from a database (**All Instances**), or only data provided in the request and created by the rules themselves (**Incoming/New Instances**).
11. **Database Access Properties File** - The path and filename of the database access properties file (that was typically created in Corticon Studio) to be used by Corticon Server during runtime database access. Use the adjacent  button to navigate to a database access properties file.
12. **Dynamic Reload** - If **Yes**, then Corticon Server will periodically look to see if a Deployment Descriptor file, or any of the Decision Service entries in that file, has changed since the `.cdd` was last loaded. If so, it will be automatically reloaded. The time interval between checks is defined by property `com.corticon.ccserver.serviceIntervals` in CcServer.properties. Even if **No**, Corticon Server will still use the most recent *`Ruleflow`* when it adds new Reactors into the pool.
13. **XML Messaging Style** - Determines whether request messages for this Decision Service should contain a flat (**Flat**) or hierarchical (**Hier**) payload structure. The Decision Service Contract Structures section of the Integration chapter provides samples of each. If set to **Auto Detect**, then Corticon Server will accept either style and respond in the same way.

The indicated buttons at the bottom of the Decision Service Deployment Properties section provide the following functions:

- **(A) Add Ruleflow** - Creates a new line in the Decision Service Deployment Properties list. There is no limit to the number of *`Ruleflow`*s that can be included in a single Deployment Descriptor file.
- **(B) Remove Ruleflow** - Removes the selected row in the Decision Service Deployment Properties list.
- **(C) Pre-compile Decision Services** - Compiles the Decision Service before deployment, and then puts the `.eds` file (which contains the compiled executable code) at the location you specify. (By default, Corticon Server does not compile *`Ruleflow`*s *until* they are deployed to Corticon Server. Here, you choose to pre-compile *`Ruleflow`*s in advance of deployment.) The `.cdd` file will contain reference to the `.eds` instead of the usual `.erf` file. Be aware that setting the EDC properties will optimize the Decision Service for EDC.
- **(D) Save Deployment File** - Saves the `.cdd` file. (Same as the menu **File** > **Save** command.)

Using the Deploy API to precompile rule assets

The Corticon Server for Java provides a script, `testDeployConsole.bat` at its `\Server` root, that lets you precompile rule assets by responding to a series of prompts. The following example shows the command options and the default prompts for #6:



Note: When you precompile a Decision Service for use in EDC, you must avoid mismatches of the EDC compilation type and the Decision Service deployment type, else an exception is thrown.

Content of a Deployment Descriptor file

A Deployment Descriptor file is saved as an XML-structured text document. The following example shows how the columns of each Decision Service are tagged:

```
<?xml version="1.0" encoding="UTF-8"?>
<cdd soap_server_binding_url="http://localhost:8082/axis">
  <decisionservice>
    <ruleset-uri>tutorial_example_v0_16.eds</ruleset-uri>
    <auto-reload-if-modified>false</auto-reload-if-modified>
    <database-access>
      <mode>R</mode>
      <entities-returned>ALL</entities-returned>

<runtime-properties>TutorialAccess_SQLServer.properties</runtime-properties>
    </database-access>
    <msg-struct-type />
    <pool>
      <name>tutorial_example</name>
      <min-size>1</min-size>
      <max-size>5</max-size>
    </pool>
  </decisionservice>

  <decisionservice>
    <ruleset-uri>Life Insurance/iSample_policy_pricing.erf</ruleset-uri>
    <auto-reload-if-modified>false</auto-reload-if-modified>
    <database-access>
```

```
      <mode />
      <entities-returned />
      <runtime-properties />
    </database-access>
    <msg-struct-type />
    <pool>
      <name>iSample_policy_pricing</name>
      <min-size>1</min-size>
      <max-size>5</max-size>
    </pool>
  </decisionservice>
</cdd>
```

In the Deployment Descriptor file shown above, note the following:

- There are two `<decisionservice>` sections, indicating that two Ruleflows were listed.
- The first `<decisionservice>` defines database access properties, storing the value `R` as the Read-Only setting
- The **XML Message Type**, tagged as `<msg-struct-type/>`, is not assigned a value in either *`Ruleflow`* sections, thus accepting the default value **Auto detect**.
- The path names to the *`Ruleflow`* `.erf` files (or `.eds` files if pre-compiled) are expressed *relative* to the location of the Deployment Descriptor file (indicated by the `../../` syntax). If the saved location of the Deployment Descriptor file has path in common with the location of the `.erf` file, then the `.erf` path included here is expressed in relative terms. If the two locations have no path in common (they are saved to separate machines, for example), then the `.erf` 's path will be expressed in absolute terms. These paths can be manually edited if changes are required. UNC paths can also be used to direct Corticon Server to look in remote directories. Use caution when moving `.cdd` files - a relative-path pointer to the `.erf` or `.eds` could become invalid.

Important: If you are using the bundled Apache Tomcat to test and deploy your *`Ruleflow`*, copy the Deployment Descriptor file to the Corticon Server installation's `{CORTICON_WORK_DIR]\cdd` directory. Corticon Server reads all `.cdd` files in that default location during start up.

Telling the server where to find Deployment Descriptor files

A Deployment Descriptor file (`.cdd`) tells Corticon Server everything it needs to know to load *`Ruleflow`*s (uncompiled `.erf` or pre-compiled `.eds`) and prepare to execute them, but Corticon Server needs to know where to find the Deployment Descriptor file (or files) itself. There are few ways to do this.

Using the Administrative API set

This is accomplished using admin API methods named `loadFromCdd()` or `loadFromCddDir()`. `loadFromCdd()` requires as an argument a complete path to the Deployment Descriptor file you want to load. `loadFromCddDir()` takes as an argument a path to a directory where Corticon Server will look and load **all** Deployment Descriptor files it finds inside.

These methods are summarized in the Java API Summary in API summary on page 183 and described fully in the *JavaDoc*.

These two methods are used in the API testing batch files `testServer.bat` and `testServerAxis.bat` described in installation testing sections (testing the Servlet installation and testing the In-process Installation) above. These methods are invoked from the Command Prompt window menus by choosing command **110** and command **111** , and then entering path information (the methods' arguments) when prompted.

How Axis.war Finds the Deployment Descriptor Files

If you have installed Corticon Server using `Axis.war`, you may have wondered how it knows to look in the Corticon installation 's `Server\Tomcat\CcServer\cdd` directory to read and load Deployment Descriptor files (`.cdd` files).

Inside `Axis.war` is a class named `CcSoapServerInit`. The source code for this class is available in the `[CORTICON_HOME]\Server\Tomcat\CcServer\src` installation directory. Open the source file and scroll down until you see this portion:

Figure 29: loadFromCddDir() Inside CcSoapServerInit.java

```
50        try
51        {
52           //  Create a new CcServer Instance
53           ICcServer lCcServer = CcServerFactory.getCcServer();
54
55           //  Call the CcServer's loadFromCddDir...this will locate all cdd files in the
56           //  directory and load them
57           lCcServer.loadFromCddDir(lstrPath);
58        }
```

We're already familiar with `loadFromCddDir()` – it causes Corticon Server to look in the directory defined by the argument string and load any and all `.cdd` files it finds inside. This is a very convenient way to load Decision Services onto Corticon Server because one call to `loadFromCddDir()` is all you need to load multiple Decision Services on Corticon Server.

But it is less obvious where the argument for the method is coming from. Looking lower in `CcSoapServerInit.java`, we see some utility methods that define the argument.

`findCorticonDirectory()` tries to obtain the value of `autoloaddir` from `CcProperties`. `CcProperties` holds all property values from `CcConfig.jar` for that Corticon Server instance. So in effect, the method is looking into the active `CcConfig.jar` file and trying to retrieve the value of `com.corticon.ccserver.autoloaddir`. But `autoloaddir` is not defined in any of the `.properties` files inside `CcConfig.jar`.

Figure 30: Searching for CDD Path

```
84    /**
85     * Find the Corticon Cdd Directory under Apache.
86     */
87    private String findCorticonDirectory()
88    {
89       //  Check to see if the user defined an alternative directory.
90       //  If it is not there, then we default back to the Parent/cdd dir
91       String lstrAutoLoadDir =
       CcProperties.getCcProperty("com.corticon.soap.autoloaddir");
92       if (CcUtil.isValid(lstrAutoLoadDir))
93       {
94          //  Verify that the File Directory exists
95          File lFile = new File(lstrAutoLoadDir);
96          if (lFile.exists() && lFile.isDirectory())
97             return lstrAutoLoadDir;
98
99          Log.logInfo("Auto-loading of .cdd files was terminated.  Unable to find
            directory: " + lstrAutoLoadDir, this);
100         if (ibDisplayServerMessages == true)
101            System.out.println("Auto-loading of .cdd files was terminated.  Unable
               to find directory: " + lstrAutoLoadDir);
102
103         return null;
104      }
```

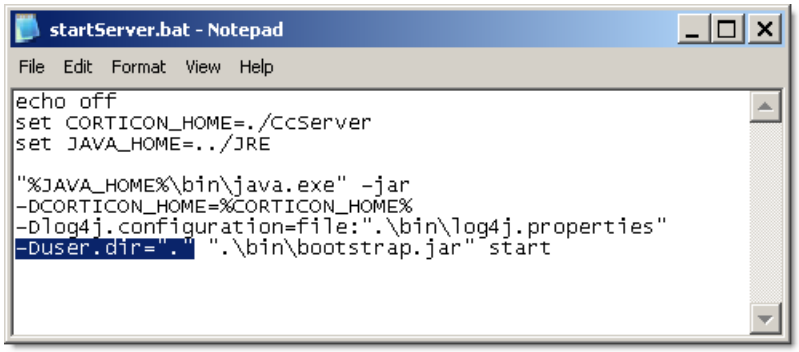
So, failing to find a value for `autoloaddir`, `findCorticonDirectory()` next assembles a pathname by combining the values of `lstrUserDir` and `cstrServerCddPath`.

Figure 31: Assembling a Path to Locate CDD Files

```
106      //  Property was not passed in...use defaults
107      String lstrUserDir = System.getProperty(cstrUserDirPropName);
108      if (CcUtil.isValid(lstrUserDir) == false)
109         return null;
110
111      String lstrPath = lstrUserDir + File.separator + cstrServerCddPath;
112
113      //  Verify that the File Directory exists
114      File lFile = new File(lstrPath);
115      if (lFile.exists() && lFile.isDirectory())
116         return lstrPath;
117
118      Log.logInfo("Auto-loading of .cdd files was terminated.  Unable to find
         directory: " + lstrPath, this);
119      if (ibDisplayServerMessages == true)
120         System.out.println("Auto-loading of .cdd files was terminated.  Unable to
            find directory: " + lstrPath);
121
122      return null;
123   }
124}
125
```

`cstrServerCddPath` is a constant defined as `CcServer/cdd` inside this class. `lstrUserDir`, on the other hand, comes from `cstrUserDirPropName`, which is a constant defined earlier in the class as `user.dir`. `User.dir` is a variable set during JVM startup. In the case of the bundled Tomcat installation, `user.dir` is defined in the `startServer.bat` file.

Figure 32: Tomcat User.dir assignment



When Tomcat starts, `user.dir` is given the value of `[CORTICON_HOME]\Server\Tomcat` by default, and then when Corticon Server starts, it retrieves this value of `user.dir` in order to assemble a pathname string to use as an argument in `loadFromCddDir()`. When all these variables are assembled, the final argument to the method is: `[CORTICON_HOME]\Server\Tomcat\CcServer\Cdd`.

Using the autoloaddir Property

Although the `autoloaddir` property isn't set by default, it is easy to add and use in your deployments. If you decide you'd rather specify your cdd path in a class rather than hard-code in your application server launch scripts, simply add the `com.corticon.ccserver.autoloaddir` property to any .properties file inside `CcConfig.jar`. A pathname will then be available when the `findCorticonDirectory()` method looks for it when `CcSoapServerInit.class` runs during Corticon Server initialization.

```
com.corticon.ccserver.autoloaddir=[CORTICON_HOME]/Server/Tomcat/CcServer/cdd
```

where `[CORTICON_HOME]` is the absolute path of the installation, typically `C:/Program Files (x86)/Progress/Corticon x.x`

Be sure to write your absolute pathname using *forward* slashes, as shown above.

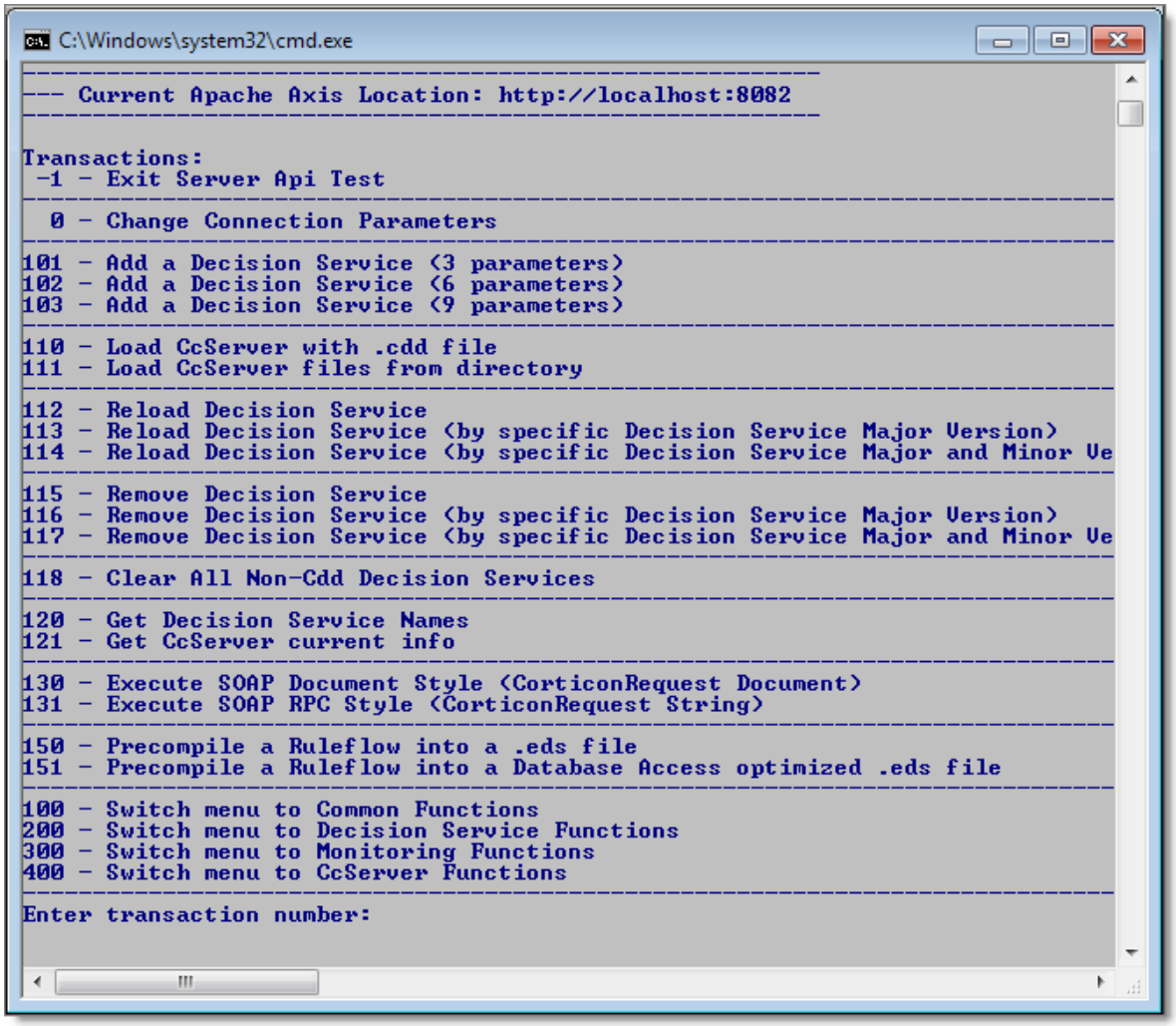
Remember, even if you chose to use the `autoloaddir` property, you will still need code in your interface class (wrapper) to read the property and insert it into the `loadFromCddDir()` method, as shown above. And you will also need code that then invokes the `loadFromCddDir()` method as shown above.

Using the Java API

While Deployment Descriptor files are a convenient way to specify the *`Ruleflow`*s to be deployed and their configuration settings, there is another way to communicate the same parameters contained in the Deployment Descriptor file to a running Corticon Server: the Java API set.

In the section above describing testing a remote installation, use a `testServerAxis` command:

Figure 33: testServerAxis.bat 100 commands



Enter command **103** named **Add a Decision Service (9 parameters)**. Enter the arguments as prompted invokes the `addDecisionService()` method. The arguments used by this method include:

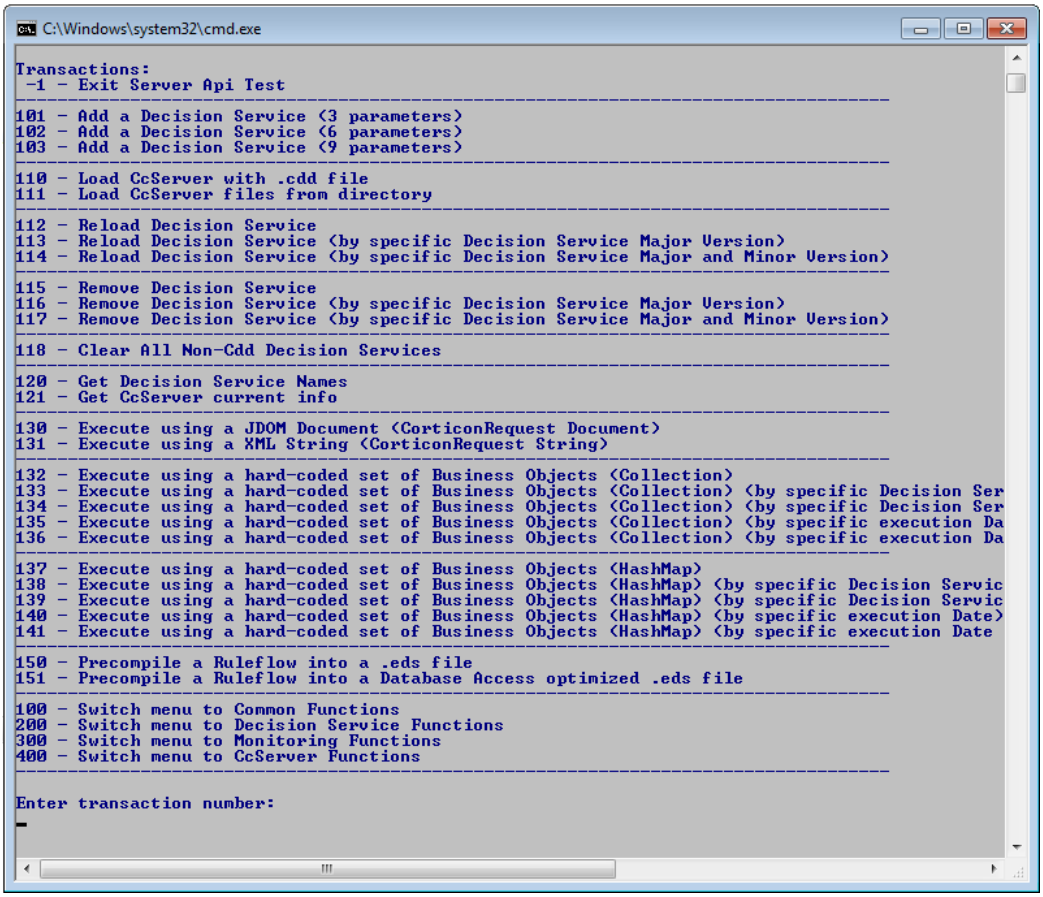
1. Decision Service Name
2. Decision Service path
3. Dynamic Reload setting (also referred to as "auto-reload")
4. Minimum Pool Size
5. Maximum Pool Size
6. Message Structure Type
7. Database Access Mode (R, RW) If you skip it, the Server will turn process against R)
8. Return Entities Mode (ALL, IN) If you skip it, the Server defaults to ALL)
9. Database Access Properties Path

If you are not using database connectivity, use command **102** named **Add a Decision Service (6 parameters)**. It asks for the first six arguments listed above.

The command **Add a Decision Service (3 parameters)** provides for backwards compatibility with previous versions of Corticon Server that had fewer features. The three parameter version, command **101**, uses the first three arguments listed above.

Note: These commands are also available for in-process:

Figure 34: testServer.bat 100 commands



Publish and Download Wizards

The following section describes the use of Publish and Download wizards.

Using the Publish wizard for Deploying Decision Services

After the rule flow sheet is defined, it is ready to be deployed as a Decision Service on the Corticon server.

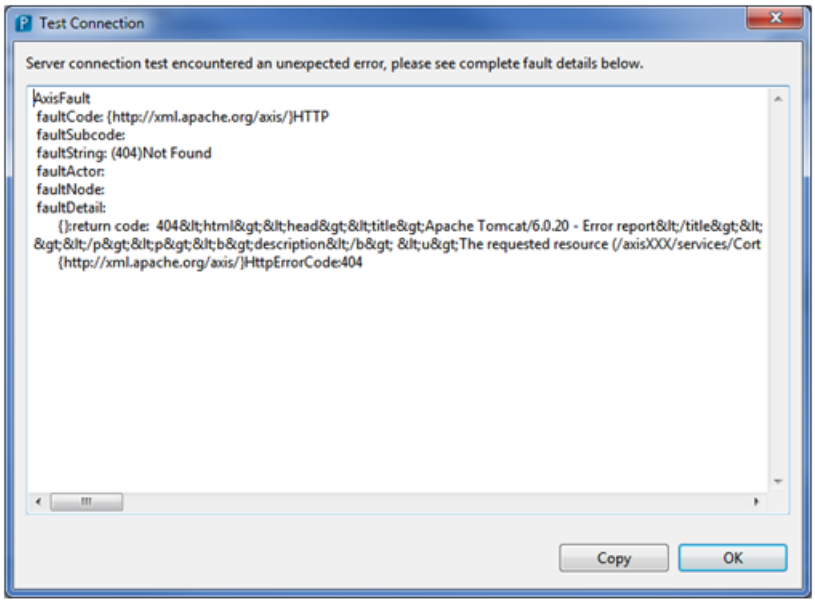
To deploy a Decision Service using the Publish wizard:

1. Switch to the Corticon perspective.
2. Select **Project** > **Publish**.
3. Enter the **Server URL** of the Corticon server.

4. Enter **Username** and **Password**.
5. Click **Test Connection**.

   - For successful connection, the system displays: **Server connection test was successful.**
   - For invalid username or password, the system displays: **User does not have rights to upload/download content to/from the server.**
   - For commonplace errors such as the server being down or unreachable, the system displays: **Server connection test failed. Server may be off-line, unreachable or not listening on specified port.**
   - For unexpected 'Hard' failure, the system displays:

     Figure 35: Unexpected Failure

     Note:

     The hard-fail handler unwraps the **Axis Fault** and finds the underlying cause. As shown in Unexpected Failure, **404** results when the user specifies a wrong URL.

6. Click **Next**.

   If the connection test is successful, the system attempts to discover Ruleflow assets and displays a progress bar reflecting the discovery.

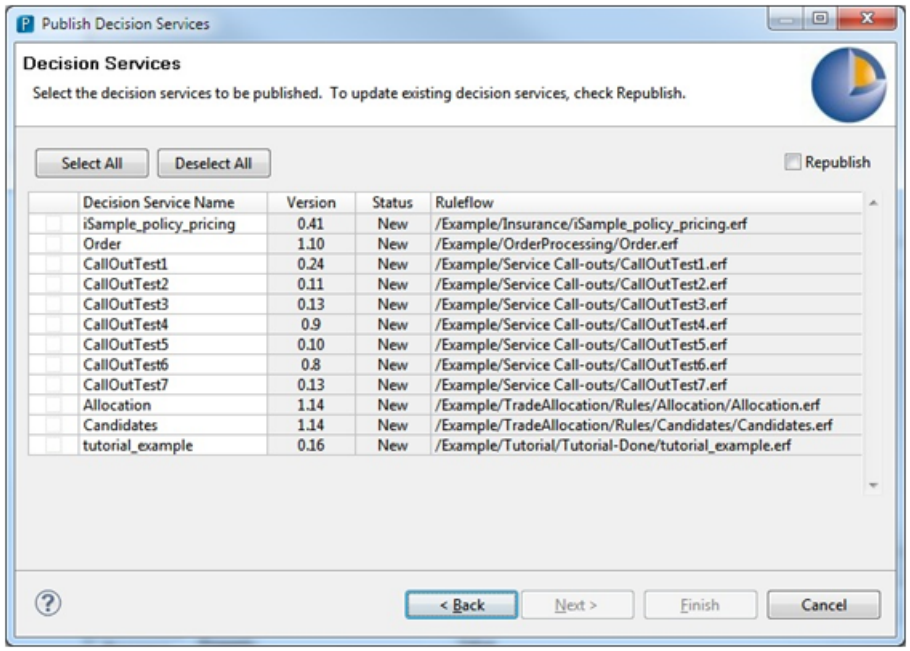
   Note:

   - The scope of the discovery process is a function of the set of selected projects in the Project Explorer and the open editors.
   - If any projects are selected in the Rule Project explorer, the system will limit the Ruleflow discovery process to only those selected projects.
   - If no projects are selected in the Rule Project explorer, the system infers the set of projects based on the open Corticon editors. The system includes all the projects containing those open assets are used.

- If no projects are selected nor any Corticon Editors are open, the system scans all open projects in the Eclipse workspace.
- The discovery process displays progress a bar. You can cancel (interrupt) the discovery process using the stop button, which is adjacent to the progress bar.

When the discovery process is complete, the system displays a list of all of the Ruleflow assets discovered as shown in Discovered Local Ruleflow Assets .

Figure 36: Discovered Local Ruleflow Assets



Note:

- The system automatically infers the Decision Service Name from the Ruleflow file name.
- The Version field is extracted from the Ruleflow Asset (Properties).
- The Status field indicates whether the decision-service/version is New or Update (that is, whether the decision-service/version is already deployed on the server).
- The Ruleflow is presented as the Eclipse workspace-relative URI.

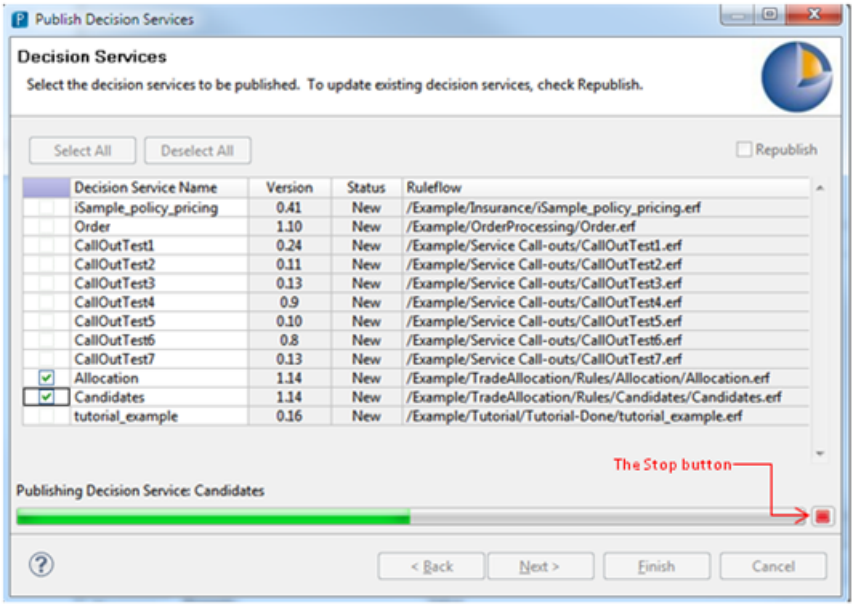
7. Select the Decision Services, as shown in Publishing Selected Decision Services, to be published and click **Finish** to publish the selected Decision Services.

   If the publishing process is successful, the system displays the **Deployment Success** message box. In case of an error, the system displays a scrolling message dialog, which bears a list of all of the errors that occurred during deployment.

   - You may optionally update the **Decision Service Name** if it is necessary to publish the decision service under a name other than that of the Ruleflow asset. When the name is changed in this manner, the system may change the **Status** field from New to Update (or vice versa) depending on whether that name is already deployed.

- The system disables the **Finish** button if a checked decision service has 'Update' as its Status value. However, the you can override this behavior by selecting the **Republish** check-box, which will cause the **Finish** button to be enabled. This is a special feature to reduce the probability of accidental harm to running decision services.

Figure 37: Publishing Selected Decision Services



Note:

The publishing process displays the progress bar as shown inPublishing Selected Decision Services . You can cancel or interrupt the process using the stop button, which is adjacent to the progress bar.

Using the Download Wizard

The Download wizard is used to download decision services from the Corticon server.

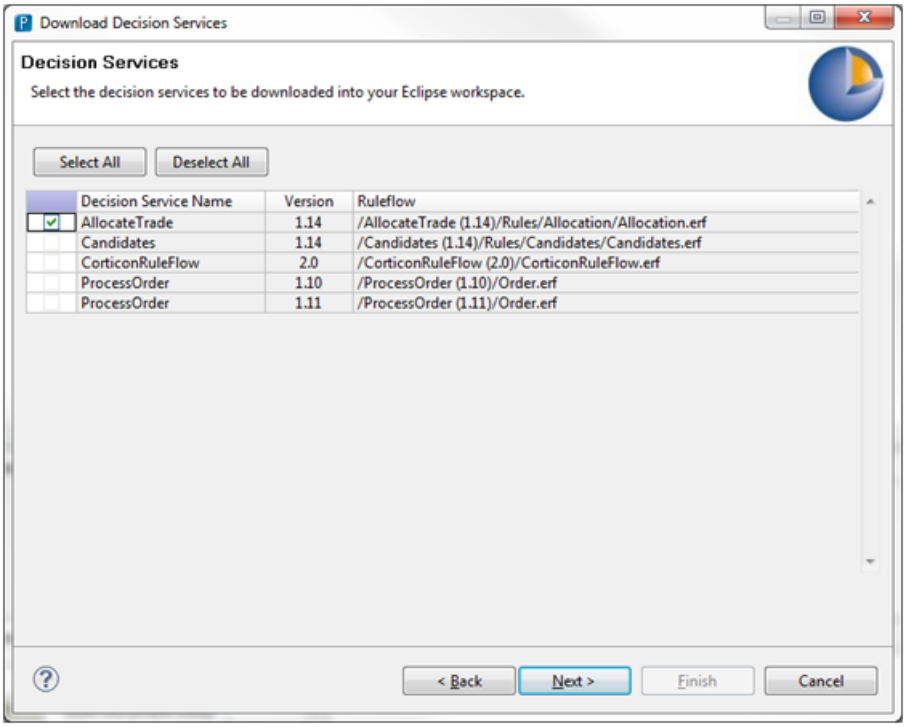
To download a decision service:

1. Select **Project** > **Download**.
2. Enter the **Server URL** of the Corticon server.
3. Enter **Username** and **Password**.
4. Click **Test Connection**.
   - For successful connection, the system displays: **Server connection test was successful.**
   - For invalid username or password, the system displays: **User does not have rights to upload/download content to/from the server.**
   - For commonplace errors such as the server being down or unreachable, the system displays: **Server connection test failed. Server may be off-line, unreachable or not listening on specified port.**
5. Click **Next**.

It populates a list of the available decision services.

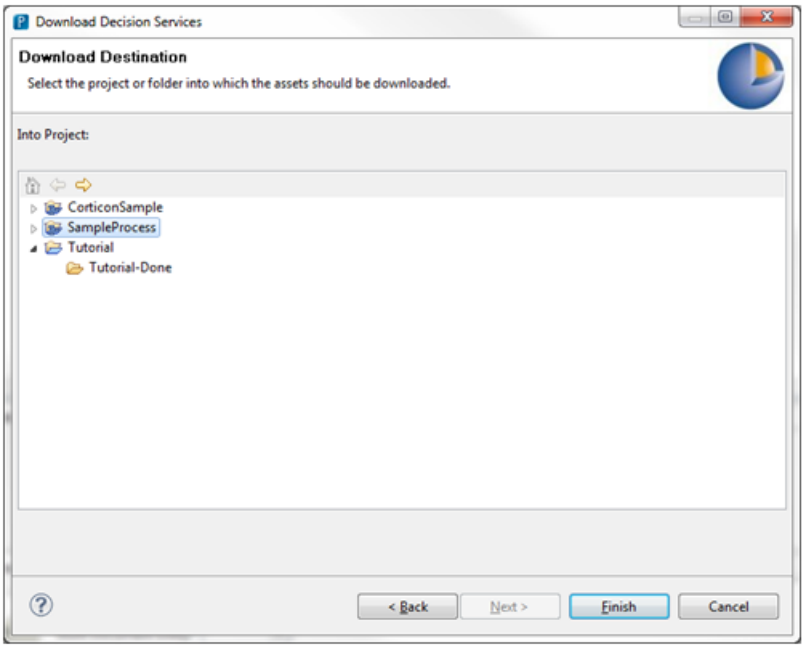
6. Select the decision service to be downloaded, as shown in Available Decision Services, and click **Next**.

Figure 38: Available Decision Services



7. Select the destination project or a folder, as shown in Download Destination for the Selected Decision Services, in which the decision service is to be downloaded and click **Finish**.

Figure 39: Download Destination for the Selected Decision Services



When you click **Finish**, the system downloads the selected assets in the selected project or folder.

Wizard Dialog Memory Settings

The Publish and Download wizards store their dialogs settings in the Eclipse workspace metadata. This allows easy reuse of the following:

1. The last used Server URL.
2. The list of the last five server URLs used, which are captured and used to populate the Server URL drop-down list.
3. The last used username and password.
4. The last selected destination folder for download.

Using the Java Server Console to Download Decision Services

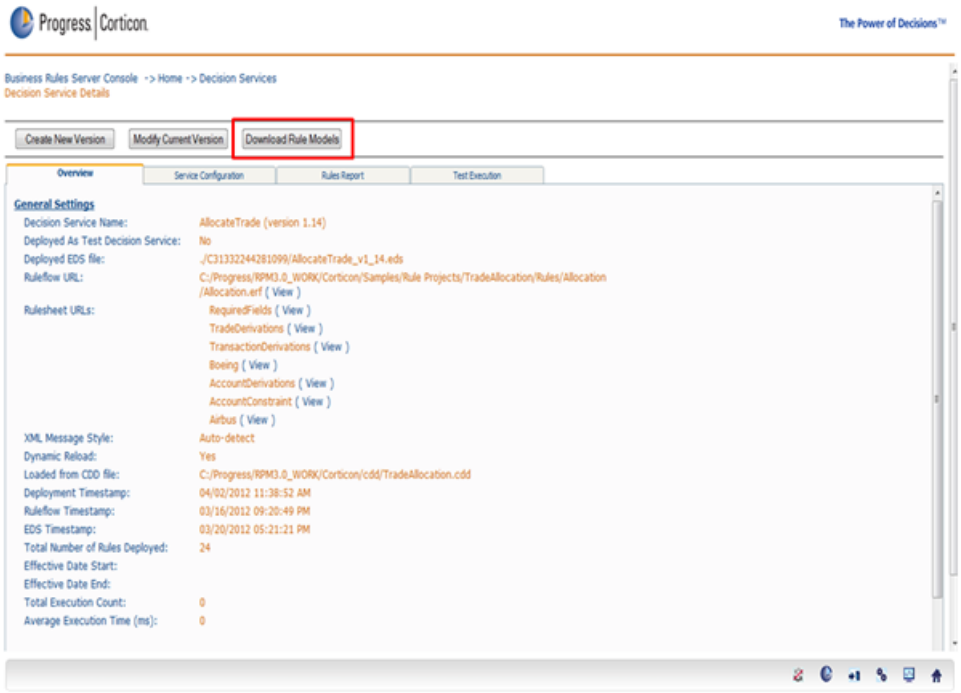
For the Corticon Java server, you can connect to its Server Console, and then click its **Download Rule Models** button to download a ZIP file to your local machine, which contains all the assets belonging to a decision service. The ZIP file preserves the folder structure and the relative relationships between the assets. The ZIP file can be extracted to the Eclipse workspace, and the assets can be opened and edited without any errors.

To download decision services using the Java Server Console:

1. In a browser, connect to the Corticon Java Server Console (`http://localhost:8082/axis`). Log in.
2. Click **Decision Services**. It displays a list of all the deployed decision services.

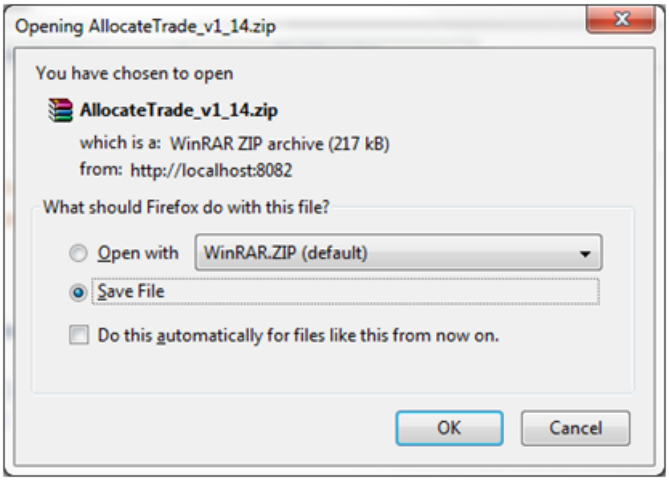
3. Click the decision service name that you want to download.
4. Click **Download Rule Models** as shown in Downloading Decision Service from the Server Console.

Figure 40: Downloading Decision Service from the Server Console



It downloads the decision service to the local machine as shown in Downloading Decision Services on to the Local Machine .

Figure 41: Downloading Decision Services on to the Local Machine



Testing the deployed Corticon Decision Service

This testing only verifies that a Decision Service is loaded on Corticon Server. It does not actually request Corticon Server to *execute* a Decision Service because we have not yet learned how to compose or deliver a full request message to Corticon Server. Actual execution will be covered in the Integrating Corticon Decision Services chapter.

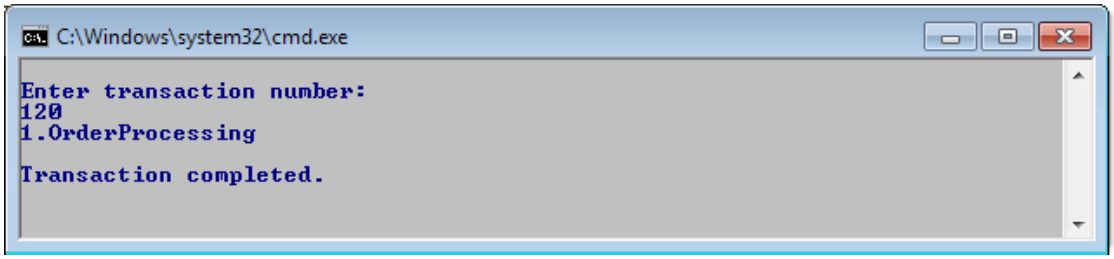
In this section, we will load Decision Services using the two methods described above, and then test the deployment using the API method `getDecisionServiceNames()`, which queries a running Corticon Server and requests the names of all deployed *Ruleflow*s.

Deploying and testing Decision Services via Deployment Descriptor files (.cdd)

Start the bundled instance of *Apache Tomcat* and allow about 10 seconds for the web services, including the Corticon Server Servlet, to start up. Now, run the Axis test batch file `[CORTICON_HOME]\Server\Tomcat\testServerAxis`.

Selecting command **120**, shown in #unique_34/unique_34_Connect_42_series100_testServerAxisTomcat on page 49, invokes the `getDecisionServiceNames()` method (this method requires no arguments).

Figure 42: Get Decision Service Names Method Success



When Corticon Server started, the Decision Service listed in Get Decision Service Names Method Success was deployed *automatically* using Deployment Descriptor files included and pre-installed in `[CORTICON_HOME]\Server\Tomcat\CcServer\cdd`. If you create your own Deployment Descriptor file and put it in this directory, the *Ruleflow*s it contains and describes will also be deployed automatically whenever Corticon Server Servlet starts up inside Tomcat.

It is important to note that Decision Services deployed using Deployment Descriptor files may only be removed from deployment or have deployment properties changed via their Deployment Descriptor file. In other words, a Decision Service deployed from Deployment Descriptor file cannot have its deployment properties changed by direct invocation of the API set; for example, if we try to use the Axis test batch file to **Remove a Decision Service** (commands **115-117** from shown in #unique_34/unique_34_Connect_42_series100_testServerAxisTomcat on page 49.

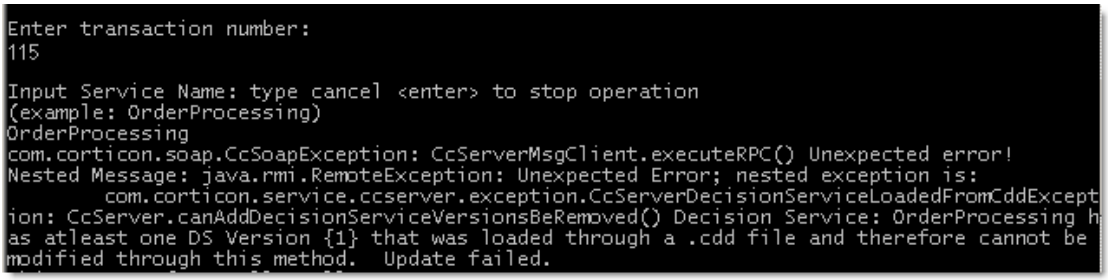
Important:

This is an important rule and worth reiterating:

The deployment settings of Decision Services deployed via Deployment Descriptor files can **only** be changed by modifying their Deployment Descriptor file. The Java API methods have no effect on these Decision Services.

The deployment settings of Decision Services deployed via Java API methods can **only** be changed via Java API methods. Deployment Descriptor files have no effect on these Decision Services.

Figure 43: Remove Decision Service Method Failure



In **Remove Decision Service Method Failure**, above, we see that the attempt to remove `OrderProcessing`, the Decision Service deployed by `OrderProcessing.cdd` (located in `[CORTICON_HOME]\Server\Tomcat\CcServer\cdd`) has failed. To remove this Decision Service from deployment, or to modify any of its deployment settings, we must return to the Deployment Descriptor file, make the changes, and then allow the Corticon Server to update via its **Dynamic Reload** feature.

Deploying and testing Decision Services via APIs

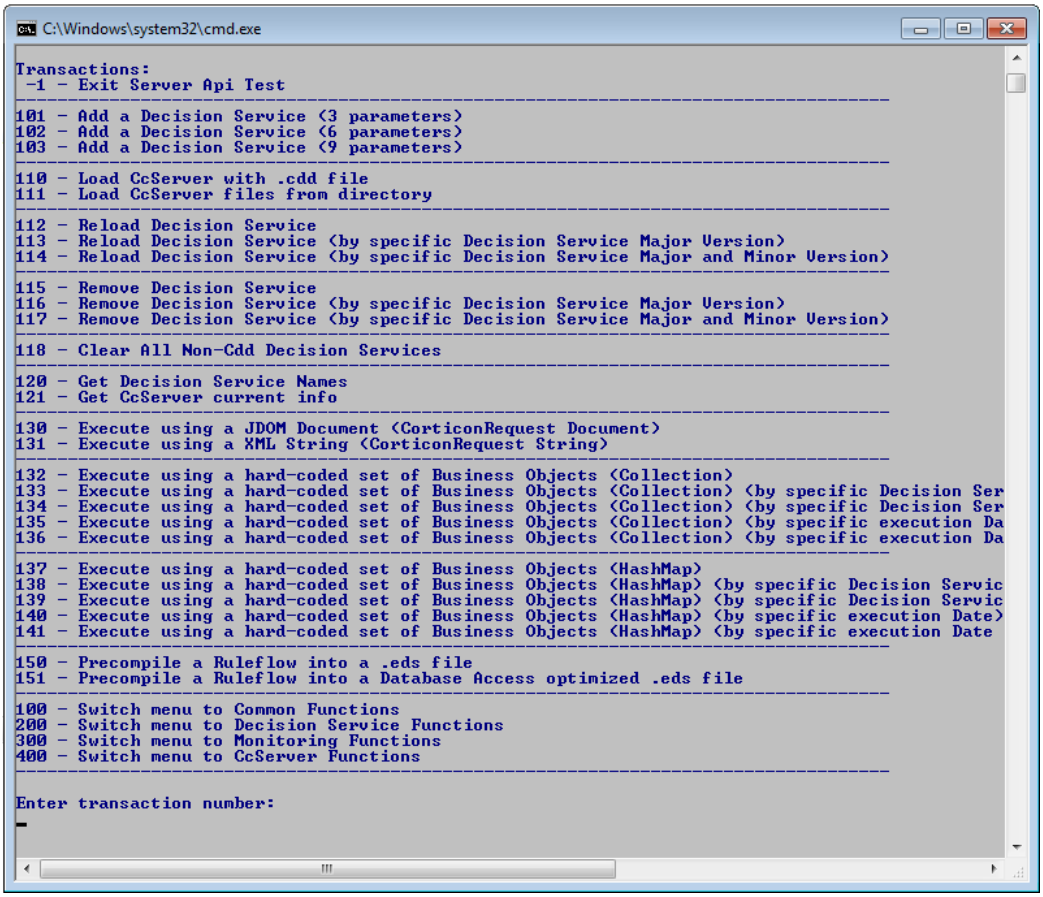
Rather than remove any Deployment Descriptor files already located by default in `[CORTICON_HOME]\Server\Tomcat\CcServer\cdd`, we will instead use the in-process test batch file to:

1. Verify that no Decision Services are deployed.
2. Deploy a new Decision Service.
3. Verify that the Decision Service is loaded.
4. Remove the Decision Service from Deployment.
5. Verify that the Decision Service has been removed from deployment.

Verifying no Decision Services are loaded

As shown:

Figure 44: testServer.bat 100 commands



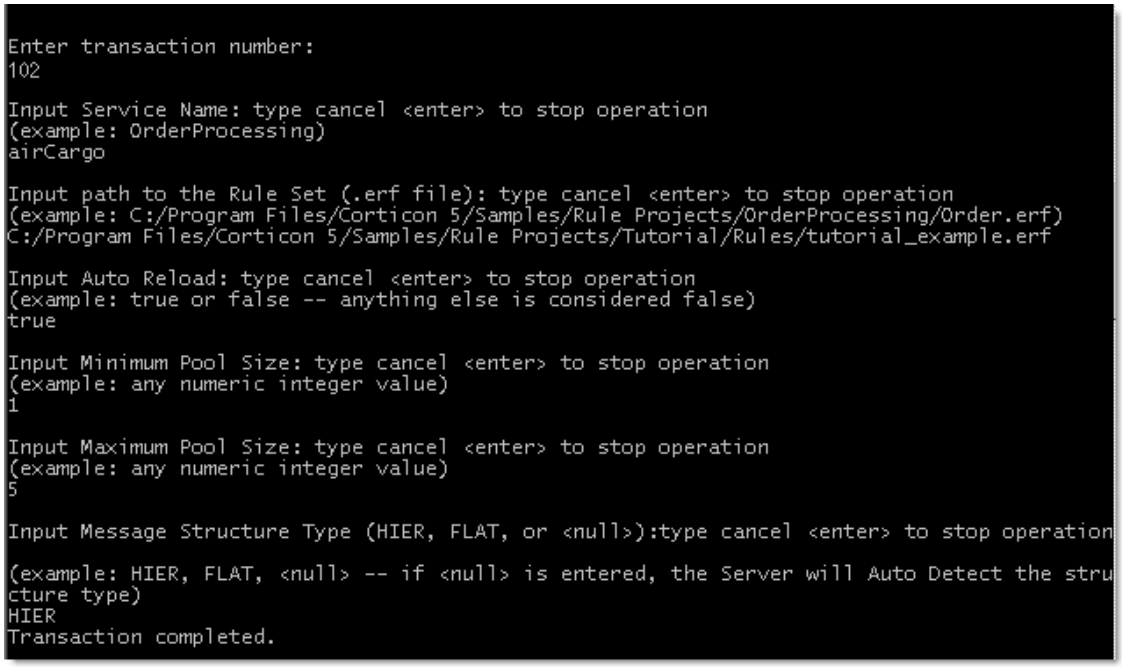
Enter command **120** requests the names of deployed Decision Services:

Figure 45: Get Decision Service Names Method Invoked

The response in Get Decision Service Names Method Invoked shows that no Decision Services are currently deployed to *Corticon Server* running in-process.

Now, let's load the Decision Service named `tutorial_example.erf` located in `[CORTICON_WORK_DIR]\samples\Rule Projects\Tutorial\Tutorial-Done\` using the 6 parameter load method (option **102**) in #unique_34/unique_34_Connect_42_series100_testServerTomcat on page 50

Figure 46: Add a Decision Service Method Invoked



Notice that the 6 parameters we entered are the same 6 entries in the list of deployment properties described in the Deployment Descriptor file section, Left Portion of Deployment Console, with Deployment Descriptor File Options Numbered and Right Portion of Deployment Console, with Deployment Descriptor File Options Numbered. The very first parameter, Decision Service Name, has been given the value of `airCargo`.

You may notice a pause before the `Transaction completed` message is displayed. This is due to the deployment-time compilation performed by *`Corticon Server`*. Larger *`Ruleflow`*s (more *`Rulesheet`*s or rules) will require more time to compile.

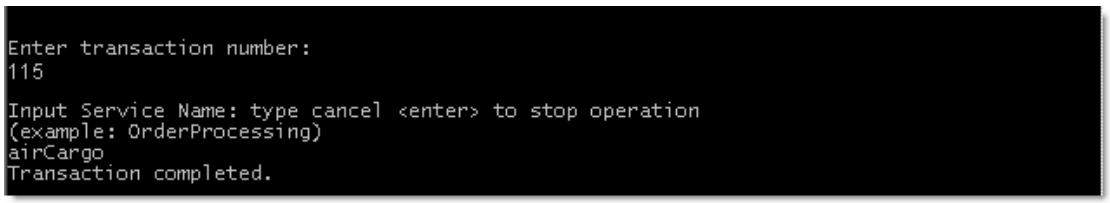
Now, to verify that this Decision Service is deployed, we will re-invoke the `getDecisionServiceNames()` method using command **120**.

Figure 47: Get Decision Service Names Method, Re-invoked

That figure shows that the Decision Service named `airCargo` is deployed to Corticon Server.

Now, to remove airCargo from deployment, we will use command **115**.

Figure 48: Remove Decision Service Method



And then to verify that `airCargo` has been removed from *`Corticon Server`*, we invoke `getDecisionServiceNames()` once again using command **120**:

Figure 49: Get Decision Service Names Method

That figure shows that `airCargo` has successfully been removed from deployment.

Deploying uncompiled vs. pre-compiled decision services

You have the choice to deploy uncompiled `.erf` files or pre-compiled `.eds` files as Decision Services. There are advantages and disadvantages to each method, which result from the basic difference between the two file types:

ERF files

The `.erf` file is really just a "pointer" file that informs Corticon Server where, specifically, to look for the component parts of the Decision Service, including the `.ecore` and all included `.ers` files. This has the following important consequences:

- All of the component files (`.ecore` and `.ers`) must be accessible to Corticon Server, so that it is able to read them during compilation. This may mean you need to keep tighter access controls on the various component files involved, which may be less convenient in production environments.
- `.erf` compilation occurs "on-the-fly", which may take a few seconds when first deployed
- If an `.ers` file is shared among multiple `.erf` files, then a change to the `.ers` will cause `.erf`'s to update also if their auto-reload property is set to `true` and Corticon Server's dynamic monitoring update service is running.

Because of these considerations, uncompiled Decision Service deployments are often used in non-production environments, where component files (`.ecore` and `.ers`) are more likely to change frequently or require looser controls.

EDS Files

The `.eds` file is a self-contained, complete deployment asset that includes compiled versions of all component files, including `.ecore` and `.ers` file. This has the following important consequences:

- Only the `.eds` file needs to be accessible to Corticon Server. Original component files (`.ecore` and `.ers`) can be archived offline and accessed only when changes need to be made to them.
- `.eds` files are already compiled, so Corticon Server can load them quickly upon deployment, without the lag time required by `.erf` on-the-fly compilation.
- if a rule changes inside a component `.ers` file, then the `.eds` must be recompiled and redeployed. The Corticon Server's dynamic monitoring update service will detect changes only to the `.eds` dateTimeStamp, not to the dateTimeStamp changes of any internal component files.

Because of these considerations, pre-compiled Decision Service deployments are often used in production environments, where component files (`.ecore` and `.ers`) are less likely to change frequently or require tighter controls.

If your `.erf` contains Service Call-Outs (SCOs), be sure to add the SCO classes to `deployConsole.bat` so that they are included in the classpath when the `.eds` is compiled. More information about SCOs can be found in the *Extensions User Guide*.

5

Integrating Corticon Decision Services

This chapter explains how to correctly call or "consume" a Decision Service. Corticon `Ruleflow`s, once deployed to a Corticon Server, are services, a fact we emphasize by calling them *Decision Services*.

Services in a true Service Oriented Architecture have established ways of receiving requests and sending responses. It is important to understand and correctly implement these standard ways of sending and receiving information from Decision Services.

In this chapter, we will not actually make a call to a deployed Decision Service – that will come in the Invocation chapter that follows. Instead, we focus on the types of calls, their components, and the tools available to help you assemble them.

For details, see the following topics:

Components of a call to Corticon Server

Before going any further, let's clarify what "calling a Decision Service" really means. Technically, we will be making an "execute" call to, or invocation of, Corticon Server. The call/invocation/request (we will use these three terms interchangeably) consists of:

- The name and location (URL) of the Corticon Server we want to call.
- The name of the Decision Service we want Corticon Server to execute.
- The data needed by Corticon Server to process the rules inside the Decision Service, structured in a way Corticon Server can understand. We often call this the "payload".

The name and location of Corticon Server we want to call will be discussed in the Invocation chapter, since this information is concerned more with protocol than with content. The focus of this chapter will be on the other two items, Decision Service Name and data payload.

The Decision Service Name

The name of the Decision Service has already been established during deployment. Assigning a name to a Decision Service can be accomplished through a Deployment Descriptor file, shown in Left Portion of Deployment Console, with Deployment Descriptor File Options Numbered. Or it can be defined as an argument of the API deployment method `addDecisionService()`. Both current versions of this API method (the 6 and 9 argument commands) as well as the legacy 3 argument command, require the Decision Service Name argument. Once deployed, the Decision Service will always be known, referenced and invoked in runtime by this name. Decision Service Names must be unique, although multiple *versions* of the same Decision Service Name may be deployed and invoked simultaneously. See Versioning for more details.

The Data

While the data itself may vary for a given Decision Service from transaction to transaction and call to call, the **structure** of that data – how it is arranged and organized – must not vary. The data contained in each call must be structured in a way Corticon Server expects and can understand. Likewise, when Corticon Server executes a Decision Service and responds to the client with new data, that data too must be structured in a consistent manner. If not, then the client or calling application will not understand it.

Service contracts: Describing the call

Generically, a service contract defines the interface to a service, informing consuming client applications what they must send to it (the type and structure of the *input* data) and what they can expect to receive in return (the type and structure of the *output* data). If a service contract conforms to a standardized format, it can often be analyzed by consuming applications, which can then generate, populate and send compliant service requests automatically.

Web Services standards define two such service contract formats, the Web Services Description Language, or WSDL (sometimes pronounced "wiz-dull") and the XML Schema (sometimes known as an XSD because of its file extension, `.xsd`). Because both the WSDL and XSD are physical documents describing the service contract for a specific Web Service, they are known as *explicit* service contracts. A Java service may also have a service contract, or interface, but no standard description exists for an explicit service contract. Therefore, most service contract discussions in this chapter relate to Web Services deployments only.

Depending on the choice of architecture made earlier, you have two options when representing data in a call to Corticon Server: an XML document or a set of Java Business Objects.

XML workDocument

If you chose Option 1 or 2 in Table 1, then the payload of your call will have the form of an XML document. Full details on the structure of these two service contract options (WSDL and XSD) and their variations can be found in section XML Service Contract Descriptions and examples can be found in Service contract and message samples on page 159.

Java business objects

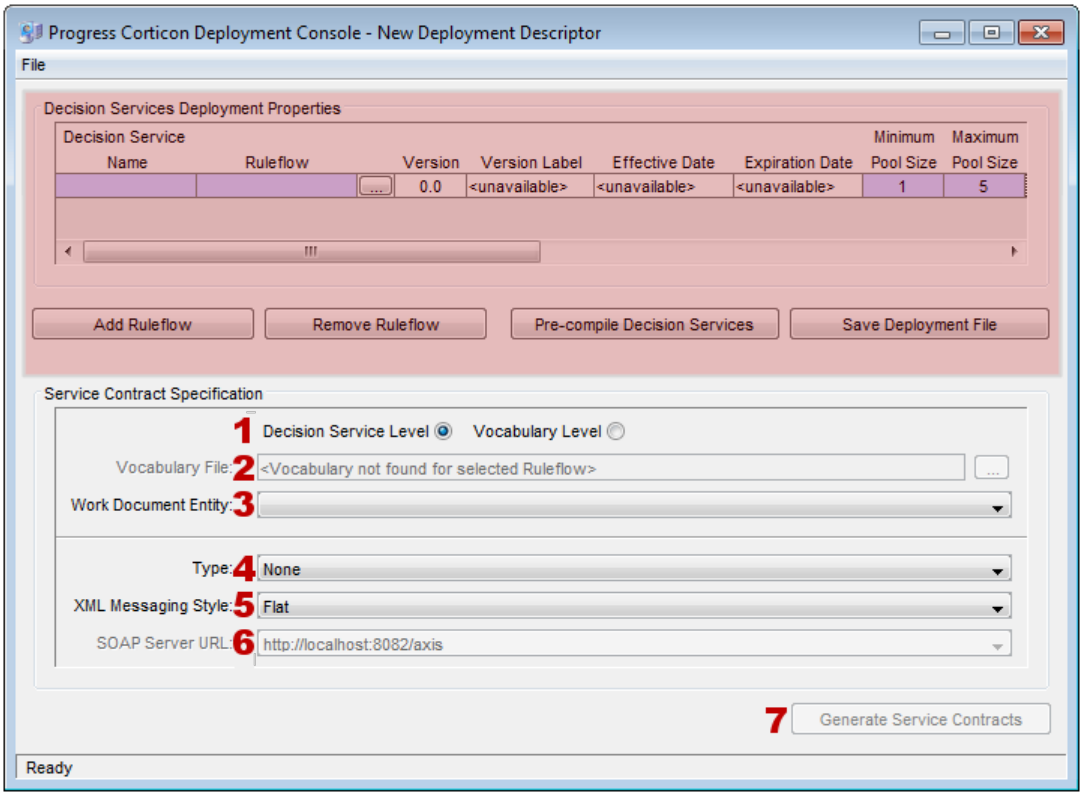
Unfortunately, no standard method of describing a service contract for a Java service yet exists, so Corticon Studio provides no tools for generating such contracts.

Creating XML service contracts with Corticon Deployment Console

The earlier Deployment chapter introduced the *Deployment Console* as a way to create Deployment Descriptor files to easily deploy Decision Services and manage their deployment settings. The screenshot in Using the Deployment Console tool's Decision Services on page 40 hides the lower portion of the *Deployment Console*, however, because that portion is not concerned with Deployment Descriptor file generation, which is the focus of that chapter. Now is the time to discuss the lower portion of the *Deployment Console*.

Instructions for starting or launching the Deployment Console are located in the earlier Deployment chapter.

Figure 50: Deployment Console with Input Options Numbered



Service contracts provide a published XML interface to Decision Services. They inform consumers of the necessary input data and structure required by the Decision Service and of the data structure the consumer can expect as output.

1. **Decision Service Level/Vocabulary Level** - These radio buttons determine whether one service contract is generated for each `Ruleflow` listed in the Deployment Descriptor section of the Deployment Console (the upper portion), or for the Vocabulary listed in section **Vocabulary File**.

   Often, the same payload structure flows through many decision steps in a business process. While any given Decision Service might use only a fraction of the payload's content (and therefore have a more efficient invocation), it is sometimes convenient to create a single "master" service contract from the Decision Service's Vocabulary. This simplifies the task of integrating the Decision Services into the business process because a request message conforming to the master service contract can be used to invoke any and all Decision Services that were built with that Vocabulary. This master service contract, one which encompasses the entire Vocabulary, is called **Vocabulary Level**.

   The downside to the Vocabulary-level service contract is its size. Any request message generated from a Vocabulary-level service contract will contain the XML structure for *every term* in the Vocabulary, even if a given Decision Service only requires a small fraction of that structure. Use of a Vocabulary-level service contract therefore introduces extra overhead because request messages generated from it may be unnecessarily large.

   In an application or process where performance is a higher priority than integration flexibility, using a **Decision Service Level** service contract is more appropriate. A Decision Service-level service contract contains the bare minimum structure necessary to consume that specific Decision Service – no more, no less. A request message generated from this service contract will be the most compact possible, resulting in less network overhead and better overall system performance. But it may not be reusable for other Decision Services.

2. **Vocabulary File** - When generating a Vocabulary-level service contract, enter the Vocabulary file name (`.ecore`) here. When generating a Decision Service-level contract, this field is read-only and shows the Vocabulary associated with the currently highlighted `Ruleflow` row above. See Corticon Decision Service Contracts for details.

3. **Select Work Document Entity** - Before any service contract can be generated, a Work Document Entity (WDE) must be chosen. The WDE serves as the root context for the XML document, whether flat or hierarchical in format. To assign a WDE in Corticon Studio, use menubar option **Ruleflow** > **Properties** and select from the drop-down in the **Properties** window.

   A Decision Service invocation should contain a data payload, and the WDE is simply a way of defining the minimum payload necessary. The entity selected as WDE becomes a mandatory element in the invocation.

   If generating a Vocabulary-level service contract, select a WDE from the list. If generating a Decision Service-level contract, select the `Ruleflow` row in the Deployment Descriptor file section to highlight it, then select a WDE from the list. Do this for each `Ruleflow`. If a WDE has already been selected for a `Ruleflow`, it should appear in this field. If it has already been specified, you can change it here and the `Ruleflow` file will also be updated.

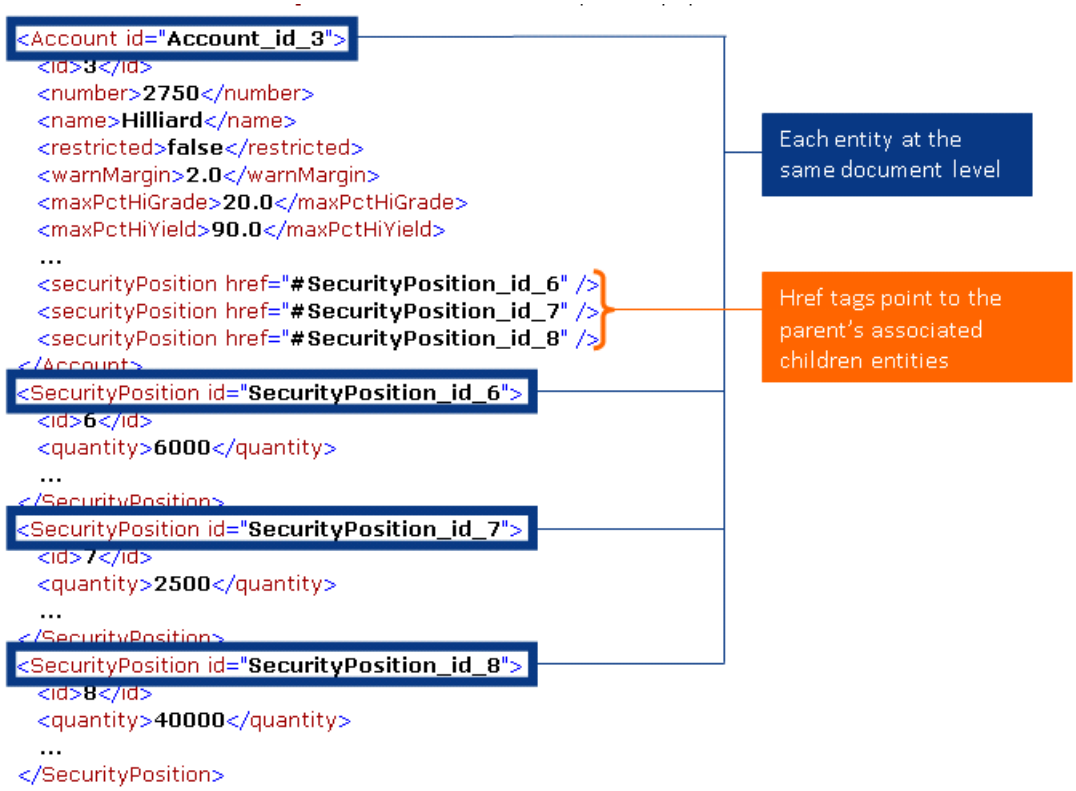
4. **Type** - The service contract type: WSDL or XML Schema. A WSDL can also be created from within Corticon Studio with the menu command **Ruletest** > **Testsheet** > **Data** > **Export WSDL**. See the `Ruletest` chapter of the *Corticon Studio: Quick Reference Guide* for more information.

5. **XML Messaging Style** - When using XML to describe the payload, there are two structural styles the payload may take, "flat" and "hierarchical". Flat payloads have every entity instance at the top ("root") level with all associations represented by reference. Hierarchical payloads

represent associations with child entity instances embedded or “indented” within the parent entity structure.

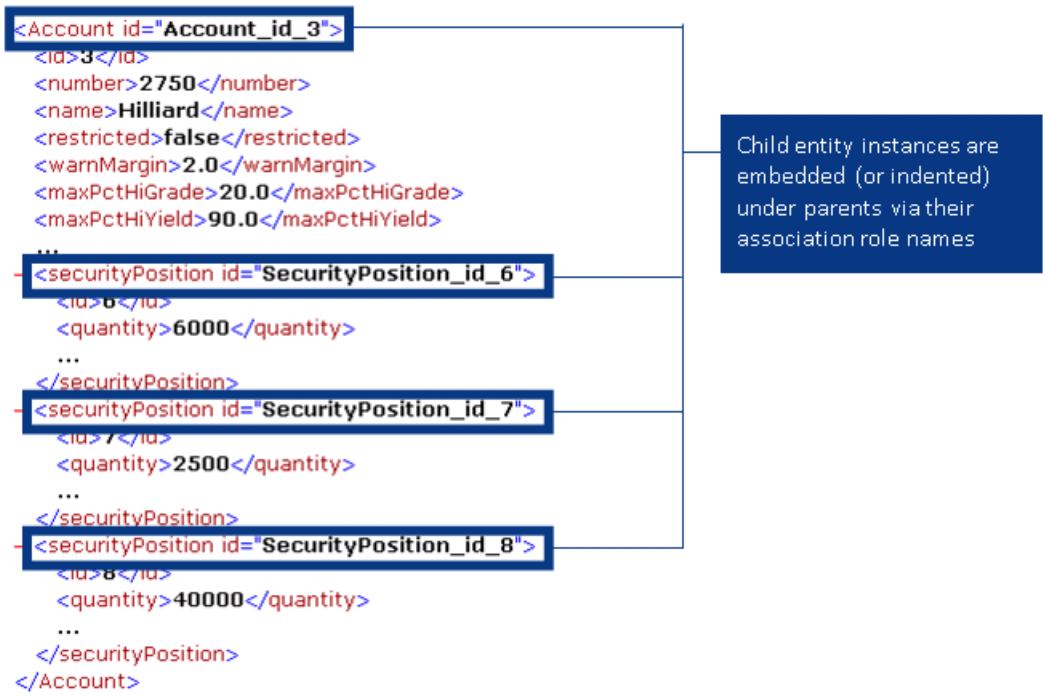
Both styles are illustrated below. Assume a Decision Service uses an `Account` and its associated `SecurityPositions`. In the Flat style, the payload is structured as:

Figure 51: Example of a Flat (FLAT) Message



In the **hierarchical** style, the payload is structured as:

Figure 52: Example of a Hierarchical (HIER) Message



The Hierarchical style need not be *solely* hierarchical; some associations may be expressed as flat, others as hierarchical. In other words, hierarchical can really be a *mixture* of both flat and hierarchical. A mixture of hierarchical and flat elements is sometimes called a *hybrid* style.

Note that in the flat style, all entity names start with an upper case letter. Associations are represented by `href` tags and role names which appear with lowercase first letters. By contrast, in the hierarchical style, an embedded entity is identified by the role name representing that nested relationship (again, starting with a lowercase letter). Role names are defined in the Vocabulary.

This option is enabled only for Vocabulary-level service contracts, because the message style for Decision Service-level service contracts is specified in the Deployment Descriptor file section (the upper portion) of the Deployment Console.

6. **SOAP Server URL** - URL for the SOAP node that is bound to the Corticon Server. Enabled for WSDL service contracts only. The default URL is `http://localhost:8082/axis/services/Corticon` that makes a Decision Service available to the default Corticon Server's Tomcat installation. This default may be changed in `CcDeployment.properties` – see CcDeployment.Properties.

7. **Generate Service Contracts** - Generates the WSDL or XML Schema service contracts into the output directory. When you select Decision Service-level contracts, one service contract per *`Ruleflow`* listed in the Deployment Descriptor section (the upper portion) of the Deployment Console will be created. When you select Vocabulary-level service contracts, only one contract is created for the Vocabulary file specified in section **1**. Note that this button is not enabled until you have chosen a **Type**.

Types of XML service contracts

XML Schema (XSD)

The purpose of an XML Schema is to define the legal or "valid" structure of an XML document. In our case, this XML document will carry the data required by Corticon Server to execute a specified Decision Service. The XML document described by an XSD is the payload (the data and structure of that data) of a SOAP call to the Corticon Server, or may also be used as the payload of a Java API call or invocation.

XSD, by itself, is only a method for describing payload structure and contents. It is not a protocol that describes how a client or consumer goes about invoking a Decision Service; instead, it describes what the data inside that request must look like.

For more information on XML Schemas, see
`http://www.w3schools.com/schema/schema_intro.asp`

Web Services Description Language (WSDL)

A WSDL service contract differs from the XSD in that it defines both invocation payload and protocol. It is the easiest as well as the most common way to integrate with a *`Web Services Server`*. The WSDL file defines a complete interface, including: [1] For more information on WSDL, see
`http://www.w3schools.com/wsdl/default.asp`

- Corticon Server SOAP binding parameters.
- Decision Service Name.
- Payload, or XML data elements required inside the request message (this portion of the WSDL is identical to the XSD).
- XML data elements provided within the response message.
- The Web Services standard allows for two messaging styles between services and their consumers: RPC-style and Document-style. Document-style, sometimes also called Message-style, interactions are more suitable for Decision Service consumption because of the richness and (potential) complexity common in business. RPC-style interactions are more suitable for simple services that require a fixed parameter list and return a single result. As a result,

---

Important: Corticon Decision Service WSDLs are always Document-style! If you intend to use a commercially available software toolset to import WSDL documents and generate request messages, be sure the toolset contains support for Document-style WSDLs.

Annotated Examples of XSD and WSDLs Available in the Deployment Console

The eight variations of service contract documents generated by the Corticon Deployment Console are shown and annotated in Service contract and message samples on page 159. The table below provides direct links to each variation.

Section	Type	Level	Style
1	XSD	Vocabulary	Flat
2	XSD	Vocabulary	Hierarchical
3	XSD	Decision Service	Flat
4	XSD	Decision Service	Hierarchical
5	WSDL	Vocabulary	Flat
6	WSDL	Vocabulary	Hierarchical
7	WSDL	Decision Service	Flat
8	WSDL	Decision Service	Hierarchical

Passing null values in XML messages

Passing a null value to Corticon Server using XML payloads is accomplished in the following ways:

Vocabulary Type	How to Pass As Null
An attribute of any type	Omit the XML tag for the attribute, or Use the XSD special value of `xsi:nil='1'` as the attribute's value.
An attribute except String types	Include the XML tag for the attribute but do not follow it with a value, for example, `<weight></weight>` or simply `<weight/>`. If the type is String, this form is treated as an empty string (a string of length zero, which is not the same as null).
An association	Do not include an `href` to a potentially associable Entity (Flat model) or do not include the potentially associable role in a nested child relationship to its parent.
An entity	Omit the `complexType` from the payload entirely.

6

Invoking Corticon Server

The previous chapter discussed the *contents* of a call to Corticon Server, and what those contents need to look like. This chapter discusses the options available to make the actual call itself, and the types of call to which a Corticon Server can respond.

For details, see the following topics:

Methods of calling Corticon Server

There are two ways to call Corticon Server:

- A Web Services (SOAP) request message.
- A Java method invocation.

SOAP call

The structure and contents of this message are described in the Integration chapter. Once the SOAP request message has been prepared, a SOAP client must transmit the message to the Corticon Server (deployed as a Web Service) via HTTP.

If your SOAP tools have the ability to assemble a compliant request message from a WSDL you generated with the Corticon Deployment Console, then it is very likely they can also act as a SOAP client and transmit the message to a Corticon Server deployed to the bundled Apache Tomcat web server. The *Corticon Server: Deploying Web Services for Java* describes shows how to test this method of invocation with a third-party tool or application (like Altova XMLSpy, for example) as a SOAP client.

When developing and testing SOAP messaging, it is very helpful to use some type of message interception tool (such as tcpTunnelUI or TCPTrace) that allows you to "grab" a copy of the request message as it leaves the client and the response message as it leaves Corticon Server. This lets you inspect the messages and ensure no unintended changes have occurred, especially on the client-side.

Java call

The specific method used to invoke a Decision Service is the `execute()` method. The method offers a choice of arguments:

- An XML string, which contains the Decision Service Name as well as the payload data. The payload data must be structured according to the XSD generated by the *`Deployment Console`*. Defining this data payload structure to include as an argument to the execute method is the most common use of the XSD service contract.
- A JDOM XML document, which contains the Decision Service Name as well as the payload data (array).
- The Decision Service Name plus a collection of Java business objects which represent the WorkDocuments.
- The Decision Service Name plus a map of Java business objects which represent the WorkDocuments.

Optional arguments representing Decision Service Version and Effective Timestamp may also be included – these are described in the Versioning and Effective Dating chapter of this manual.

All variations of the `execute()` method are described fully in the *JavaDoc*.

These arguments are passed by reference.

Vocabulary Term	Corresponding Java Construct
Entity	Java class (JavaBean compliant)
Attribute	Java property within a class
Association	Class reference to another class

For example, in the *Corticon Studio: Basic Rule Modeling Tutorial*, the three Vocabulary entities: `FlightPlan`, `Cargo`, `Aircraft` would be represented by the consumer as three Java classes. Each class would have properties corresponding to the Vocabulary entity attributes (for example, `volume`, `weight`). The association between `Cargo` and `FlightPlan` would be handled by Java class properties containing object references; the same would be true for the association between `Aircraft` and `FlightPlan`.

Note that even if there is only a one-way reference between two classes participating in an association (from `FlightPlan` to `Cargo`, for example), if the association is defined as bidirectional in the Vocabulary, rules may be written to traverse the association in either direction. Bidirectionality is *asserted* by Corticon Server even if the Java association is unidirectional (as most are, due to inherent synchronization challenges with bidirectional associations in Java objects).

Use the same `testServer.bat` (located in `[CORTICON_HOME]\Server`) to see how the `execute()` method is used with Java objects. In addition to the 6 base Corticon Server JARs, the batch file also loads some hard-coded Java business objects for use with the Java Object Messaging options (menu option **132-141** in #unique_34/unique_34_Connect_42_series100_testServerTomcat on page 50. These hard-coded classes are included in `CcServer.jar` so as to ensure their inclusion on the JVM's classpath whenever `CcServer.jar` is loaded. The hard-coded Java objects are intended for use when invoking the `OrderProcessing.erf` Decision Service included in the default installation.

Request/Response mode

Regardless of which invocation method you choose to call Corticon Server, keep in mind that it, by default, acts in a "request—response" mode. This means that one request sent from the client to Corticon Server will result in one response sent by the Server back to the client. Multiple calls may be made by different clients simultaneously, and the Server will assign these requests to different Reactors in the pool as appropriate. As each Reactor completes its transaction, the response will be sent back to the client.

Also, the form of the response will be the same as the request: if the request is a web services call (SOAP message), then the response will be as well. If a Java application uses the `execute()` method to transmit a map of objects, then will also return the results in a map.

Administrative APIs

In addition to the `execute()` method (and its variations), a set of administrative APIs allows client control over most Corticon Server functions. These methods are described in more detail in the *JavaDoc*, `CcServerAdminInterface` class, including:

- Adds (deploying) a specific Decision Service onto Corticon Server (`addDecisionService`) without using a `.cdd` file. Available in 3, 6, and 9 parameter versions.
- Removes all Decision Services which were loaded (deployed) via the `addDecisionService` method (`clearAllNonCddDecisionServices`). Does not affect Decision Services deployed via a `.cdd` file.
- Queries Corticon Server for admin information such as version number, deployed Decision, and Service settings. (`getCcServerInfo`).
- Queries Corticon Server for a list of loaded (deployed) Decision Service Names (`getDecisionServiceNames`)
- Initializes Corticon Server, causing it to start up and restore state from `ServerState.xml` (`initialize`)
- Queries Corticon Server to see if a Decision Service Name (or specific version or effective date) is deployed (`isDecisionServiceDeployed`)
- Instructs Corticon Server to load all Decision Services in a specific `.cdd` file (`loadFromCdd`)
- Instructs Corticon Server to load all Decision Services from all `.cdd` files located in a specific directory (`loadFromCddDir`)
- Changes the auto-reload setting for a Decision Service (or specific version) (`modifyDecisionServiceAutoReload`). Does not affect Decision Services deployed via a `.cdd` file.
- Changes the message structure type setting (FLAT or HIER) for a Decision Service (or specific version) (`modifyDecisionServiceMessageStructType`). Does not affect Decision Services deployed via a `.cdd` file.
- Changes the min and/or max pool settings for a Decision Service (or specific version) (`modifyDecisionServicePoolSizes`). Does not affect Decision Services deployed via a `.cdd` file.
- Changes the path of a deployed Decision Service (Ruleflow) (`modifyDecisionServiceRuleflowPath`). Does not affect Decision Services deployed via a `.cdd` file.
- Instructs Corticon Server to dump and reload (refresh) a specific Decision Service (or version) (`reloadDecisionService`).
- Removes (undeploying) a Decision Service (or specific version) (`removeDecisionService`). Does not affect Decision Services deployed via a `.cdd` file.
- Sets Corticon Server's Log level (`setLogLevel`). This is the same Log level that can also be set statically using CcCommon.properties inside `CcConfig.jar`. All properties files are described in Configuring Corticon properties and settings on page 185.
- Sets Corticon Server's Log path (`setLogPath`). This is the same Log path that can also be set statically using CcCommon.properties inside `CcConfig.jar`. All properties files are described in Configuring Corticon properties and settings on page 185.
- Starts Corticon Server's Dynamic Update monitoring service (`startDynamicUpdateMonitoringService`). This is the same update service that can also be set statically using CcServer.properties inside `CcConfig.jar`. All properties files are described in Configuring Corticon properties and settings on page 185.

- Stops Corticon Server's Dynamic Update monitoring service (`stopDynamicUpdateMonitoringService`). This is the same update service that can also be set statically using CcServer .properties inside `CcConfig.jar`. All properties files are described in Configuring Corticon properties and settings on page 185.

Important: A Decision Service deployed using a .cdd file may only have its deployment setting changed by modifying the .cdd file. A Decision Service deployed using APIs may only have its deployment settings modified by APIs.

All APIs are available as both Java methods (described fully in the *JavaDoc*) and as operations in a SOAP request message. Corticon provides a WSDL containing full descriptions of each of these methods so they may be called through a SOAP client.

When deployed as a Servlet, Corticon Server automatically publishes an *`Administration Console`* on port `8082`, which among other things, exposes as set of WSDLs. See the next chapter for more details.

7

Relational database concepts in the Enterprise Data Connector (EDC)

Corticon's Enterprise Data Connector integrates its Decision Services with implementations of the relational database model.
For details, see the following topics:

- Identity strategies
- Advantages of using Identity Strategy rather than Sequence Strategy
- Key assignments
- Conditional entities
- Support for catalogs and schemas
- Support for database views
- Fully-qualified table names
- Dependent tables
- Inferred property values (“Best Match”)
- Join expressions
- Java Data Objects

Identity strategies

Because EDC allows Studio and Server to dynamically query an external database during Rulesheet/Decision Service execution, the Vocabulary must contain the necessary key and identity information to allow Studio and Server to access the specific data required. There are two identity types which may be selected for each Vocabulary entity: application and datastore.

Application Identity

With application identity, the field(s) of a given table's primary key are present as attributes of the Vocabulary entity. As a result, application identity normally means that the table's primary key field(s) have some business meaning themselves; otherwise they wouldn't be part of the Vocabulary. The `Cargo` sample (described in the *Basic Rule Modeling* and *Using EDC* tutorials) illustrate entities using application identities. In the case of entity `Aircraft`, the unique identifier (primary key) is `tailNumber`. In the database metadata, `tailNumber` is the designated primary key field. The presence in the Vocabulary of a matching attribute named `tailNumber` informs the auto-mapper that this particular entity must be application identity.

Datastore Identity

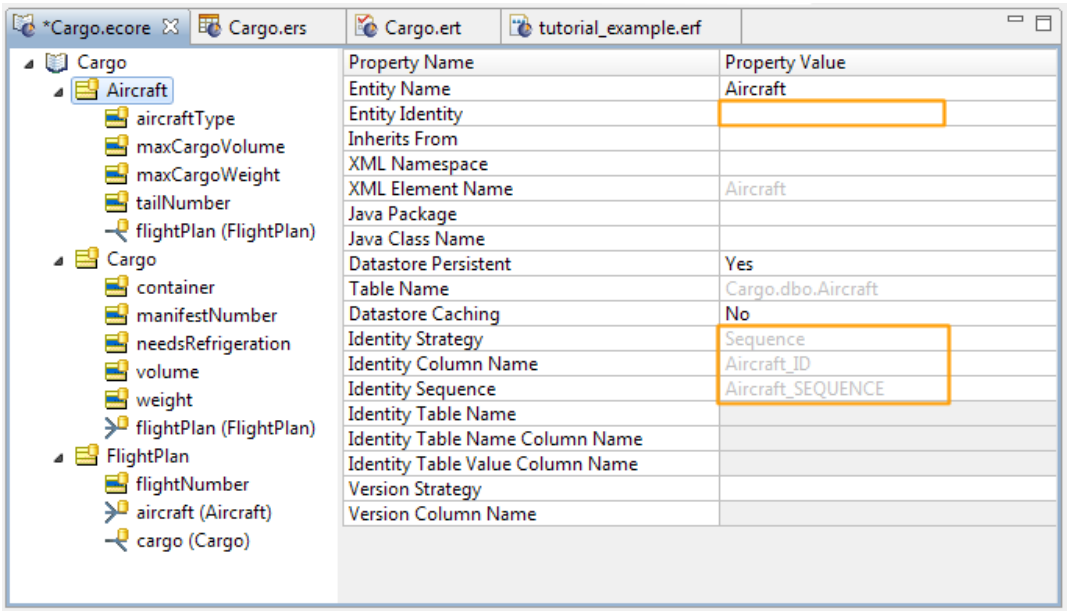
A Vocabulary entity uses datastore identity when it does not have an attribute that matches the database table's primary key field(s) . The table's primary key is effectively a *surrogate key* which really has no business meaning. If the designated primary key fields in the imported database metadata are not present as attributes in the Vocabulary entity, then the Vocabulary Editor will assume datastore identity and insert the table's primary key field(s) in the **datastore-identity:column** property.

We have modified our `Aircraft` table slightly to change the primary key. Previously, we assumed that `tailNumber` was the unique identifier for each `Aircraft` record – in other words, every aircraft must have a tail number and no two can have the same one. Let's assume now that this is no longer the case – perhaps `tailNumber` is optional (perhaps aircraft based in some countries don't require one?) or we somehow acquired two aircraft with the same `tailNumber`. So instead of `tailNumber`, we adopt a surrogate key for this table named `Aircraft_ID` that will always be non-null and unique. And since this field has no real business meaning (and we never expect to write rules with it), it isn't included in the Vocabulary.

Note: We can get to this state by clearing the database metadata, and then -- in the database - clearing (or deleting/recreating) the database. When we create the database schema again, the entity identities are all defaulted to datastore identities.

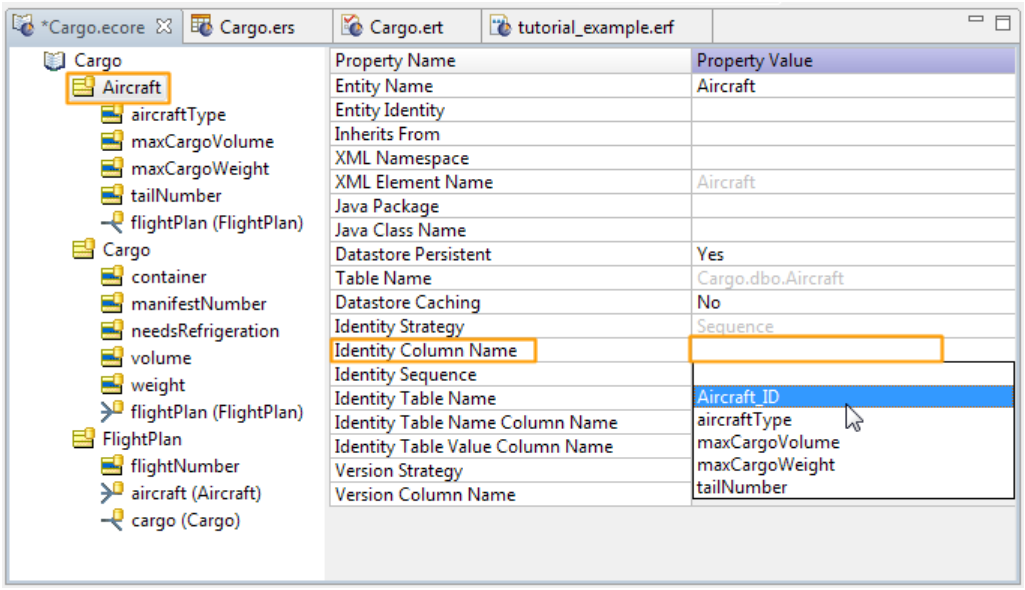
When the auto-mapper updated the schema, the Entity Identity was set to a `NULL`, and set the primary key field(s) in **Identity Column Name** as `Aircraft_ID`, as shown:

Figure 53: Automatic Mapping of Datastore Identity Column



If the auto-mapper does not detect the correct primary key in the metadata, we may need to manually select the field from the drop-down list, as shown:

Figure 54: Manual Mapping of Datastore Identity Column



By choosing datastore identity we are delegating the process of identity generation to the JDO implementation. This does not mean that we cannot control how it does this. The Vocabulary Editor offers the following ways to generate the identities:

- **Native** - Lets Hibernate choose the appropriate method for the underlying database. This usually means a Sequence in the RDBMS. Depending on the RDBMS you use, a sequence may require the addition of a sequence object or generator in the database.
- **Table** - Uses a table in the datastore with one row per table, storing the latest max id.

- **Identity** - Uses *identity* (Requires *identity* support in the underlying database.)
- **Sequence** - Uses *sequence* (Requires *sequence* support in the underlying database.)
- **UUID** - A UUID-style hexadecimal identity.

All of these strategies are database-neutral except for *sequence*. It is generally recommended that identity strategy be adopted for Vocabularies that are used to generate the database. When mapping to an existing database either *identity* or *sequence* strategies are typically used, depending on the database design.

Note:

These generators can be used for both datastore and application identities. The datastore identity is always using a strategy; if not explicitly set by the user, a default strategy is used. The application identity does not have a default strategy.

All strategies are using the integer data type with the exception of UUID which is using a string data type. If the type of the application identity attribute type does not fit the selected value strategy (for application identity), you get an alert.

Advantages of using Identity Strategy rather than Sequence Strategy

Consider the following points when deciding whether to use *identity* strategy or *sequence* strategy:

- When using the **Create/Update Database Schema** function in the Vocabulary, the sequences are generated automatically and tied to the **table id** fields on the database side. On the other hand, when using *sequence* strategy, the sequences are not generated during the **Create/Update Database Schema** process. If Corticon, at runtime, attempts to access a sequence and finds it missing, it will try to create it on the fly. But such a dynamic creation of sequences is tricky and does not always work properly.
- Using *identity* strategy should result in better performance when inserting a large number of records into the database. This is simply because the database I/O is cut in half since there is no need to retrieve the next unique id from the database prior to adding a new record.
- Using *sequence* strategy tends to not be compatible with read-only database access which may result in runtime exceptions.
- Using *identity* strategy makes a Vocabulary more portable across databases since not all databases support sequences.

Hibernate supports Sequence strategy for all databases; in a case where the database does not support it -- such as SQL Server -- Hibernate emulates it. However, in a case where the database does not support Identity strategy -- such as Oracle -- there is no emulation. This makes Sequence more portable.

Key assignments

Key designations occur automatically once an entity identity has been defined in the Vocabulary Editor.

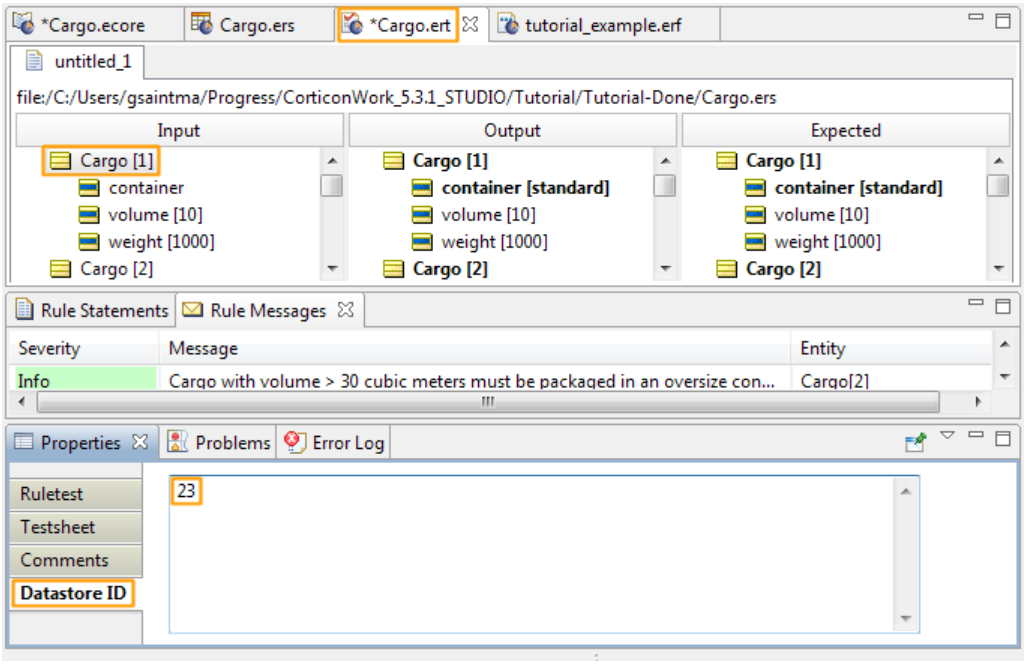
Primary Key

If the chosen (or auto-mapped) entity identity appears in the Vocabulary as an attribute (see Application identity), then that attribute receives an asterisk character to the right of its node in the Vocabulary's TreeView. Attributes with asterisks are part of the entity's primary key as shown in Automatic Mapping of Vocabulary Entity.

If the chosen (or auto-mapped) entity identity does **not** appear in the Vocabulary as an attribute see Datastore identity), then no attribute receives an asterisk character. None of the attributes in the Vocabulary are part of the entity's primary key, as shown in Automatic Mapping of Datastore Identity Column. This causes complications when testing and invoking Decision Services with connected databases. If no primary key is visible in the Vocabulary, then how do we indicate in an unambiguous way the specific records(s) to be used by the Decision Service?

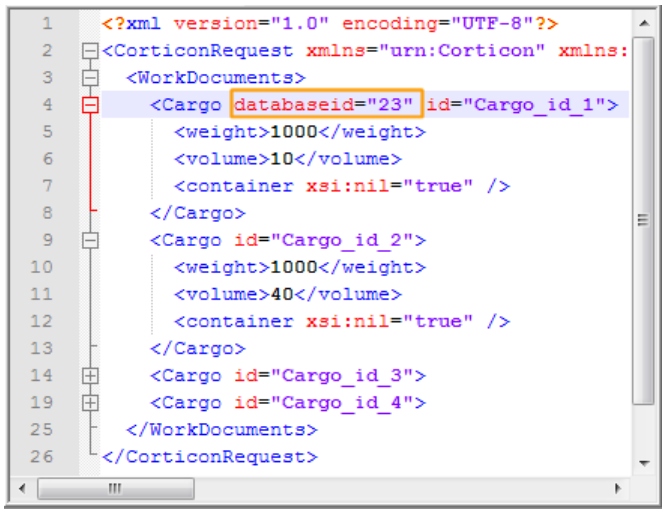
In the Studio Test, an entity using Datastore identity has its key set in the entity's **Properties** window. The following figure shows that the Ruletest was chosen. Right-clicking on first Cargo entity, and choosing **Properties** on the menu opened the **Properties** tab where the **Datastore ID** side tab was selected. The value `23` was entered for the test:

Figure 55: Setting the Identity for Entities Using Datastore Identity



When we export the ruletest to XML (**Ruletest > Testsheet > Data > Output > Export Response XML**) illustrates how this Database ID appears in the XML message. In the following figure, we see how the **Database ID** value is included in the XML as an attribute (an XML attribute, not a Vocabulary attribute). Your XML toolset and client may need to insert this data into a `CorticonRequest` message.

Figure 56: Datastore Identity inside the XML Request



Foreign Key

Foreign key relationships between database tables are represented in the Vocabulary via association mappings. As we see in Manual Mapping of Vocabulary Association , the association mappings are entered (or auto-mapped) in the **Join Expression** field.

Composite Key

Multiple keys may be selected (if not auto-mapped) by choosing the **Select All** option, or by holding the **Control** key while clicking on all the items you want on the **Entity Identity** drop-down. If multiple selections are made, then all Vocabulary attributes will have asterisk characters to indicate that they are part of the primary key.

Conditional entities

Although all database properties will unconditionally be displayed, their applicability and enablement is often dependent upon the values of other properties.

Universally, EDC properties are applicable only for entities whose Datastore Persistent flags are set to Yes. For entities that are not datastore-persistent, all EDC properties for that entity, including EDC properties belonging to the entity's attributes and associations, will be disabled.

For datastore-persistent entities, fields that are applicable will be enabled and editable, while fields that are not applicable will be disabled and will have a light-gray background. The applicability of fields will change dynamically based on the values of other fields.

Generally, fields which are not applicable in a given context will be disabled; however, any values that were previously entered into those fields will be preserved notwithstanding their lack of applicability, even if the field itself is disabled. Specific rules governing applicability are detailed in Entity Properties, Attribute Properties and Association Properties below.

Support for catalogs and schemas

Catalogs and schemas refer to the organization of data within relational databases. Data is contained in *tables*, tables are grouped into *schemas*, and then schemas are grouped into *catalogs*. The concepts of *schemas* and *catalogs* are defined in the SQL 92 standard yet are not implemented in all RDBMS brands, and, even then, not consistent in their meaning.

For example, in SQL Server, tables are grouped by owner and catalogs are called *databases*. In that case, a list of database names is filtered by a *Catalog filter*, and a list of table owners is filtered by a *Schema filter*. The owner of all tables is typically the database administrator, so if you do not know the actual owner name, select '**dbo**' (under SQL Server or Sybase), or the actual name of the database administrator.

Note: The term *schema*, as used in Corticon's **Import Database Metadata** feature, does not refer to the 'schema objects' that the mapping tool manipulates.

Support for database views

Many RDBMS brands support *views*, a virtual table that is essentially a stored query. Your database administrator might have set up views to:

- Combine (JOIN) columns from multiple tables into a single virtual table that can be queried
- Partition a large table into multiple virtual tables
- Aggregate and perform calculations on raw data
- Simplify data enrichment

It is common practice to constrain staff users to accessing *only* views in their database connection credentials.

Corticon's Enterprise Data Connector supports mapping a Vocabulary to an RDBMS view.

Using Associations

When Corticon Entities are mapped to View tables that were created without any `WHERE` clause in the Select statement (in other words, Corticon filters are NOT applied), Associations (in a View table) are not required as the Entities mapped to the View tables with no Join Expressions in the Vocabulary returns the expected results that include the Association.

Note: When Entities are mapped to View tables that were created *with* a `WHERE` clause in the Select statement (in other words, Corticon filters *are* applied), results are incorrect: Associations are required even when there is a View table for the Join Expressions. Attempts to map the View tables to the Entities in the Vocabulary will generate validation warnings for lost Join Expressions. A Join Expression currently cannot be mapped to its related View tables.

Fully-qualified table names

Whenever table names appear in properties, Corticon uses fully-qualified names; thus, a table name may consist of up to three nodes separated by periods. The JDBC specification allows for up to three levels of qualification for a table name:

- Catalog Name
- Schema Name
- Table Name

For databases that support all three levels of qualification, table names take the form:

```
<catalog>.<schema>.<table>
```

Microsoft SQL Server uses all three levels of qualification. For example, `Accounting.dbo.Customer`

Others, such as Oracle, do not use Catalog Name, and therefore use only schema and table. For example, `corticon.Customer`

Corticon can infer which levels of qualification are applicable by checking for null values in database metadata. For example, for databases that do not support Catalog Name, that field will be null for all tables.

Dependent tables

Sometimes the existence of a record in one table is dependent upon the existence of another record in a related table. For example, a `Person` table may be related to a `Car` table (one-to-many). A car may exist in the `Car` table independent of any entry in the `Person` table. In other words, a car record does not require a related person – a physical object exists on its own. Likewise, a person record could exist without an associated car (the person might not own a car). These two tables are independent, even though a relationship/association exists between them.

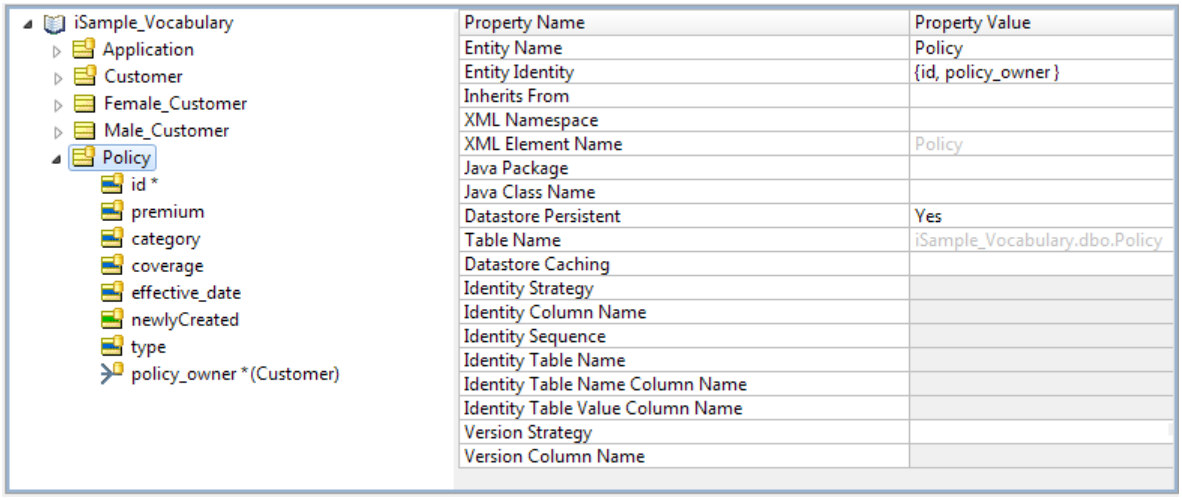
Some tables are not independent. Take `Customer` and `Policy` tables – if each policy record must have a person to whom the policy is "attached," we say the `Policy` table is dependent upon the `Customer` table. A person may or may not have a policy, but each policy must have a person.

Dependency normally comes into play when records are being removed from a table. In the first example, removing a person record has no affect on the associated car record. Although the person may no longer function as the car's owner, the car itself continues to exist. A car doesn't automatically vanish just because a person dies. On the other hand, removing a person *should* remove all associated policies. A person who switches insurance companies (and is deleted from its database) can expect his previous company to cancel and delete his old policies, too.

A Dependent table normally contains as part of its primary key the foreign key of the independent table. Since a Corticon Vocabulary represents a foreign key relationship as a **Join Expression** in the association mapping (see Manual Mapping of Vocabulary Association), a dependent entity will have a composite key with the association name participating in the key.

As we can see in the following figure, the composite key contains both `id`, which is the application identity for the `Policy` entity and `policy_owner`, which is the association between `Customer` and `Policy` entities. This indicates that `Policy` is a dependent table, and that removing a `Customer` record will also remove all associated policy records.

Figure 57: Primary Key of a Dependent Table Includes the Role Name



Inferred property values (“Best Match”)

Corticon attempts to infer the best possible matches for database table names, column names and related information such as the association join expressions.

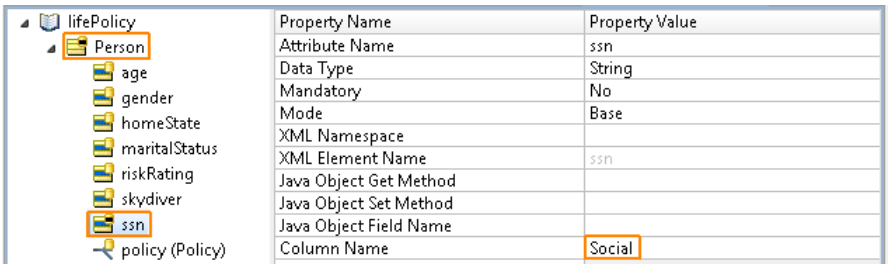
The ability to match entity names with table names is a key capability, because many of the other matching strategies will indirectly rely on this capability. Generally speaking, the system will locate the first table in database metadata that matches the entity name (ignoring case). For the purpose of this matching logic, catalog, schema and domains will be ignored. Similarly, the system will try to find the best matching columns for attributes, and the best matching join expressions for associations.

When a value is inferred in this manner, that value will not be stored in the model; rather, the system will dynamically infer the derived value whenever the Vocabulary Properties table is refreshed.

The system will display inferred properties in light gray font to prevent them from being confused with explicitly-specified values.

The user will always have the ability to override the inferred value by choosing an explicit value from a drop-down list, or by entering value manually. In such case, the explicitly-specified value will be displayed in black font, and the database decoration in the tree view has a black bar at its center, as illustrated for an entity:

. The specified value will be stored in the model. Such explicitly-specified values will take precedence over the inferred values. The following image illustrates how an attribute that is overidden is marked, as well as its entity:



If the user clears an explicitly-specified value by selecting the blank row in a drop-down list or by clearing the cell text, the system will once again display the inferred value in light gray.

Vocabulary facades `IEntity`, `IAttribute` and `IAssociationEnd` will minimize the distinction between inferred and explicitly-entered values. From the viewpoint of how Corticon creates objects it will use in rule execution, the distinction between the inferred and explicitly-specified values is immaterial. Vocabulary facade getters will always return the effective value, either inferred or explicit.

This strategy optimizes utility while minimizing user input. The user should favor inferred values whenever possible, because these values will automatically be updated as database metadata evolves. Conversely, explicitly-entered values will require ongoing attention as the schema is updated.

Table 3: Corticon inference rules

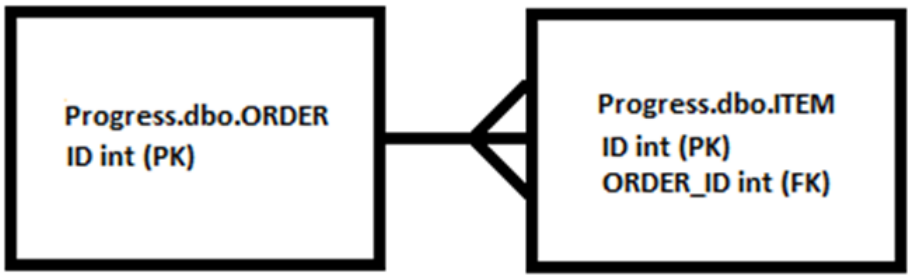
Vocabulary Property	Automatic Inference Rules
`Table Name`	Derived from table metadata. The first table located in database metadata that matches the entity name (ignoring case) will be chosen. This matching process will ignore catalog, schema and domains. The inferred value will be displayed as a fully-qualified name including catalog and schema, if applicable.
`Column Name`	Derived from first column in database metadata that matches the attribute name (ignoring case). For this purpose, the Table Name (whether explicitly-specified or inferred) will be used.
`Join Expression`	Complex derivation algorithm involving table data, column data, primary key and foreign key definitions. The algorithm must find the best matching join expression, which defines the relationships between database columns, typically along the lines of foreign keys.

Join expressions

Each association in a Corticon Vocabulary will have a join expression that is used to establish the relationships between matching columns in the database. The syntax is similar to the SQL `WHERE` clause and can best be illustrated by examples.

Examples of Join Expressions

Consider a bidirectional one-to-many relationship between tables:



`Progress.dbo.ORDER` and `Progress.dbo.ITEM` both have primary key ID (integer) and `Progress.dbo.ITEM.ORDER_ID` is a foreign key that "points" to primary key `Progress.dbo.ORDER.ID`. In such case the join expressions would be as follows:

Vocabulary Association	Join Expression
`Order.item`	`Progress.dbo.ORDER.ID = Progress.dbo.ITEM.ORDER_ID`
`Item.order`	`Progress.dbo.ITEM.ORDER_ID = Progress.dbo.ORDER.ID`

Note that in a bidirectional association, the two join expressions are mirror images of one another.

Unlike ANSI SQL, the order of operands in the join expression is significant.

For a multi-column primary key, all key columns must be specified in the join expression; in such case, the join expression becomes a set.

Again, consider a one-to-many, bidirectional association between `Progress.dbo.ORDER` and `Progress.dbo.ITEM`, but assume that both `Progress.dbo.ORDER` and `Progress.dbo.ITEM` have multi-column primary keys (`ID1, ID2`), and that `Progress.dbo.ITEM` also has multi-column foreign key (`ITEM.ORDER_ID1, ITEM.ORDER_ID2`). In such case, the join expressions would be as follows:

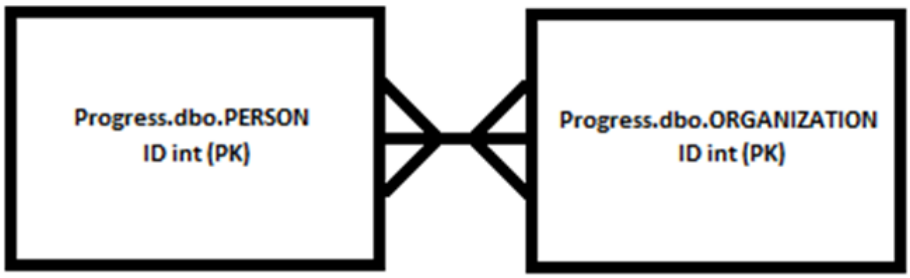
Vocabulary Association	Join Expression
`Order.item`	`{ Progress.dbo.ORDER.ID1 = Progress.dbo.ITEM.ORDER_ID1, Progress.dbo.ORDER.ID2 = Progress.dbo.ITEM.ORDER_ID2 }`
`Item.order`	`{ Progress.dbo.ITEM.ORDER_ID1 = Progress.dbo.ORDER.ID1, Progress.dbo.ITEM.ORDER_ID2 = Progress.dbo.ORDER.ID2 }`

Note the braces surrounding the comma-separated relational expressions, denoting that in this case, the join expressions are sets.

Finally, consider a bidirectional many-to-many association between two tables:

Such an association will involve a join table, an artificial table not represented in the Vocabulary, whose sole purpose is to associate records in `PERSON` and `ORGANIZATION`. Typically, this join table would have a self-documenting name such as `PERSON_ORGANIZATION` and would contain foreign keys that "point" to `PERSON` and `ORGANIZATION` (for example, `PERSON_ORGANIZATION.PERSON_ID, PERSON_ORGANIZATION.ASSOCIATION_ID`).

In such case, the join expressions would be as follows:



Vocabulary Association	Join Expression
Person.organization	`{ Progress.dbo.PERSON.ID = Progress.dbo.PERSON_1.PERSON_ID, Progress.dbo.PERSON_1.ORGANIZATION_ID = Progress.dbo.ORGANIZATION.ID }`
Organization.person	`{ Progress.dbo.ORGANIZATION.ID = Progress.dbo.PERSON_1.ORGANIZATION_ID, Progress.dbo.PERSON_1.PERSON_ID = Progress.dbo.PERSON.ID }`

Again, set notation is used, but instead of multi-column primary keys, the relational expressions describe the relationships between the two tables and the connecting join table.

Inferring Join Expressions

Because join expressions are cumbersome to enter, it is crucial that Corticon 5 have the best possible logic for automatically inferring them from metadata. For one-to-many associations, the join expression can frequently be inferred from primary and foreign key metadata, assuming that the entities can be successfully mapped to particular tables, and the foreign key relationships between those tables are properly declared. Exceptions to this rule include:

- Unary one-to-one associations (that is, *self-joins*), where it is impossible to infer which “side” of the association corresponds to the primary or foreign key
- Unary many-to-many associations, where it is impossible to infer which of the join table foreign keys should be used for each “side” of the association
- Tables that have multiple foreign key relationships between them with different meanings for each.

Corticon recognizes when it is not possible to unambiguously infer the proper join expression, and allow the user to choose from a set (drop-down list) of choices.

Corticon infers the join expressions in all cardinalities.

Java Data Objects

Most applications require some sort of data persistence. Developers traditionally have built applications with a specific data store and source in mind, using data store-specific APIs. This approach becomes troublesome and resource-intensive when trying to support and certify an application on numerous persistent data stores. Corticon has chosen the Java Data Objects (JDO) standard to standardize the way in which external data is accessed by Studio and Server.

The JDO specification defines a set of Java APIs that exposes a consistent model to programmers interacting with disparate data sources. More information on JDO is available at `http://java.sun.com/products/jdo/`.

8

Implementing EDC

The functionality described in this chapter requires an installed instance of both Corticon Studio and Corticon Server where you have a license file for Server that enables EDC. Contact Progress Corticon technical support or your Progress Software representative for confirm that you have such a license.

Note: Documentation topics on EDC:

- The tutorial, *Using Enterprise Data Connector (EDC)*, provides a focused walkthrough of EDC setup and basic functionality.
- *Writing Rules to access external data* chapter in the *Rule Modeling Guide* extends the tutorial into scope, validation, collections, and filters.
- "Relational database concepts in the Enterprise Data Connector (EDC)" in this guide discusses identity strategies, key assignments, catalogs and schemas, database views, table names and dependencies, inferred values, and join expressions.
- This chapter, "Implementing EDC" discusses the mappings and validations in a Corticon connection to an RDBMS.
- "Deploying Corticon Ruleflows" in this guide describes the Deployment Console parameters for Deployment Descriptors and compiled Decision Services that use EDC.
- *Vocabularies: Populating a New Vocabulary: Adding nodes to the Vocabulary tree view* in the *Quick Reference Guide* extends its subtopics to detail all the available fields for Entities, Attributes, and Associations.

A complete example of connecting a Rulesheet to an external database, including settings configuration and invocation details, is described in the *Corticon Tutorial: Enterprise Data Connector (EDC)*. We recommend completing or reviewing that Tutorial before reading this chapter. This chapter extends the information covered by the Tutorial.

For details, see the following topics:

Managing User Access in EDC

Because EDC carries the potential for data loss or corruption due to unintended updates, we recommend the following precautions:

1. Use a test instance of a database whenever testing EDC-enabled Rulesheets from Studio. If unintended changes or deletions are made during rule execution, then only test database instances have been changed, not production databases.
2. Even if using test instances, you may still want to restrict the ability to read and update connected databases to those users who understand the possible impact. For other rule modelers without a solid understanding of databases, you may want to provide them with read-only access.
3. As you approach production, you might want to reserve the ability to **Create/Update Database Schema** to a small set of senior administrators as that action drops database tables.

About Working Memory

When a Reactor (an instance of a Decision Service) processes rules, it accesses the data resident in “working memory”. Working memory is populated by any of the following methods:

1. The payload of the Corticon Request. In the case of integration Option 1 or 2, this payload is expressed as an XML document. In the case of Option 3, this payload consists of a reference to Java business objects. Regardless of form, the data is inserted into working memory when

the client's request (invocation) is received. When running a Studio Test, the Studio itself is acting as the client, and it inserts the data from the Input Ruletest into working memory.

2. The results of rules. During rule processing, some rules may create new data, modify existing data, or even delete data. These updates are maintained in working memory until the Rulesheet completes execution.
3. An external relational database. If, during the course of rule execution, some data is required which is not already present in working memory, then the Reactor asks Corticon Server to query and retrieve it from an external database. For database access to occur, Corticon Server or Studio Test must be configured correctly and the Vocabulary must be mapped to the database schema.

Configuring Corticon Studio

Corticon Studio is ready for database access as installed – just set the preference to expose it in the editors.

EDC in Corticon Studio allows the working memory created during a Studio Test to be populated from all three possible sources listed above, including from queries to an external database. The Vocabulary maintains the database metadata, schema, and the connection definition. Ruletests by default have no database access mode, allowing the user to choose **Read Only** access or **Read/Update** access for the test.

See the EDC Tutorial for a detailed walkthrough of the effect of these options.

Configuring Corticon Server

Corticon Server requires that you have a license that enables EDC, and that you register it for the Deployment Console, Java Server Console, and the server, both as in-process and as a remote server.

When you create a Corticon Deployment Descriptor (`.cdd`) in either the Deployment Console or the Java Server Console, you are provided the option to choose the database access mode, allowing the user to choose **Read Only** access or **Read/Update** access for the test.

You also can specify the database access connection parameters that were generated from Studio.

See the EDC Tutorial for a detailed walkthrough of the effect of these options.

Connecting a Vocabulary to an external database

The process for connecting a Corticon Vocabulary to an external RDBMS is described in the *Corticon Tutorial: Using Enterprise Data Connector (EDC)*. While an active Studio license always enables EDC, your Corticon Server license must explicitly support EDC. With a supported RDBMS brand installed and running in a network-accessible location, you can connect to an established database instance. Consult the Progress Corticon support pages to review the currently supported brands for your platform and product version.

Database drivers

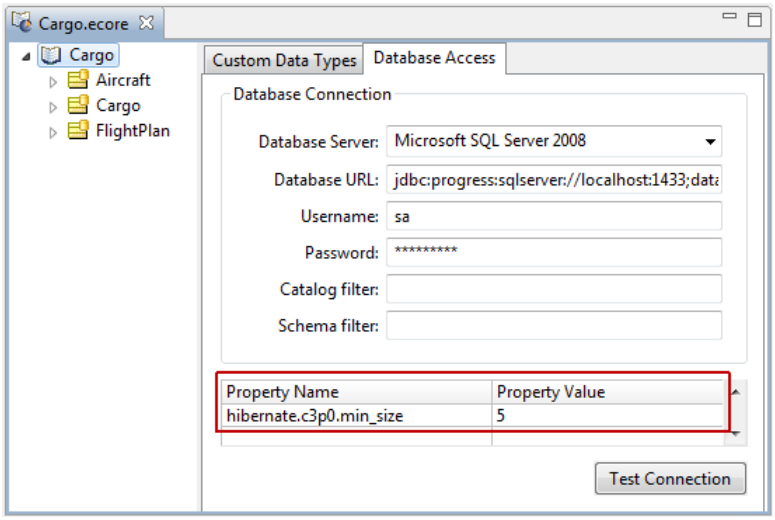
Corticon embeds Progress DataDirect JDBC Drivers that provide robust, configurable, high-availability functionality to RDBMS brands as well as full support for deployment with the object relational mapping (ORM) technology of Hibernate and Apache Tomcat.

The drivers are pre-configured and do not require performance tuning.

Connection Pooling

Corticon uses C3P0, an open source JDBC connection pooling product, for connection pooling to Hibernate.

The following properties might help tune connection pooling. You set override values in the **Property** table of the Vocabulary editor's **Database Access** tab, as illustrated:



Note: It is a good practice to test your connection before and after changing these properties.

The following properties let you tune connection pooling:

Table 4: Settable C3P0 properties and their default value

Property Name	Default value	Comment
`hibernate.c3p0.min_size`	1	Minimum number of Connections a pool will maintain at any given time.
`hibernate.c3p0.max_size`	100	Maximum number of Connections a pool will maintain at any given time.

Property Name	Default value	Comment
hibernate.c3p0.timeout	1800	Number of seconds a Connection will remain pooled but unused before being discarded. Zero sets idle connections to never expire.
hibernate.c3p0.max_statements	50	Size of C3P0's PreparedStatement cache. (Zero turns off statement caching. You might then need to declare required JAR and configuration files on the classpath, depending on the alternative connection pooling mechanism requirements.)

You can bypass the use of C3P0 for connection pooling by setting the Property name `hibernate.use.c3p0.connection_pool` to the value `false`.

For more information about C3P0 and its use for its use with Hibernate, see their *JDBC3 Connection and Statement Pooling* page at `http://www.mchange.com/projects/c3p0/index.html#appendix_d`.

Hibernate and EHCache override properties

Corticon has no recommendations for adjusting the properties in the Hibernate product or the EHCache product. Refer to their respective web locations for details. Then consult with Progress Corticon Support to note the behaviors you are attempting to adjust before taking action.

Mapping and validating database metadata

Performing the tasks in the following section requires Studio to be in **Integration and Deployment mode**. To set this or to confirm the setting, choose the Studio menu command **Window > Preferences**, then expand **Progress Corticon** and click on **Rule Modeling**. Choose the option **Integration and Deployment**. This is a good time to also check that Studio is pointing to your Corticon license file.

Using a Vocabulary to generate database metadata in an external RDBMS

The process of using an existing Studio Vocabulary as a template for generating corresponding metadata in an external RDBMS is detailed in the *Corticon Tutorial: Using Enterprise Data Connector*.

Importing database metadata into Studio from an external RDBMS

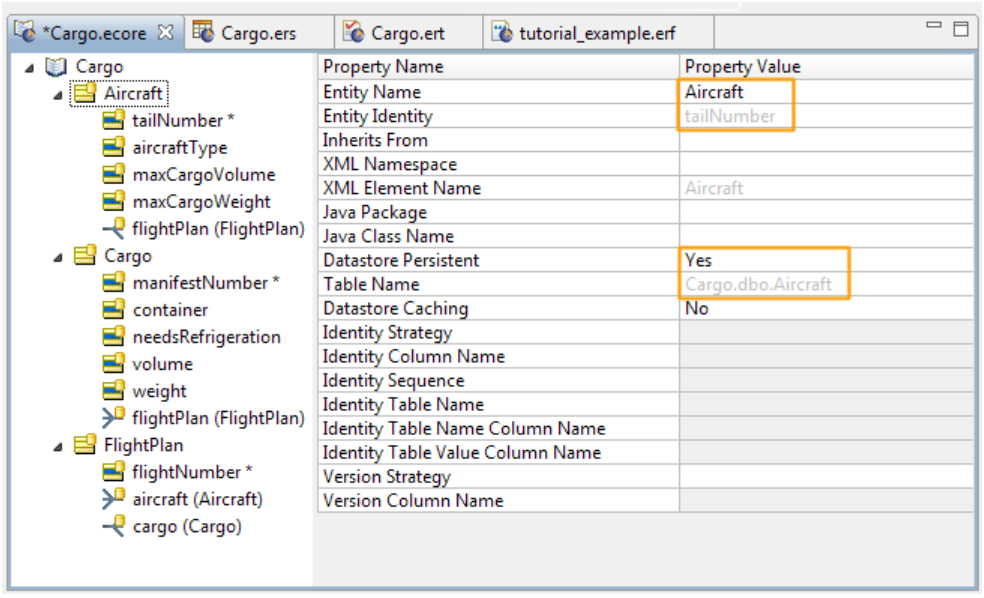
When an external database schema exists, we can import the metadata into Corticon Studio and create mappings between our Vocabulary and the metadata. Once the database connection is established as shown the *Corticon Tutorial: Using Enterprise Data Connector*, choosing the **Vocabulary>Database Access>Import Database Metadata** command from Studio's menubar imports the metadata into Studio.

When database metadata is imported into a Vocabulary, the Vocabulary Editor's automatic mapping feature attempts to find the best match for each piece of metadata. The matching process is case-insensitive. An entity will be auto-mapped to a table if the two names are spelled the same way, regardless of case.

Mapping database tables to Vocabulary Entities

Not all Vocabulary entities must be mapped to corresponding database tables - only those entities whose attribute values need to be persisted in the external database should be mapped. Those entities not mapped should have their `Datastore Persistent` property set to `No`. Mapped entities must have their `Datastore Persistent` property set to `Yes`, as shown circled in orange in the following figure:

Figure 58: Automatic Mapping of Vocabulary Entity

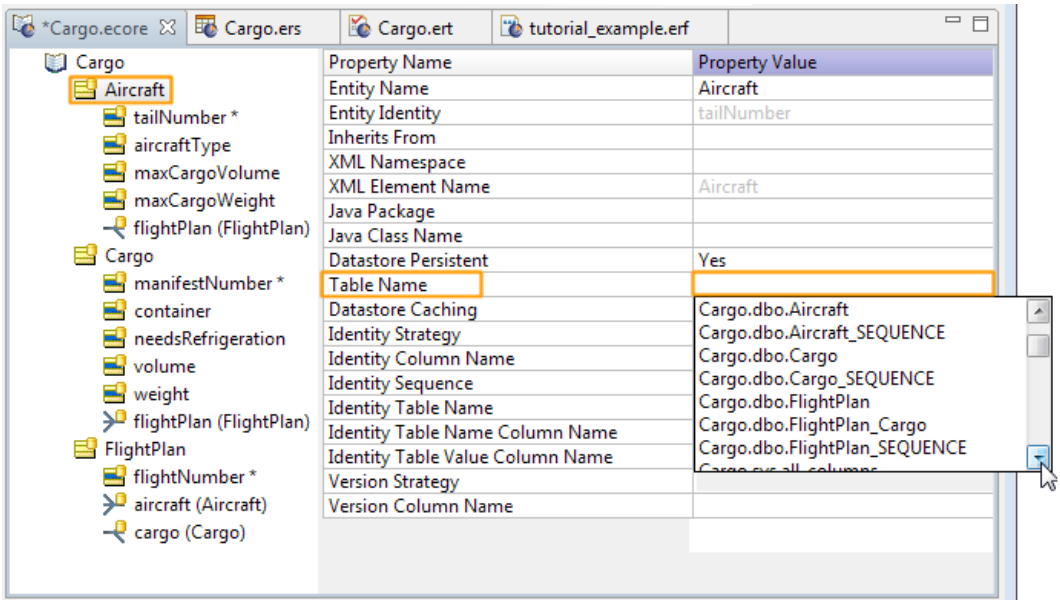


It is also possible for an external database to contain tables or fields not mapped to Vocabulary entities and attributes - these terms are simply excluded from the Vocabulary.

In the preceding, database metadata containing a table named `Aircraft` was imported. Because the table's name spelling matches the name of entity `Aircraft`, the **Table Name** field was mapped automatically. Automatic mappings are shown in light gray color, as highlighted above. Also, note that the primary key of table `Aircraft` is a column named `tailNumber`. The Vocabulary Editor detects that too, and determines that the property **Entity Identity** should be mapped to attribute `tailNumber`.

If the automatic mapping feature fails to detect a match for any reason (different spellings, for example), then you must make the mapping manually. In the **Table Name** field, use the drop-down to select the appropriate database table to map, as shown:

Figure 59: Manual Mapping of Vocabulary Entity



Enumerations updated from a database

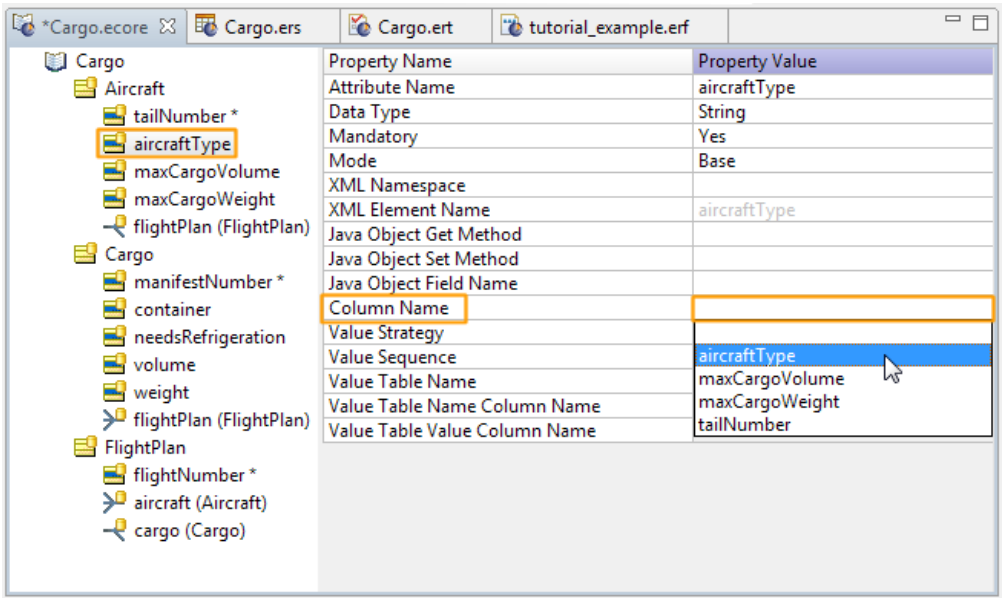
You can use Custom Data Types to retrieve unique name/value or just value lists from specified columns in a table, as described in the following topics:

- *"Enumerated values" in the Quick Reference Guide.*
- *"Enumerations retrieved from a database" in the Rule Modeling Guide*
- *"Importing an attribute's possible values from database tables" in the Using EDC Guide*

Mapping database fields (columns) to Vocabulary Attributes

Automatic mapping of attributes works the same as entities. If an automatic match is not made by the system, then select the appropriate field name from the drop-down in **field:column** property, as shown:

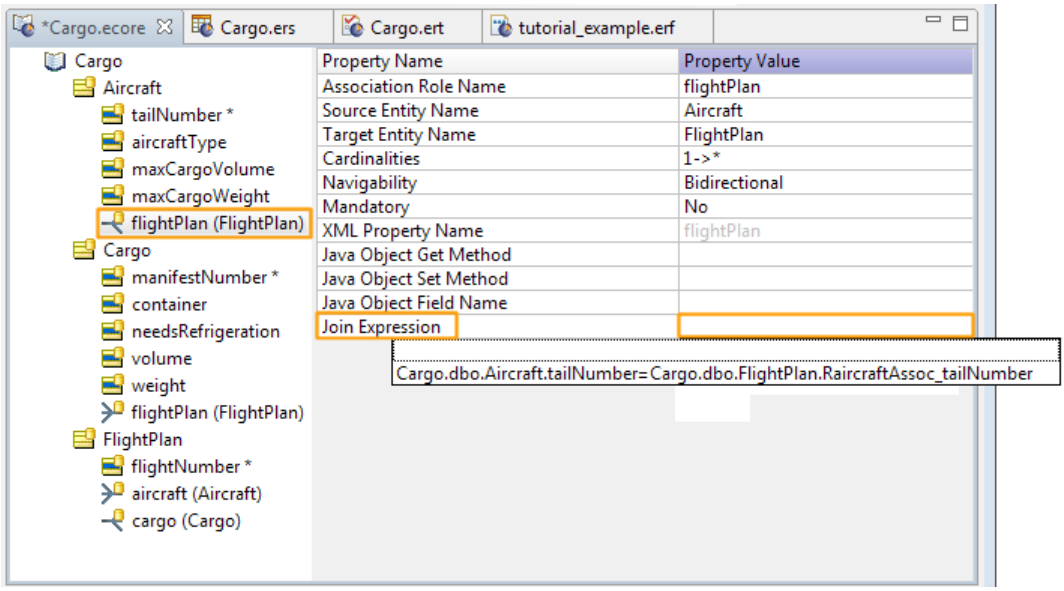
Figure 60: Manual Mapping of Vocabulary Attribute



Mapping database relationships to Vocabulary Associations

Automatic mapping of associations works the same as entities. If an automatic match is not made by the system, then select the appropriate field name from the drop-down in the **Join Expression** property, as shown:

Figure 61: Manual Mapping of Vocabulary Association

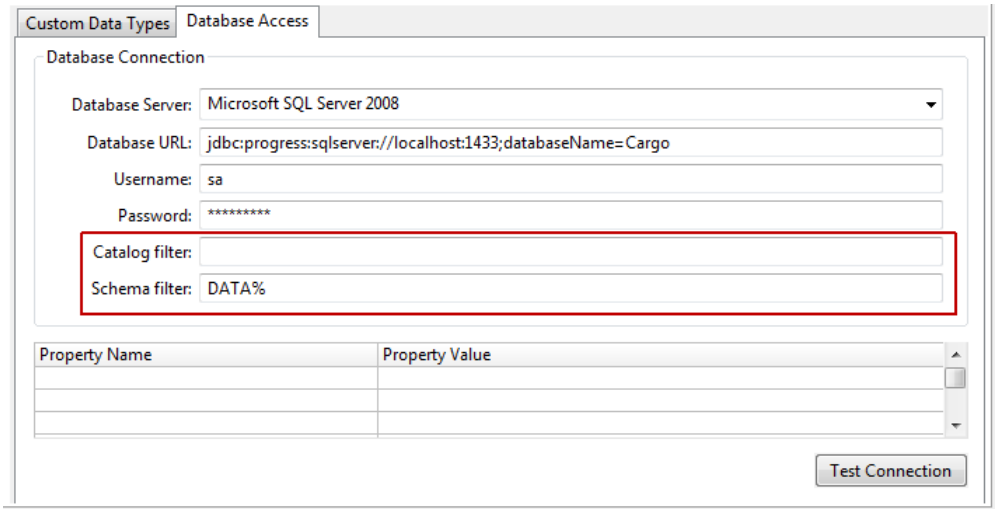


Filtering catalogs and schemas

Catalog and schema filters

Catalog and schema filters refine the metadata that is imported during an **Import Database Metadata** action (which also done in **Create/Update Database Schema**. This is crucial in production databases that might have hundreds or even thousands of schemas. Catalog filters and schema filters are defined on a Vocabulary's **Database Access** tab, as shown:

Figure 62: Metadata Import Filters



Note: These metadata import filters affect only the **Import Database Metadata** action. If you need to control the default schema and catalog used by Hibernate, enter Property Name and Property Value pairs in the lower portion of the Vocabulary's **Database Access** tab can be used, but they do not filter the imported metadata.

As the **Catalog filter** value does **not** support wildcards, distinguishing two metadata import filters enables the use of wildcards in the **Schema filter** value:

- Underscore (_) provides a pattern match for a single character.
- Percent sign (`%`) provides a pattern match for multiple characters (similar to the SQL `LIKE` clause.)

For example, you could restrict the filter to only schemas that start with `DATA` by specifying: `DATA%`, as illustrated above.

The ability to specify patterns is especially valuable when testing performance on RDBMS brands with EDC applications that use multiple schemas.

Creating a Database from a Vocabulary

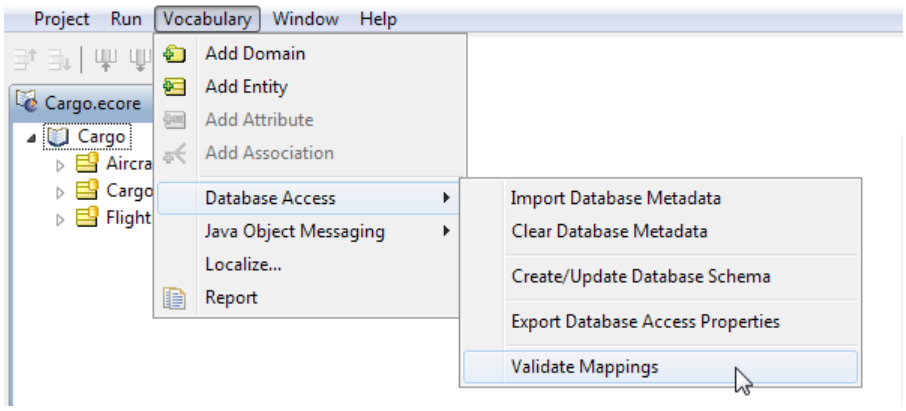
The **Create/Update Database Schema** feature will generate the structure of a database using the Corticon Vocabulary. When this technique is used, the generated tables end up in the database's default Catalog/Schema. However, if it is preferable to target a specific Catalog and/or Schema where the tables are to be generated, then the same connection properties referenced above can be used for this purpose.

Note: This feature is supported only on certain database brands. When you choose a Database Server, the feature will be enabled only it is supported for that brand.

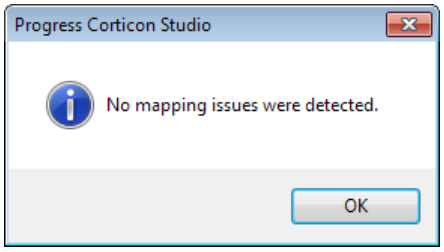
Validating database mappings

Once the Vocabulary has been mapped (either automatically or manually) to the imported database metadata, the mappings must be verified using the **Vocabulary>Database Access>Validate Database Mappings** option from the Studio menubar, as shown:

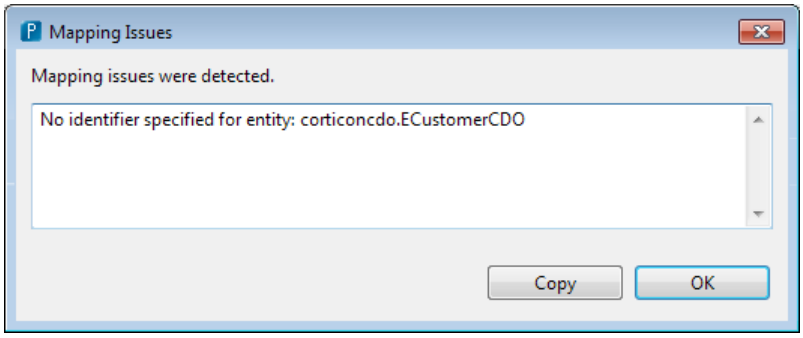
Figure 63: Validate Database Mappings Option



If all the mappings validate, then a confirmation window opens:



If anything in the mappings does not validate, then a list of problems is generated:



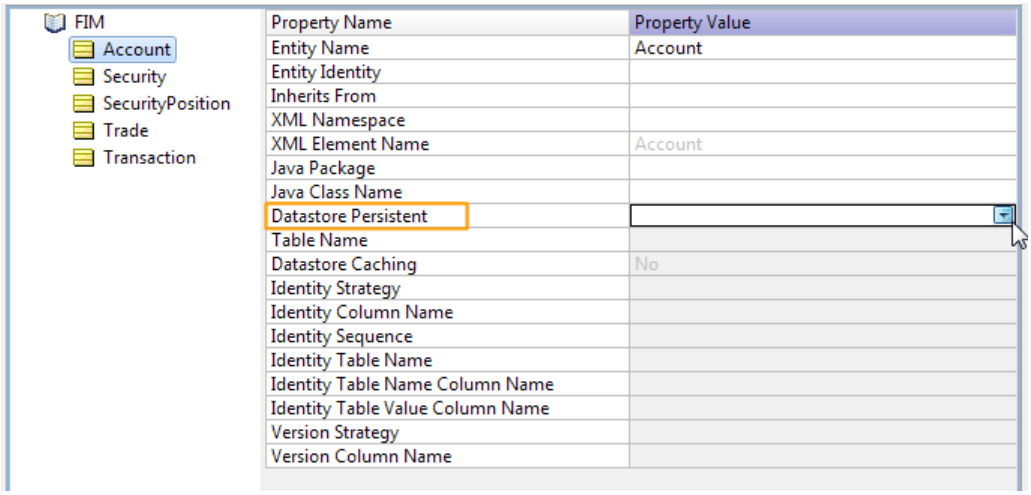
These problems must be corrected before the Ruleset can be deployed.

Note: For a more detailed discussion of validation, see the topic "Validation of database properties" in the *Rule Modeling Guide*

Creating and updating a database schema from a Vocabulary

A Vocabulary can act as the source schema for a new database. That technique “force-generates” a schema into a connected database as described in the loading of the `Cargo` Vocabulary into SQL Server in the EDC tutorial.

Before you can generate an Entity in a Vocabulary to a database you must click on its **Datastore Persistent** property, as shown:



Choose `Yes`, as shown: 

Repeat this action on every Entity in the Vocabulary that you want to map to a database table.

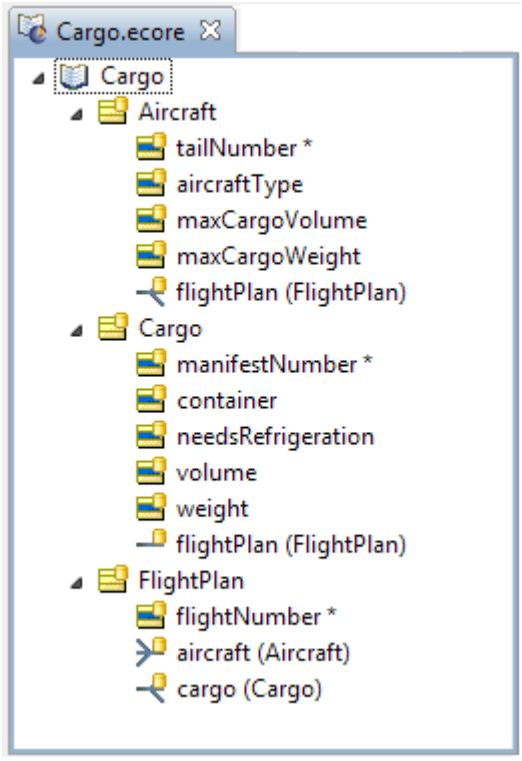
Note: Some Entities (such as `Male_Customer` and `Female_Customer` in the `Insurance` sample) are inherited from another entity and cannot be mapped to the database. For those Entities, set **Database Persistent** to `No`.

Data synchronization

EDC introduces a new dimension to rule execution. When EDC is not used, data management during Decision Service execution is relatively straight-forward: incoming data contained in the request payload is modified by rules and the resulting updated state for all objects is returned in the response.

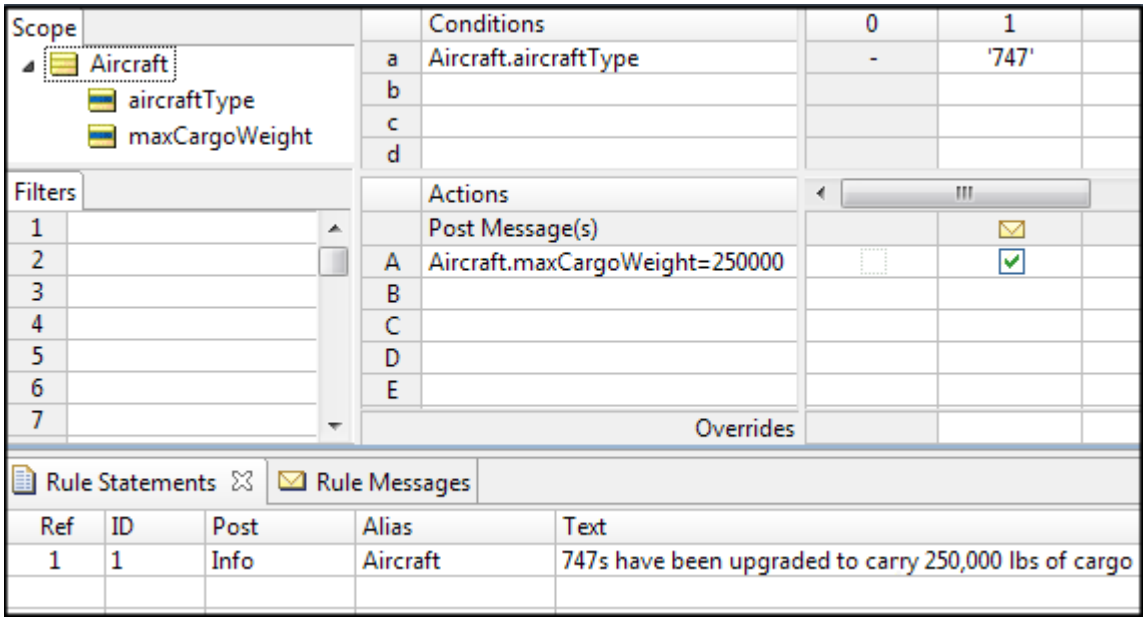
However, when EDC is used, data management becomes more complicated. How is data in the database synchronized with the data contained in the request payload and data produced or updated by Decision Service execution? Using several scenarios, this section describes the algorithms used by Corticon Server to perform this synchronization. All scenarios use the familiar `Cargo.ecore`, which was connected to a database in the *Corticon Tutorial: Enterprise Data Connector (EDC)*. If you have not completed that Tutorial, we highly recommend doing so before continuing, as this section builds on the concepts introduced there.

Figure 64: `Cargo.ecore` with Database Connection and Mappings



The sample Rulesheet we will use is defined as shown:

Figure 65: Sample Rulesheet for Synchronization Examples



Note: The RDBMS data in this section was established in the *EDC Tutorial* into a Microsoft SQL Server 2008 installation. The data was extended in the "Testing the Rulesheet with Database Access enabled" topic in the *Rule Modeling Guide* to add and populate the data that is described in this chapter's topics.

Read-Only database access

Read-Only Database Access

In **Read-Only** mode, data may be retrieved from the database in order to provide the inputs necessary to execute the rules. But the results of the rules won't be written back to the database – hence, read-only.

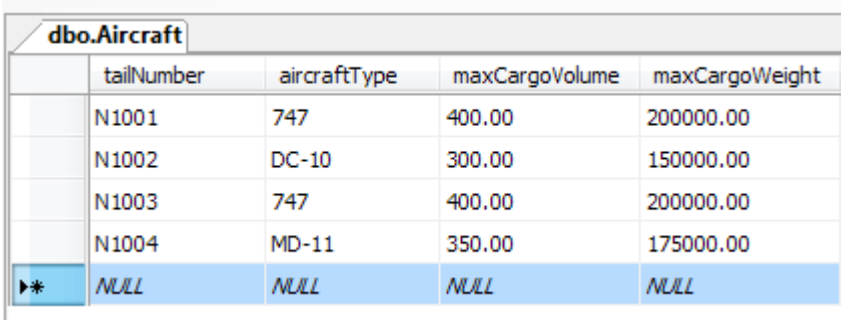
Read-Only mode is set for an Input Ruletest by selecting **Ruletest > Testsheet>Database Access>Read Only** from the Studio's menubar (the Ruletest must be the active Studio window).

Payload contains a New Record not in the Database: Alias not extended to database

This scenario assumes that the rule shown above does not make use of an alias extended to the database. See the *Rule Modeling Guide*'s chapter "Modeling Rules that Access External Data" for more information about this setting. A similar scenario using an extended-to-database alias follows this one.

First, let's look at a Studio Test with an Input Ruletest (simulating a request payload) containing a record not present in the database. The initial database table `dbo.Aircraft` is as shown:

Figure 66: Initial State of Database Table dbo_Aircraft

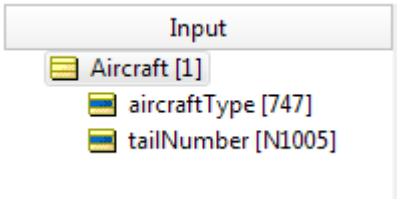


dbo.Aircraft

	tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
	N1001	747	400.00	200000.00
	N1002	DC-10	300.00	150000.00
	N1003	747	400.00	200000.00
	N1004	MD-11	350.00	175000.00
*	NULL	NULL	NULL	NULL

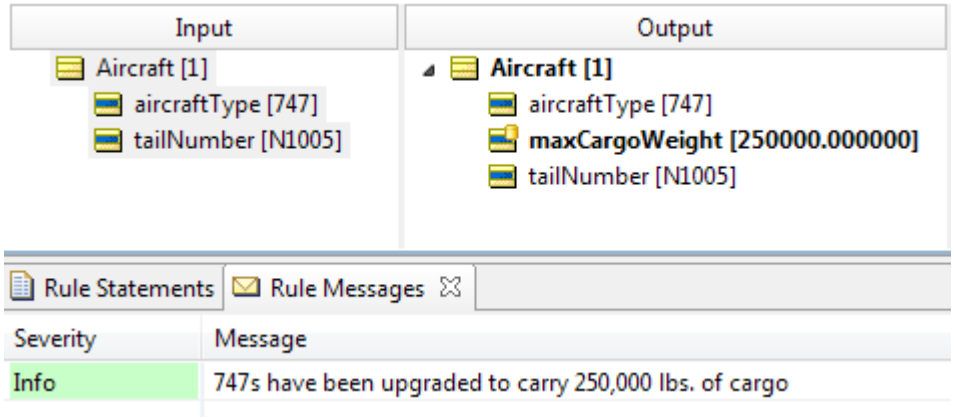
And the Studio Input Ruletest is as shown in the following figure.

Figure 67: Input Ruletest Testsheet with New Record, in Read-Only Mode



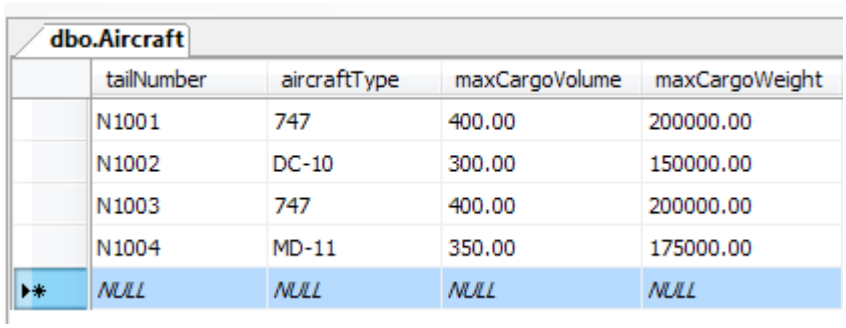
We know from our Vocabulary that `tailNumber` is the primary key for the `Aircraft` entity. We also know by examining the `Aircraft` table that this particular set of input data is not present in our database, which only contains aircraft records with `tailNumber` values `N1001` through `N1004`. So when we execute this Test, the Studio performs a query using the `tailNumber` as unique identifier. No such record is present in the table so all the data required by the rule must be present in the Input Ruletest. Fortunately, in this case, the required `aircraftType` data is present, and the rule fires, as shown:

Figure 68: Results Ruletest with New Record



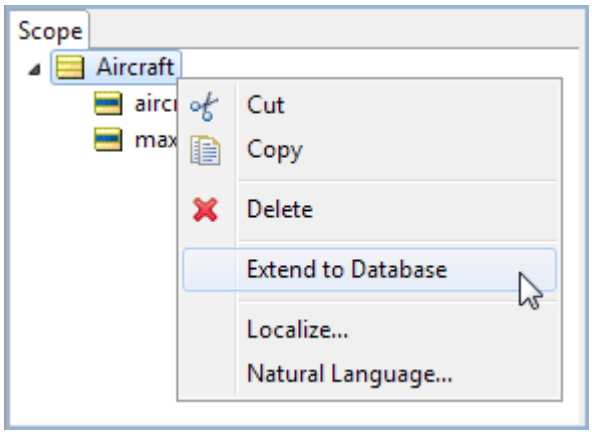
Again, since EDC is **Read-Only** for this test, no database updates are made and the end start of the `AIRCRAFT` table, as shown, is the same as the original state:

Figure 69: Final State of Database Table AIRCRAFT



Payload contains a New Record not in the Database: Alias extended to database

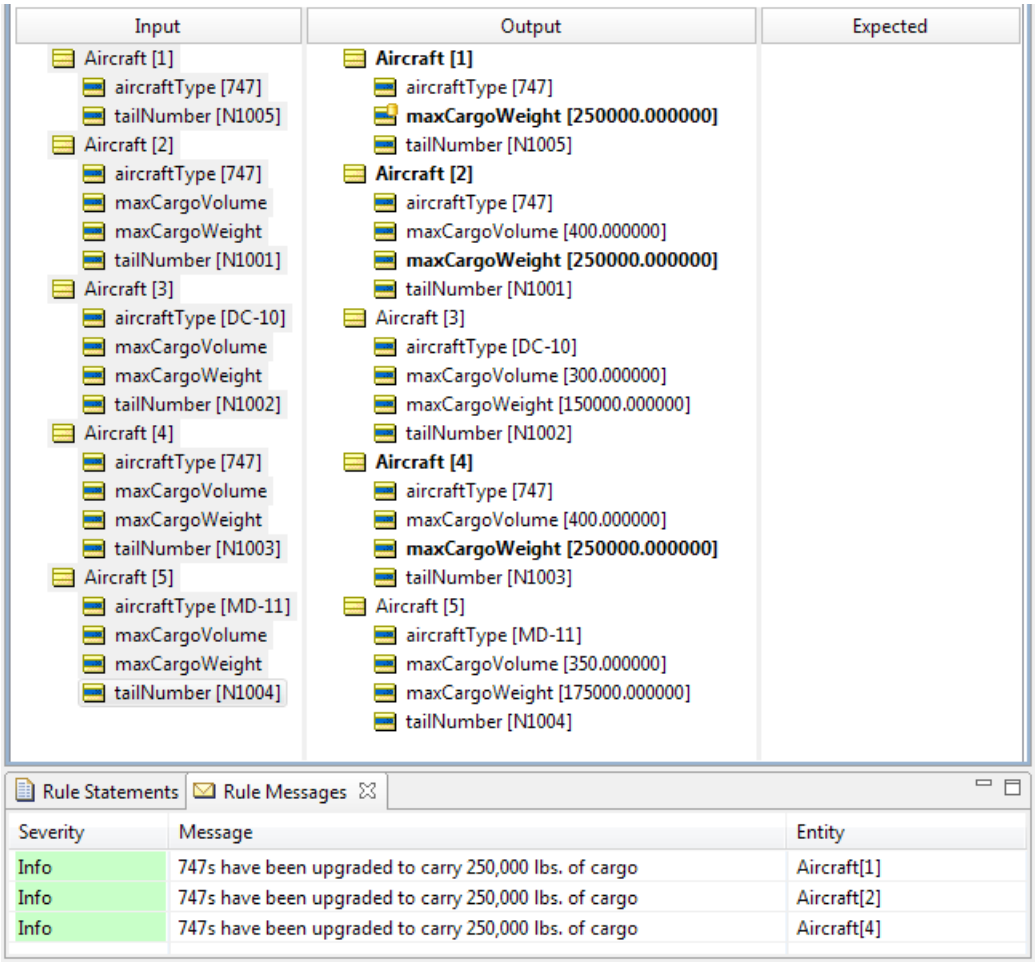
This scenario assumes the rule shown in Sample Rulesheet for Synchronization Examples makes use of an alias extended to the database. By placing the `Aircraft` Entity in the Scope of Rulesheet, we can right-click on `Aircraft` and then choose **Extend to Database** as shown:



See the *Rule Modeling Guide* chapter "Writing Rules to Access External Data" for more information about this setting. In that guide, you might want to learn about "Optimizing Aggregations that Extend to Database" which pushes these collection operations onto the database.

When our sample rule uses an alias extended to the database instead of the root-level entity shown in Sample Rulesheet for Synchronization Examples, different behavior is observed. When an Input Ruletest or request payload contains data not present in the database, as in test case `N1005` above, and the database access mode is **Read-Only**, then the rule engine dynamically re-synchronizes or "re-binds" with *only those records in the database table*. When this re-synchronization occurs, any data not present in the database table (like `N1005`) is excluded from working memory and not processed by the rules using that alias. The Results Ruletest is shown in the following figure. Notice that the Aircraft `N1005` was not processed by the rule even though, as a `747`, it satisfies the condition.

Figure 70: Results Ruletest Showing Re-Binding

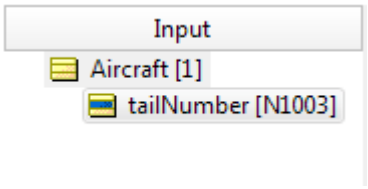


Payload Contains Existing Database Record

Now, let's change our input data so that it contains a record in the database. As we can see in the following figure, the value of `tailNumber` in the Input Ruletest has been changed to `N1003`. Also, the value of `aircraftType` has been deleted. By deleting the value of `aircraftType` from the Input Ruletest, rule execution is depending on successful data retrieval because the Input Ruletest no longer contains enough data for the rule to execute. Data retrieval is this rule's "last chance" – if no data is retrieved, then the rule simply won't fire.

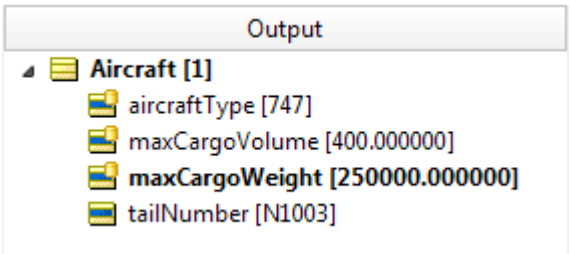
Fortunately, a record with this value exists in the database table, so when the Test is executed, a query to the database successfully retrieves the necessary data.

Figure 71: Ruletest Input with Existing Record



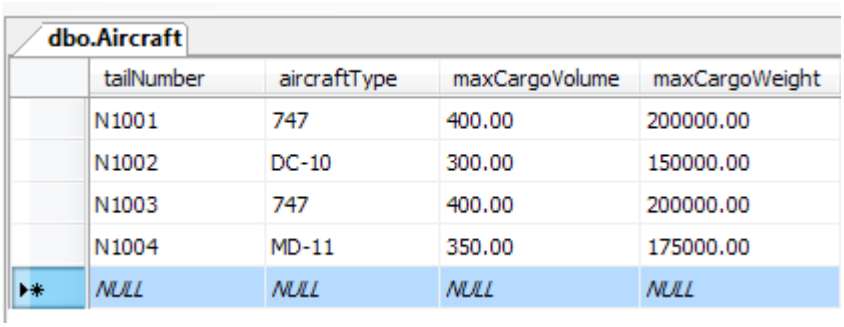
The Results Ruletest, as shown below, confirms that data retrieval was performed.

Figure 72: Ruletest Output with Existing Record



And, finding that the aircraft with `tailNumber=N1003` was in fact a `747`, the rule fired. But as before, no updates have been made to the database because this Test still uses Read-Only mode. The final database state is as shown:

Figure 73: Final State of Database Table AIRCRAFT



dbo.Aircraft

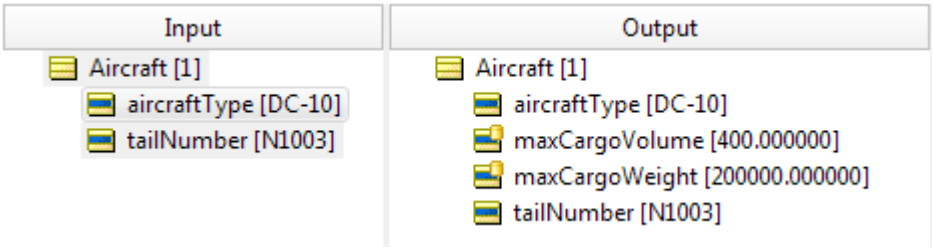
tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
N1001	747	400.00	200000.00
N1002	DC-10	300.00	150000.00
N1003	747	400.00	200000.00
N1004	MD-11	350.00	175000.00
NULL	*NULL*	*NULL*	*NULL*

Payload Contains Existing Database Record, but with Changes

What happens when, for a given record, the request payload and database record don't match? For example, look carefully at the Input Ruletest below. In the database, the record corresponding to tailNumber `N1003` has an `aircraftType` value of `747`. But the `aircraftType` attribute in the Input Ruletest has a value of `DC-10`. How is this mismatch handled?

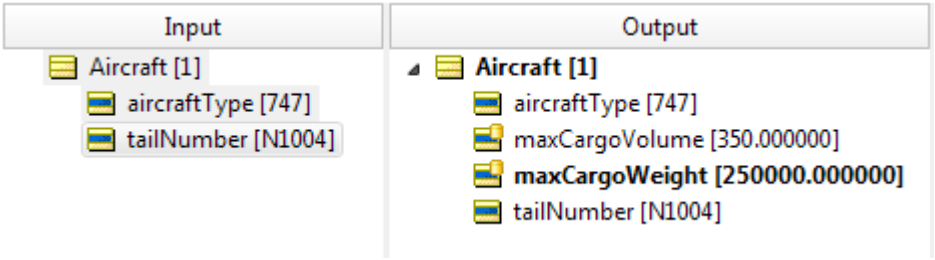
Studio still performs a query to the database because it has the necessary key information in the provided `tailNumber`. When the query returns with an `aircraftType` of `747`, the Synchronization algorithm decides that the data in the Input Ruletest has priority over the retrieved data – for the purposes of working memory (which is what the rules use during processing), the data in the Input Ruletest is treated as "more recent" than the data from the table. The state of `aircraftType` in working memory remains `DC-10`, and therefore the condition of the rule is not satisfied and the rule does not fire. Even though the database record defines the aircraft with `tailNumber` of `N1003` as a `747`, this is not good enough to fire the rule. The other piece of retrieved data, `maxCargoWeight`, is accepted into working memory and is inserted into attribute `maxCargoWeight` in the Results Ruletest upon completion of rule execution, as shown on the right side of the following figure:

Figure 74: Ruletest with Existing Record but Different Aircraft



Let's modify the scenario slightly. Look at the next Input Ruletest, as shown on the left side off the following image. It contains an `aircraftType` attribute value of `747`, but the `AIRCRAFTTYPE` value in the `AIRCRAFT` table of the database (for this value of `TAILNUMBER`) is `MD-11`. How is data synchronized in this case?

Figure 75: Ruletest with Existing Record and Same Aircraft



Once again, when a data mismatch is encountered, the data in the Input Ruletest (simulating the request payload) is given higher priority than the data retrieved from the database. Furthermore, the data in the Input Ruletest satisfies the rule, so it fires and causes `maxCargoWeight` to receive a value of `250000`, as shown on the right side of the figure above.

Effect of Rule Execution on the Database

In several of the examples above, the state of data post-rule execution differs from that in the database. In Results Ruletest with Existing Record and Results Ruletest with Existing Record, rule execution produced a `maxCargoWeight` of `250000`, yet the database values remained `200000`. The application architect and integrator must be aware of this and ensure that additional data synchronization is performed by another application layer, if necessary. When Corticon Studio and Server are configured for **Read-Only** data access, data contained in the response payload may not match the data in the mapped database.

Read/Update database access

To avoid the problem of post-rule execution data mismatch, EDC may be set to **Read/Update** mode. In this mode, Corticon Studio and Server can update the database so that data changes made by rules are persisted. This avoids the post-execution synchronization problem we encountered with **Read-Only** EDC mode, but must be used very carefully since *rules will be directly writing to the database*.

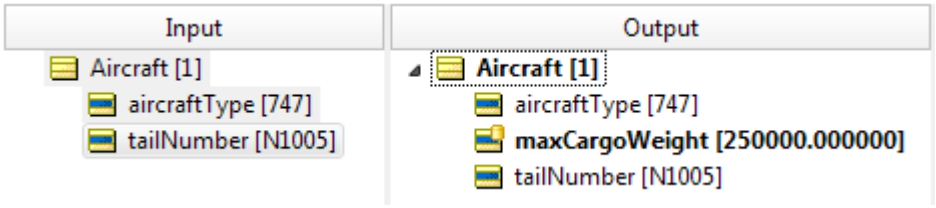
We'll use the same batch of examples as before to discuss the synchronization performed by Studio and Server when set to **Read/Update** mode for a Ruletest by selecting **Ruletest > Testsheet>Database Access>Read/Update** from the Studio's menubar (the Ruletest must be the active Studio window).

Payload contains a New Record not in the Database

Once again, the Studio Ruletest Input is shown in the following figure.

As before, no such record is present in the table so all the data required by the rule must be present in the Input section. Fortunately, in this case, the required `aircraftType` data is present, and the rule fires, as shown:

Figure 76: Ruletest with New Record



Since the EDC mode is **Read/Update**, a database update is made and the end state of the `Aircraft` table, shown below, is different from its original state.

Figure 77: Final State of Database Table Aircraft

dbo.Aircraft

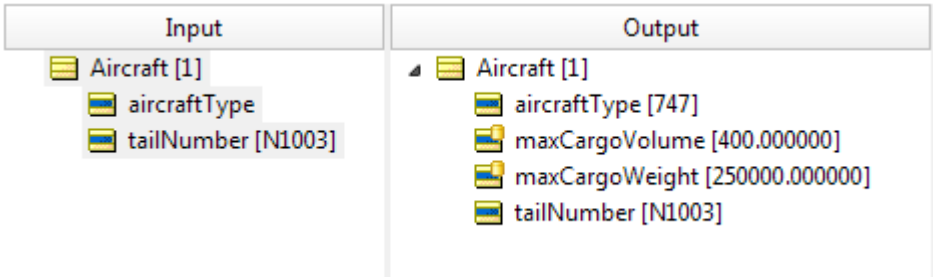
	tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
▶	N1001	747	400.00	250000.00
	N1002	DC-10	300.00	150000.00
	N1003	747	400.00	250000.00
	N1004	MD-11	350.00	175000.00
	N1005	747	*NULL*	250000.00
*	*NULL*	*NULL*	*NULL*	*NULL*

We can see that the database and the Ruletest Results (simulating the response payload) contain identical data for the record processed by the rule – no post-execution synchronization problems exist.

Payload Contains Existing Database Record

Now, let's revisit the Input Ruletest shown in Input Ruletest with Existing Record. Setting this Test to **Read/Update** mode, it appears as shown:

Figure 78: Ruletest with Existing Record



The Output section of the Ruletest confirms that data retrieval was performed. And, finding the retrieved aircraft was (and still is) a `747`, the rule fired.

Unlike the **Read-Only** example, the database has been updated with the new `maxCargoWeight` data. The final database state is as shown:

Figure 79: Final State of Database Table `Aircraft`

dbo.Aircraft

tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
N1001	747	400.00	250000.00
N1002	DC-10	300.00	150000.00
N1003	747	400.00	250000.00
N1004	MD-11	350.00	175000.00
N1005	747	*NULL*	250000.00
NULL	*NULL*	*NULL*	*NULL*

Payload Contains Existing Database Record, but with Changes

To better illustrate how the following examples affect the database when run in **Read/Update** mode, we will return the database's `Aircraft` table to its original state, as shown:

Figure 80: Original State of Database Table `Aircraft`

dbo.Aircraft

tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
N1001	747	400.00	200000.00
N1002	DC-10	300.00	150000.00
N1003	747	400.00	200000.00
N1004	MD-11	350.00	175000.00
NULL	*NULL*	*NULL*	*NULL*

When the following Ruletest is executed, we know from our experience with **Read-Only** mode that the rule will not fire. However, notice in Final State of Database Table `Aircraft` that the database record has been updated with the `aircraftType` value (`DC-10`) present in working memory when rule execution ended. And since the value of `aircraftType` in working memory came from the Input Ruletest (having priority over the original database field), that's what's written back to the database when execution is complete. The final state of the data in the database matches that in the Results Ruletest upon completion of rule execution, as shown in the Results Ruletest:

Figure 81: Ruletest with Existing Record

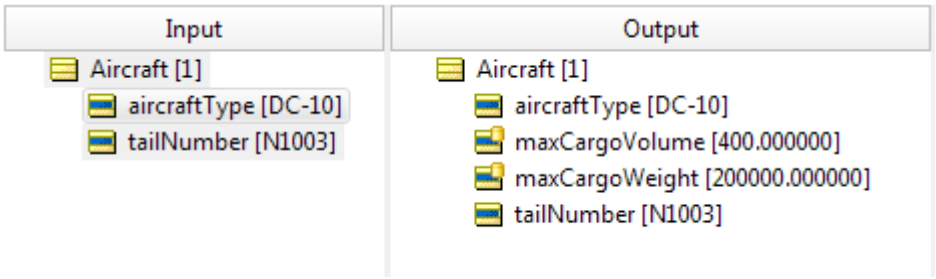


Figure 82: Final State of Database Table `Aircraft`

dbo.Aircraft

	tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
	N1001	747	400.00	250000.00
	N1002	DC-10	300.00	150000.00
▶	N1003	DC-10	400.00	200000.00
	N1004	MD-11	350.00	175000.00
*	*NULL*	*NULL*	*NULL*	*NULL*

As before, let's modify the scenario slightly. The Ruletest Input shown in the next figure now contains an aircraft record that has an `aircraftType` value of `747`, but the `aircraftType` value in the database's `Aircraft` table (for this `tailNumber`) is `MD-11`. Let's see what happens to the database upon Test execution:

Figure 83: Ruletest with Existing Record

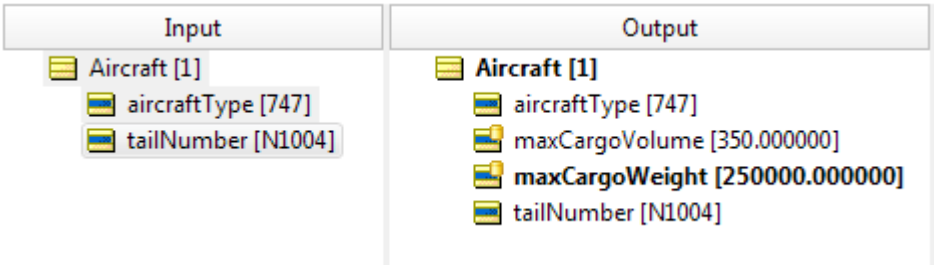
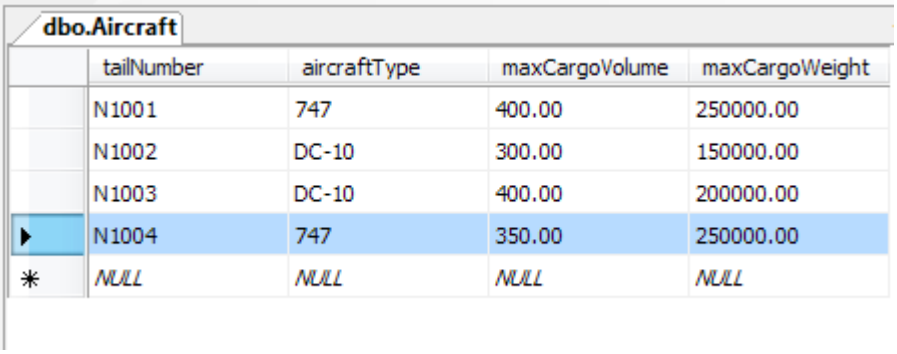


Figure 84: Final State of Database Table `Aircraft`



dbo.Aircraft

tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
N1001	747	400.00	250000.00
N1002	DC-10	300.00	150000.00
N1003	DC-10	400.00	200000.00
N1004	747	350.00	250000.00
NULL	*NULL*	*NULL*	*NULL*

Once again, when a data mismatch is encountered, the data in the Input Ruletest (simulating the request payload) is given higher priority than the data retrieved from the database. Furthermore, the data in the Input Ruletest satisfies the rule, so it fires and causes `maxCargoWeight` to receive a value of `250000`, as shown above. Unlike before, however, the new `maxCargoWeight` value is updated in the database.

9

Inside Corticon Server

This section describes how Corticon Server operates. The topics illustrate its enterprise-readiness. For details, see the following topics:

- The basic path
- Ruleflow compilation - the .eds file
- Multi-threading and concurrency reactors and server pools
- State
- Dynamic discovery of new or changed Decision Services
- Replicas and load balancing
- Exception handling

The basic path

Client applications ("consumers") invoke Corticon Server, sending a data payload as part of a request message. The invocation of Corticon Server can be either indirect (such as when using SOAP) or direct (such as when making in-process Java calls). This request contains the name of the `Corticon` Decision Service (the Decision Service Name assigned in the Deployment Descriptor file – see Deployment Console section) that should process the payload. Corticon Server matches the payload to the Decision Service and then commences execution of that Decision Service. One Corticon Server can manage multiple, different Decision Services, which may each reference different Vocabularies.

Ruleflow compilation - the .eds file

*Ruleflow*s are compiled on-the-fly during *Ruleflow* deployment or Corticon Server startup.

When Corticon Server detects a new or modified *Ruleflow* (`.erf` file) referenced in a `Deployment Descriptor` file (`.cdd`) or `addDecisionService()` API call, it compiles the `.erf` into an executable version, with file suffix `.eds`. These new `.eds` files are stored inside the Sandbox: `[CORTICON_WORK_DIR]\SER\CcServer\Tomcat\CcServer \Sandbox\DoNotDelete\DecisionServices`. Once a Decision Service has been compiled into an `.eds` file, the regular Corticon Server maintenance thread takes over and loads, unloads, and recompiles deployed `.erf` files as required.

If you want to pre-compile *Ruleflow*s into `.eds` files prior to deployment on Corticon Server, the **Pre-Compile** option in the Deployment Console enables this. See the *Deployment* chapter for more details.

Note:

In versions prior to 5.2, *Ruleflow*s were compiled during the Corticon Studio's **Save** processing of `.ers` and `.erf` assets .

Multi-threading and concurrency reactors and server pools

Corticon Server has the ability to create multiple copies ("instances") of a deployed Decision Service and execute them simultaneously in a multi-threading environment. An instance of a deployed Decision Service in the Corticon Server pool awaiting consumption by an incoming request is called a **Reactor**.

Each incoming request is processed in its own thread of execution. That is, each Reactor runs in a separate thread. Corticon Server does not create threads or perform thread management.

Note: Exception: A maintenance thread is created at Corticon Server startup time. This thread does not interact with clients of the Corticon Server. Its sole job is to scan for changes in the *Ruleflow* (or their included Rulesheet files) for dynamic reload purposes.

Rather, the thread of execution is the one assigned by the enclosing container that receives the request (such as, SOAP Servlet or Java Servlet in a web server, EJB in a J2EE application server). In the case where an in-process Java call is made from a client application, the thread of execution is the same one in which the client runs.

All requests, regardless of source, resolve to an `execute()` API method call to Corticon Server. The `execute()` method associates the incoming request to the compiled code of the requested Decision Service (the `.eds` file).

For example, if a Decision Service named `processLoan.erf` is configured with a maximum pool size of 10, then the Corticon Server can create up to 10 Reactors of this Decision Service to handle up to 10 simultaneous requests for that service. If, after processing these 10 requests, subsequent concurrent demand is less than 10, the Corticon Server will reduce the pool size accordingly to release working memory. The period of time elapsed before these inactive Reactors are released is set by property `com.corticon.ccserver.inactivity` in CcServer.properties. Note that the Corticon Server does not actually force the JVM's garbage collection to clean up – it simply notifies the JVM that garbage collection may be performed and the JVM triggers it according to its own processes. Released Reactors are "swept up" by garbage collection when the JVM performs it.

On the other hand, if 15 simultaneous requests for `processLoan.erf` are received, then Corticon Server will spawn Reactors to handle the demand until its maximum pool size limit for that Decision Service is reached. After the limit is reached, additional requests will be queued until a Reactor finishes a request and becomes available for assignment to a queued request.

Pool sizing is set at the Decision Service level in the Deployment Descriptor file (`.cdd`). See the Deployment Console section for details on setting the pool size in a `.cdd` file. Pool size parameters are also contained in `addDecisionService()` API method described here.

State

Reactor state

A Reactor is an executable *instance* of a deployed *`Ruleflow`*/Decision Service. Corticon Server acts as the broker to one or more Reactors for each deployed *`Ruleflow`*. During *`Ruleflow`* execution, the Reactor is a stateless component, so all data state must be maintained in the message payloads flowing to and from the Reactor.

If a deployed *`Ruleflow`* is composed of multiple *`Rulesheet`*s, state is preserved across those *`Rulesheet`*s as the *`Rulesheet`*s successively execute within the *`Ruleflow`*. However, no interaction with the client application occurs between or within *`Rulesheet`*s. After the last *`Rulesheet`* within the *`Ruleflow`* is executed, the results are returned back to the client as a CorticonResponse message. Upon sending the CorticonResponse message, the Reactor is released and returned to the Server pool, awaiting assignment of a new, incoming CorticonRequest thread of execution.

As an integrator, you must keep in mind that there are only two ways for you to retain state upon completion of a *`Ruleflow`* or Decision Service execution.

1. Receive and parse the data from within the CorticonResponse message. In the case of integration Option 1 or 2, the data is contained in the XML document payload or string/JDOM argument. In the case of Option 3, the data consists of Java business objects in a collection or map.
2. Persist the results of a Decision Service execution to an external database.

Once a Decision Service execution has completed, the Reactor itself does not remember anything about the data it just processed. After the CorticonResponse message is sent, the Reactor is returned to the pool in preparation for the next transaction (the next CorticonRequest).

Corticon Server state

Although data state is not maintained by Reactors from transaction-to-transaction, the names and deployment settings of Decision Services deployed to Corticon Server are maintained. A file named `ServerState.xml`, located in `[CORTICON_WORK_DIR]\CcServerSandbox\DoNotDelete`, maintains a record of the *`Ruleflow`*s and deployment settings currently loaded on Corticon Server. If Corticon Server inadvertently shuts down, or the container crashes, then this file is read upon restart and the prior Server state is re-established automatically.

A new API method initializes Corticon Server and forces it to read the `ServerState.xml` file. If the file cannot be found, then Corticon Server initializes in an empty (unloaded) state, and will await new deployments. `Initialize()` need only be called once per Server session - subsequent calls in the same session will be ignored. If other APIs are called prior to calling `initialize()`, Corticon Server will call `initialize()` itself first before continuing.

Turning off server state persistence

By default, Corticon Server automatically *creates and maintains* the `ServerState.xml` document during normal operation, and *reads* it during restart. This allows it to recover its previous state in the event of an unplanned shutdown (such as a power failure or hardware crash)

However, Corticon Server can also operate without the benefit of `ServerState.xml`, either by not reading it upon restart, or by not creating/maintaining it in the first place. In this mode, an unplanned shutdown and restart results in the loss of any settings made through the Corticon Server Console. For example, any properties settings made or `.eds` files deployed using the Console will be lost. If an `autoloaddir` property has been set inside `CcConfig.jar`, Corticon Server will still attempt to read `.cdd` files and load their `.erf` files automatically

To enable or disable creation of the `ServerState.xml` document, use the `com.corticon.server.serverstate.persistchanges` property located in CcServer.properties. to allow/prevent Corticon Server reading `ServerState.xml`, use the `com.corticon.server.serverstate.load property`, also located in CcServer.properties.

You can customize Corticon Server's state and restart behavior by combining these two property settings:

serverstate .persistchanges	serverstate .load	Server Restart Behavior
`true`	`true`	Corticon Server maintains `ServerState.xml` during operation, and automatically reads it upon restart to restore to the old state.
`true`	`false`	Corticon Server maintains `ServerState.xml` during operation, but does NOT automatically read it upon restart. New *`Server`* state upon restart is unaffected by `ServerState.xml`. This allows a system administrator to manually control state restoration from the `ServerState.xml`, if preferred.
`false`	`true`	Corticon Server attempts to read `ServerState.xml` upon restart, but finds nothing there. No old state restored.
`false`	`false`	no `ServerState.xml` document exists, and Corticon Server does not attempt to read it upon restart. No old state restored.

Dynamic discovery of new or changed Decision Services

The location of the Deployment Descriptor file(s) is identified using the `loadFromCdd()` or `loadFromCddDir()` API methods, which may be included in a deployment wrapper class (Servlet, EJB, and similar) or directly invoked from a client. See *Telling the Server Where to find Deployment Descriptor Files* for more details. A Deployment Descriptor file, in turn, contains the location of each available Decision Service. As new Decision Services are added, the Corticon Server periodically checks to see if the Deployment Descriptor files have changed or if new ones have been added. If so, the Corticon Server updates the pool for the new or modified Decision Service(s). The frequency of this check is controlled by property `com.corticon.ccserver.serviceIntervals` in CcServer.properties. The default value is 30 seconds. Alternatively, an API call to the Corticon Server can directly load new Decision Services (or sets of Decision Services).

The dynamic update monitor starts automatically by default but may be shut off by setting `com.corticon.ccserver.dynamicupdatemonitor.autoactivate` in CcServer.properties to `false`.

Replicas and load balancing

In high-volume applications, enterprises typically deploy replicas of their web application servers across multiple CPUs. The Corticon Server, as a well-behaved Java service, can be distributed across these replicas. Additional Corticon Server licenses may be necessary to support such a configuration.

A variety of means exist in modern architectures to spread the incoming workload across these replicas. These include special load balancing servers, clustering features within J2EE application servers, and custom solutions.

Exception handling

When an exception occurs, the Corticon Server throws Java exceptions. These are documented in the *JavaDoc*.

10

Decision Service versioning and effective dating

Corticon Server can execute Decision Services according to the preferred version or the date of the request.

This chapter describes how the `Version` and `Effective/Expiration Date` parameters, when set, are processed by the Corticon Server during Decision Service invocation. Assigning Version and Effective/Expiration Dates to a *`Ruleflow`* is described in the topic *"Ruleflow versioning & effective dating" in the Rule Modeling Guide.*

For details, see the following topics:

- Deploying Decision Services with identical Decision Service names
- Invoking a Decision Service by version number
- Invoking a Decision Service by date
- How Test Decision Services differ from Production (live) Decision Services
- Determining which Decision Service to execute against based on Versions and Effective/Expires Dates
- Summary of major version and effective timestamp behavior

Deploying Decision Services with identical Decision Service names

Ordinarily, all Decision Services deployed to a single Corticon Server must have unique Decision Service Names. This enables the Corticon Server to understand the request when external applications and clients invoke a specific Decision Service by its name. A Decision Service's Name is one of the parameters defined in the *Deployment Console* and included in a Deployment Descriptor file (`.cdd`). If Java APIs are used in the deployment process instead of a Deployment Descriptor file then Decision Service Name is one of the arguments of the `addDecisionService()` method. See Ruleflow deployment on page 39 for a refresher.

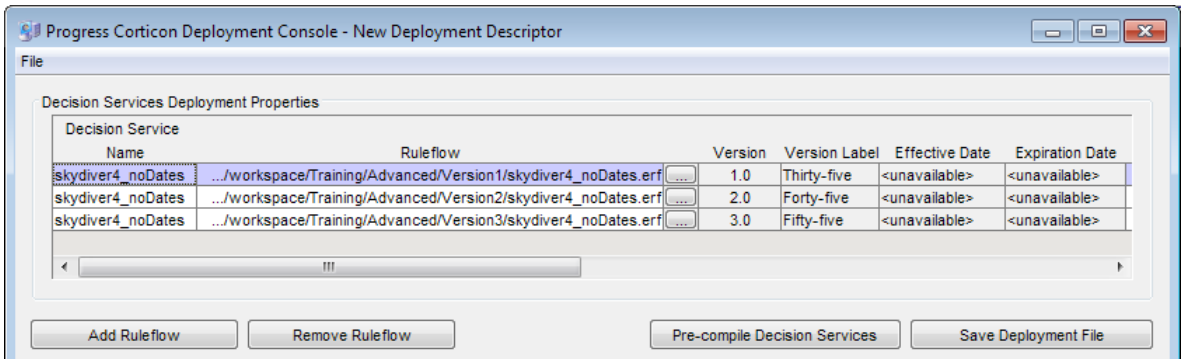
However, the Decision Service Versioning and Effective Dating feature makes an exception to this rule. Decision Services with identical Decision Service Names may be deployed on the same Corticon Server if and only if:

- They have different Major version numbers; or
- They have same Major yet different Minor version numbers

To phrase this requirement differently, Decision Services deployed with the same Major Version and Minor Version number must have different Decision Service Names.

The *Deployment Console* shown in the following figure displays the parameters of a Deployment Descriptor file with three Decision Services listed.

Figure 85: Deployment Console with Three Versions of the same Decision Service



Notice:

- All three Decision Service Names are the same: `skydiver4`.
- All three *Ruleflow* filenames are the same: `skydiver4.erf`.
- Each *Ruleflow* deploys a different *Rulesheet*. Each *Rulesheet* has a different file name, as shown on the following pages.
- The file locations (paths) are different for each *Ruleflow*. This is an operating system requirement since no two files in the same directory location may share a filename.
- All three Decision Services have different (Major) Version numbers.

It is also possible to place all *Ruleflow* files (`.erf`) in the same directory location as long as their filenames are different. Despite having different *Ruleflow* filenames, they may still share the same Decision Service Name as long as their Version or Effective/Expiration Dates are different.

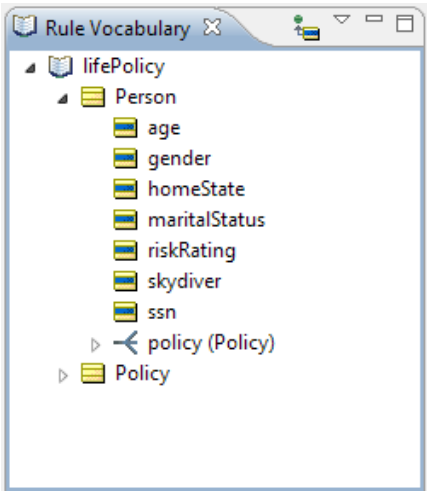
Invoking a Decision Service by version number

Both Corticon Server invocation mechanisms -- SOAP request message and Java method -- provide a way to specify Decision Service Major.Minor Version.

Creating sample Rulesheets and Ruleflows

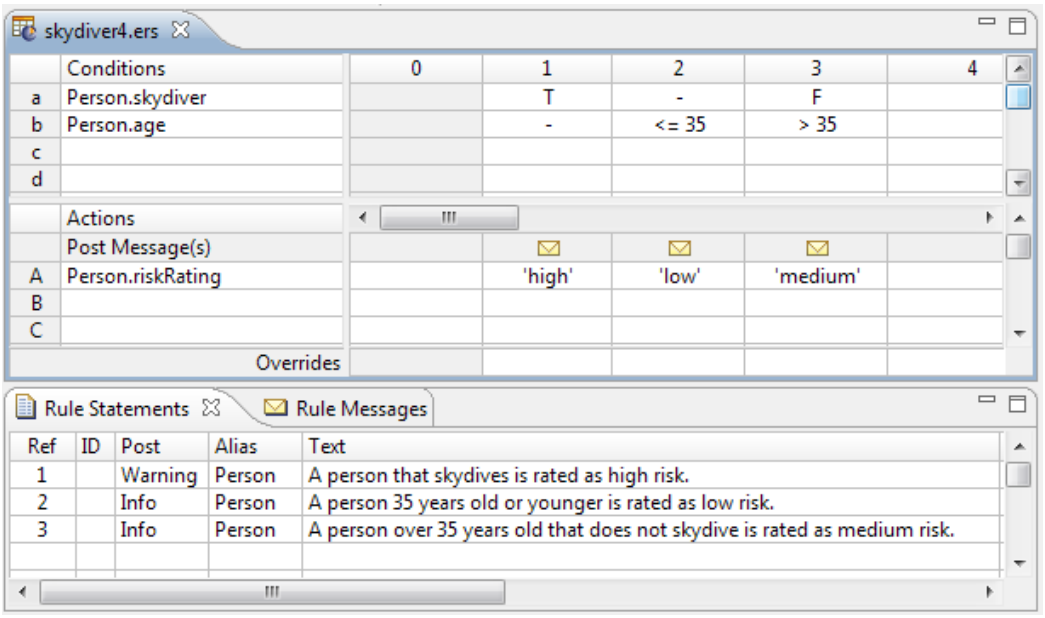
The Ruleflows we will use in this section are based on *`Rulesheet`* variations on the one rule variation. Notice that the only difference between the three *`Rulesheet`*s is the threshold for the age-dependent rules (columns 2 and 3 in each *`Rulesheet`*). The age threshold is `35`, `45`, and `55` for Version 1, 2 and 3, respectively. This variation is enough to illustrate how the Corticon Server distinguishes Versions in runtime. The Vocabulary we will use is the `lifePolicy.ecore`, located in the `Training/Advanced` project.

Figure 86: Sample Vocabulary for demonstrating versioning



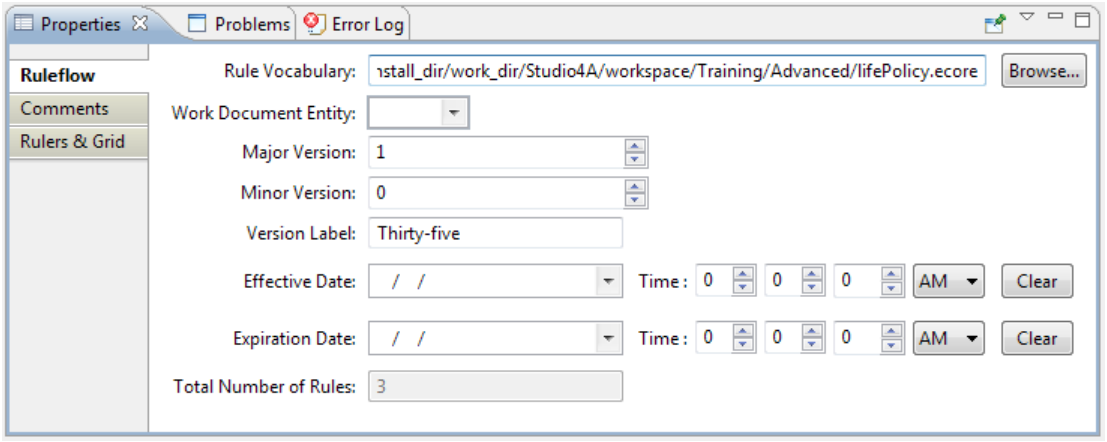
We know we want to have more than one Ruleflow with the same name and differing versions, so we first used **File > New Folder** to place a `Version1` folder in the project. Then we created a Rulesheet for defining our policy risk rating that considers age 35 as a decision point, as shown:

Figure 87: Rulesheet skydiver4.ers in folder Version1



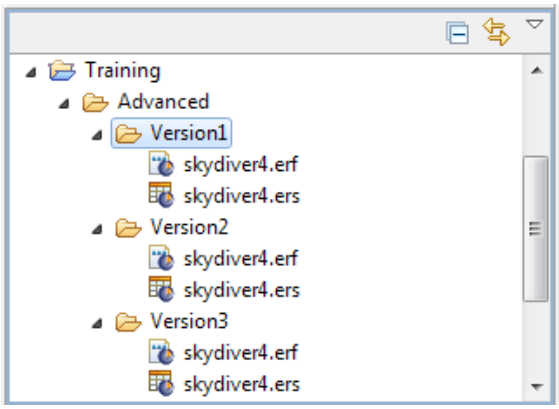
We created a new Ruleflow and added the Version1 `skydiver4.ers` Rulesheet to it. Then we set the Major version to `1` and the Minor version to `0`. The label `Thirty-five` was entered to express the version in natural language.

Figure 88: Ruleflow in folder Version1 and set as Version 1.0



After saving both files, right-click on the `Version1` folder in the **Projects** tab, and then choose **Copy**. Right-click **Paste** at the `Advanced` folder level, naming the folder `Version2`. Repeat to create the `Version3` folder. Your results look like this:

Figure 89: Folders that distinguish three versions



Note: In the examples in this section, the Ruleflows, Deployment Descriptor, and Decision Services names are elaborated as `_dates` and `_noDates` just so that we can deploy both versioned and effective-dated Decision Services at the same time.

We proceed to edit the Rulesheets and Ruleflows in the copied folders as shown, first for Version2:

Figure 90: Rulesheet skydiver4.ers in folder Version2

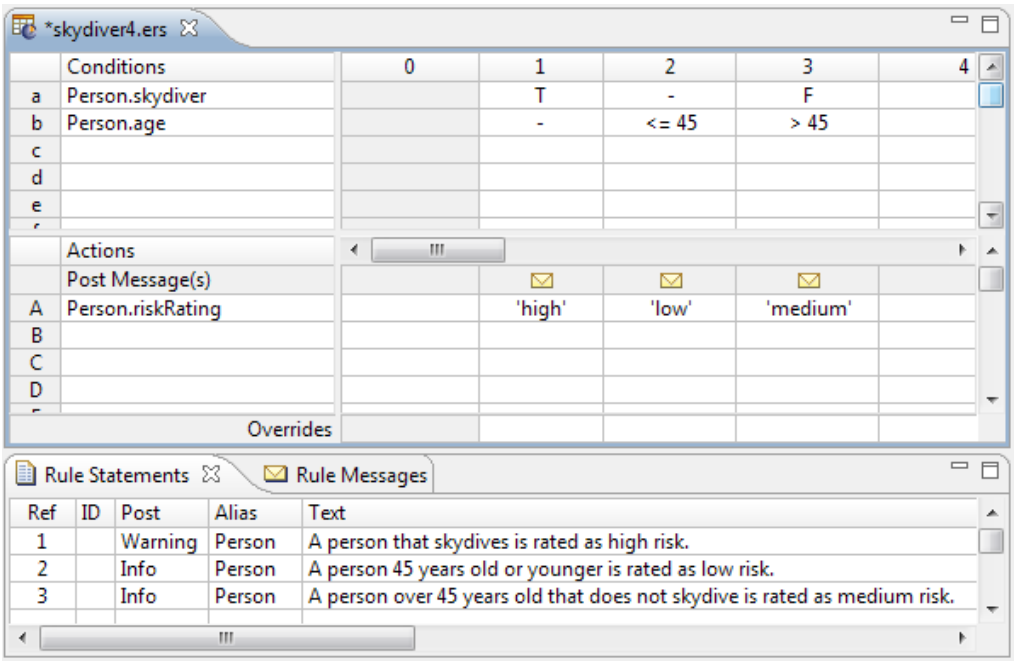
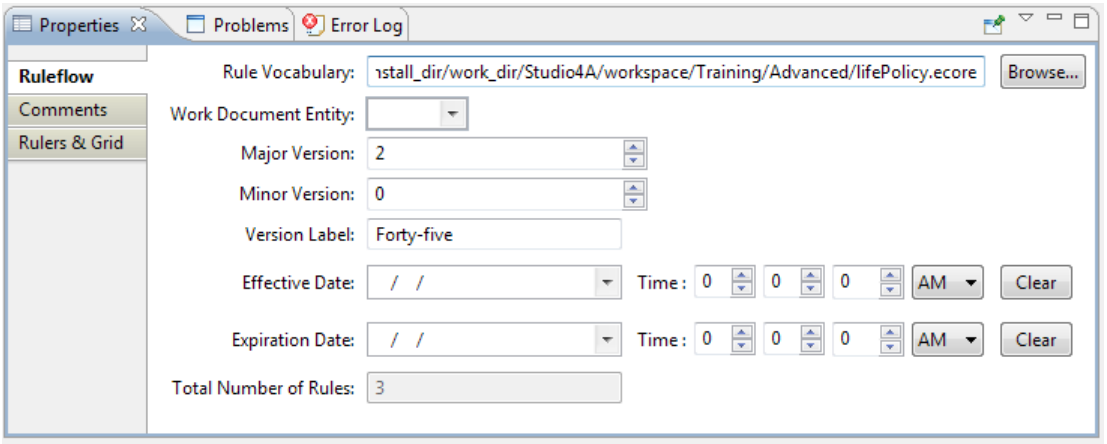


Figure 91: Ruleflow in folder Version2



And then for Version 3:

Figure 92: Rulesheet skydiver4.ers in folder Version3

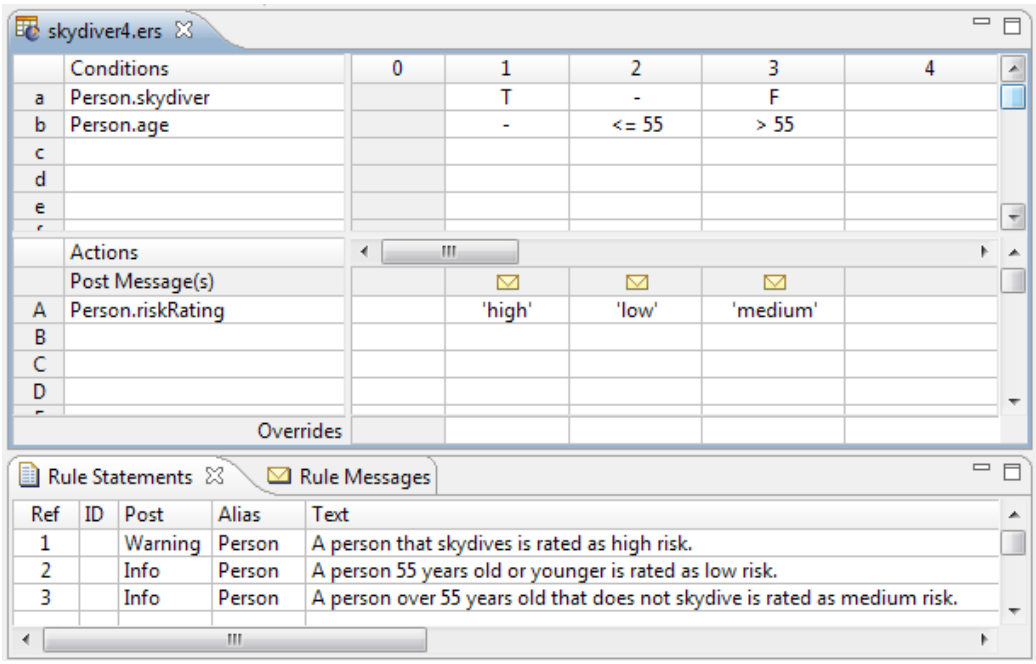
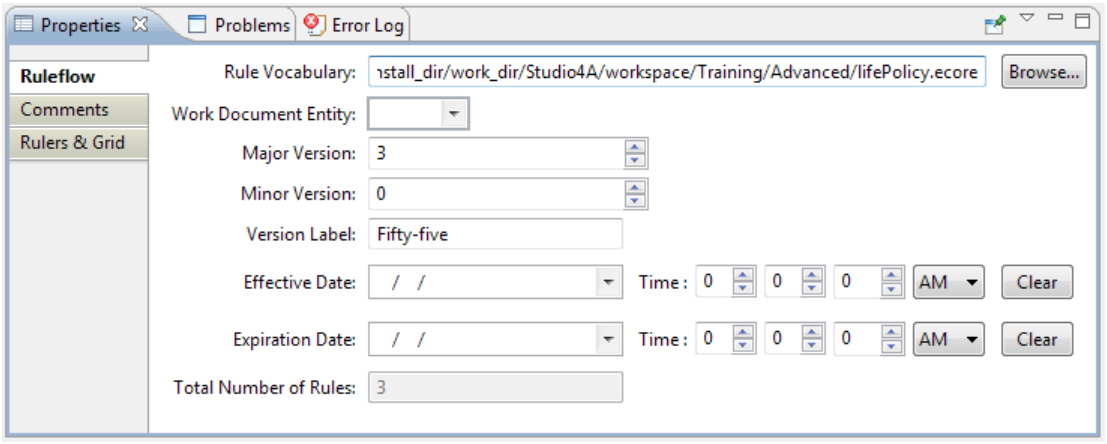


Figure 93: Ruleflow in folder Version3

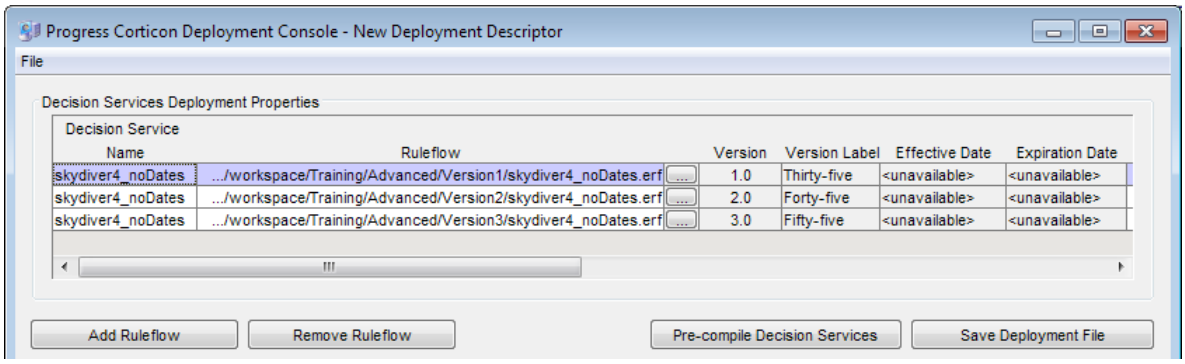


Creating and deploying the sample Decision Services

To create and deploy the sample Ruleflows:

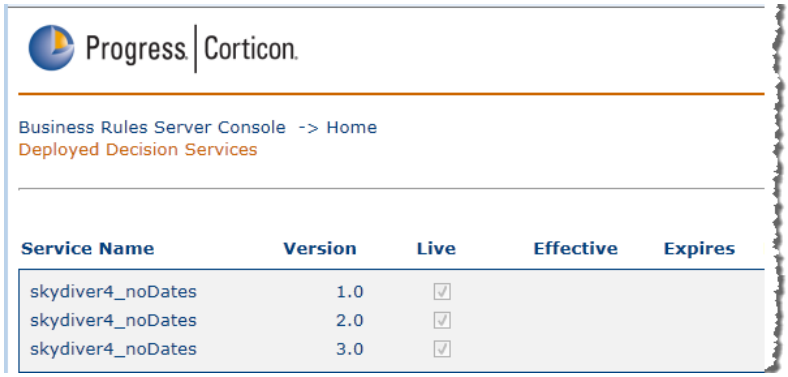
1. Open the Deployment Console, and then add the three Ruleflows we created, all under the same Decision Service name, as shown:

   Figure 94: Deployment Console with the Sample Ruleflows



2. Click **Pre-compile Decision Services**, and save the files in a local output folder.
3. Copy the three `.eds` files you created to a network-accessible directory, such as `[CORTICON_HOME]\Server\Tomcat\Compiled_EDS_Files` or simply `C:\Compiled_EDS_Files`.
4. Start the Corticon Tomcat Server.
5. In a browser, go the URL `http://localhost:8082/axis/`, and then log in with the user name and password `admin`.
6. Click **Deploy Decision Services**, and then browse to each of the `.eds` file you staged to add it to the deployment, as shown:

   Figure 95: Server Console deploying the Sample Decision Services



7. Confirm the deployment, by starting `testServerAxis.bat` (at `[CORTICON_HOME]\Server\Tomcat\`, and entering `120 - Get Decision Service Names`. The results are as shown:

   Figure 96: Confirming Deployment of Decision Service skydiver4

```
Enter transaction number:
120
AllocateTrade;Candidates;ProcessOrder;skydiver4_dates;skydiver4_noDates

Transaction completed.
```

8. Notice that the Decision Service `skydiver4_noDates` shows up only **once** in the list, even though there are actually *three* different versions of this Decision Service deployed and loaded on Corticon Server. This is normal. Corticon Server "summarizes" all deployed versions of the Decision Service as a single entry on the list, even though each entry on the list may have

multiple versions. So rather than thinking of these as three *different* Decision Services – think of them as three *versions* of the *same* Decision Service.

Specifying a version in a SOAP request message

In the `CorticonRequest` complexType, notice:

```
<xsd:attribute name="decisionServiceTargetVersion" use="optional"
type="xsd:decimal" /
```

In order to invoke a specific Major.Minor version of a Decision Service, the Major.Minor version number must be included as a value of the `decisionServiceTargetVersion` attribute in the message sample, as shown above.

As the `use` attribute indicates, specifying a Major.Minor version number is optional. If multiple Major.Minor versions of the same Decision Service Name are deployed simultaneously and an incoming request fails to specify a particular Major Version number, then Corticon Server will execute the Decision Service with *highest* version number.

If multiple instances of the same Decision Service Name and Major version number are deployed and an incoming request fails to specify a Minor version number, then Corticon Server will execute the live Decision Service with highest Minor version number of the Major version. For example, if you have 2.1, 2.2, and 2.3, and you specify 2, your request will be applied as 2.3. Note that this applies to LIVE decision services and not TEST decision services: they require a Major.Minor version.

Note: Refer to Service contract and message samples on page 159 for full details of the XML service contracts supported (XSD and WSDL).

Let's try a few invocations using variations of the following message:

```
<CorticonRequest xmlns="urn:decision:tutorial_example"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
decisionServiceName="skydiver4_noDates"
decisionServiceTargetVersion="1.0">
<WorkDocuments>
    <Person id="Person_id_1">
 <age>30</age>
 <skydiver>false</skydiver>
 <ssn>111-11-1111</ssn>
    </Person>
</WorkDocuments>
</CorticonRequest>
```

Copy this text and save the file with a useful name such as `Request_noDates_1.0.xml` in a local folder.

Run `testServerAxis` and then choose command `130` to execute the request. After it runs, you are directed to the output folder to see the result, which look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:CorticonResponse xmlns:ns1="urn:decision:tutorial_example"
xmlns="urn:decision:tutorial_example" decisionServiceName="skydiver4_noDates"
 decisionServiceTargetVersion="1.0">
```

```
      <ns1:WorkDocuments>
        <ns1:Person id="Person_id_1">
          <ns1:age>30</ns1:age>
          <ns1:riskRating>low</ns1:riskRating>
          <ns1:skydiver>false</ns1:skydiver>
          <ns1:ssn>111-11-1111</ns1:ssn>
        </ns1:Person>
      </ns1:WorkDocuments>
      <ns1:Messages version="1.0">
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>A person 35 years old or younger is rated as low
risk.</ns1:text>
          <ns1:entityReference href="#Person_id_1" />
        </ns1:Message>
      </ns1:Messages>
    </ns1:CorticonResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Note that the age stated is `35`, which is what we defined version `1.0` of the Decision Service. This should be no surprise – we specifically requested version `1.0` in our request message. Corticon Server has honored our request. .

Let's prove the technique by editing the request message to specify another version:

```
<CorticonRequest xmlns="urn:decision:tutorial_example"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
decisionServiceName="skydiver4_noDates"
decisionServiceTargetVersion="2.0">
<WorkDocuments>
    <Person id="Person_id_1">
 <age>30</age>
 <skydiver>false</skydiver>
 <ssn>111-11-1111</ssn>
    </Person>
</WorkDocuments>
</CorticonRequest>
```

The only edit is to change the version from `1.0` to `2.0`. Now execute the test using command `130`.

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:CorticonResponse xmlns:ns1="urn:decision:tutorial_example"
xmlns="urn:decision:tutorial_example" decisionServiceName="skydiver4_noDates"
 decisionServiceTargetVersion="2.0">
      <ns1:WorkDocuments>
        <ns1:Person id="Person_id_1">
          <ns1:age>30</ns1:age>
          <ns1:riskRating>low</ns1:riskRating>
          <ns1:skydiver>false</ns1:skydiver>
          <ns1:ssn>111-11-1111</ns1:ssn>
        </ns1:Person>
      </ns1:WorkDocuments>
      <ns1:Messages version="2.0">
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>A person 45 years old or younger is rated as low
risk.</ns1:text>
          <ns1:entityReference href="#Person_id_1" />
        </ns1:Message>
      </ns1:Messages>
```

```
    </ns1:CorticonResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Corticon Server has handled our request to use version `2.0` of the Decision Service. The age threshold of `45` is our hint that version `2.0` was executed.

Specifying version in a Java API call

Four versions of the `execute()` method exist -- two for Collections and two for Maps -- each providing arguments for major and major + minor Decision Service version:

```
ICcRule Messages execute(String astrDecisionServiceName,
                         Collection acolWorkObjs,
                         int aiDecisionServiceTargetMajorVersion)


ICcRule Messages execute(String astrDecisionServiceName,
                         Collection acolWorkObjs,
                         int aiDecisionServiceTargetMajorVersion,
                         int aiDecisionServiceTargetMinorVersion)

ICcRule Messages execute(String astrDecisionServiceName,
                         Map amapWorkObjs,
                         int aiDecisionServiceTargetMajorVersion)


ICcRule Messages execute(String astrDecisionServiceName,
                         Map amapWorkObjs,
                         int aiDecisionServiceTargetMajorVersion,
                         int aiDecisionServiceTargetMinorVersion)
```

where:

- `astrDecisionServiceName` is the Decision Service Name String value.
- `acolWorkObjs` is the collection of Java Business Objects – the date "payload."
- `aiDecisionServiceTargetMajorVersion` is the Major version number.
- `aiDecisionServiceTargetMinorVersion` is the Minor version number.

More information on this variant of the `execute()` method may be found in the *JavaDoc*.

Default behavior

How does Corticon Server respond when no `decisionServiceTargetVersion` is specified in a request message? In this case, Corticon Server will select the *highest* Major.Minor Version number available for the requested Decision Service and execute it.

Consider a scenario where the following versions are deployed:

```
v1.0
v1.1
v1.2
v2.0
v2.1
```

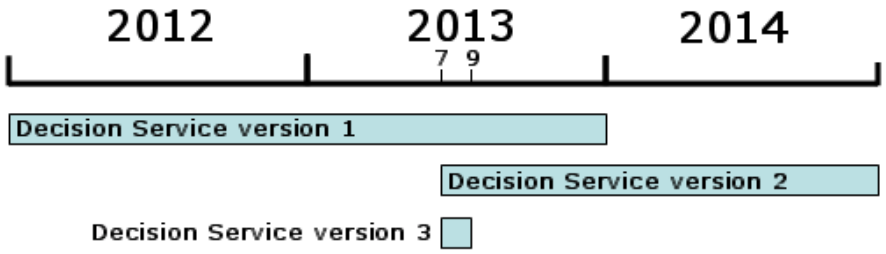
When no Version Number or EffectiveTimestamp is specified, the Server executes against v2.1 (if its Effective/Expiration range is valid). However, when Major Version 1 is passed in without an EffectiveTimestamp specified, the Server executes against v1.2 (if its Effective/Expiration range is valid).

Invoking a Decision Service by date

When multiple Major versions of a Decision Service also contain different Effective and Expiration Dates, we can also instruct Corticon Server to execute a particular Decision Service according to a date specified in the request message. This specified date is called the **Decision Service Effective Timestamp**.

How Corticon Server decides which Decision Service to execute based on the **Decision Service Effective Timestamp** value involves a bit more logic than the Major Version number. Let's use a graphical representation of the three **Decision Service Effective** and **Expiration Date** values to first understand how they relate.

Figure 97: DS Effective and Expiration Date Timeline



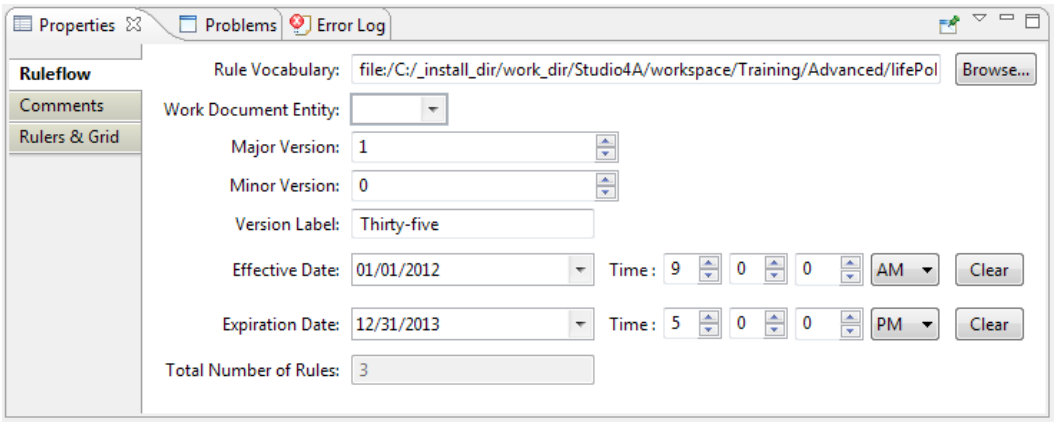
As illustrated, our three deployed Decision Services have Effective and Expiration dates that overlap in several date ranges: Version 1 and Version 2 overlap from July 1, 2013 through December 31, 2013. And Version 3 overlaps with both 1 and 2 in July-August 2013. To understand how Corticon Server resolves these overlaps, we will invoke Corticon Server with a few scenarios.

Modifying the sample Rulesheets and Ruleflows

First, let's extend or revise the Ruleflows that were specified in the previous section.

We edited the Version1 Ruleflow to set the date and time of the Effective Date and Expires Date, as shown:

Figure 98: Ruleflow in folder Version1 with dateTime set



We proceed to edit the other two Ruleflows as shown:

Figure 99: Ruleflow in folder Version2 with dateTime set

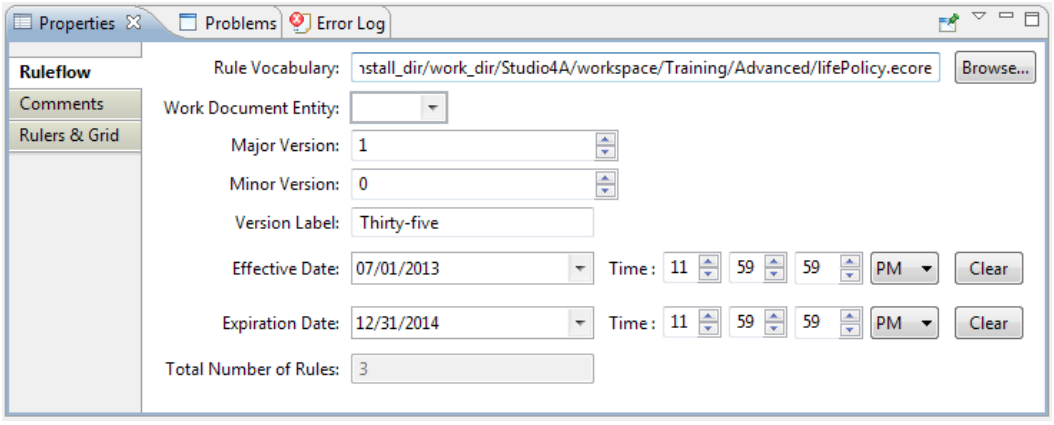
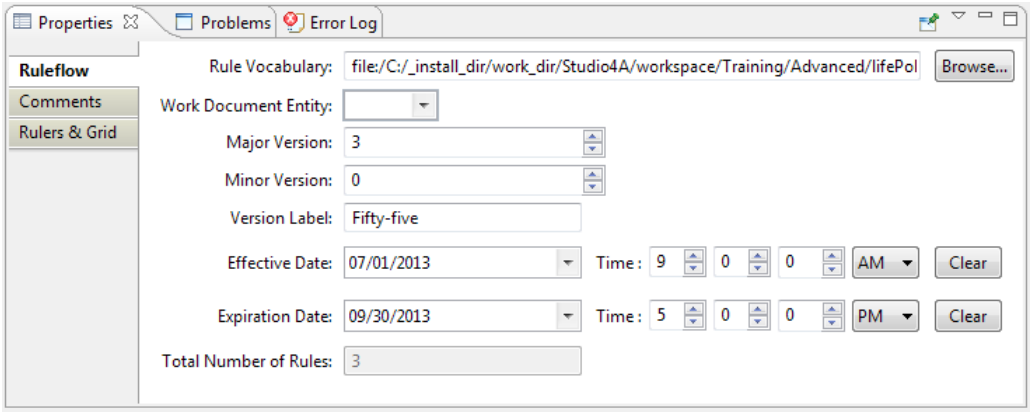


Figure 100: Ruleflow in folder Version3 with dateTime set

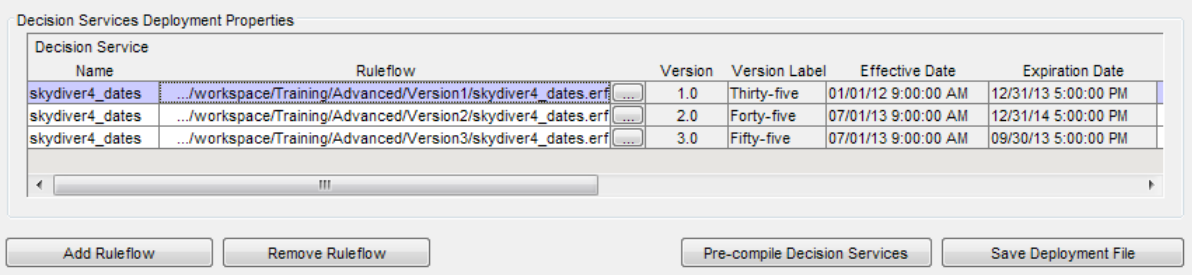


Modifying and deploying the sample Decision Services

To create and deploy the sample Ruleflows:

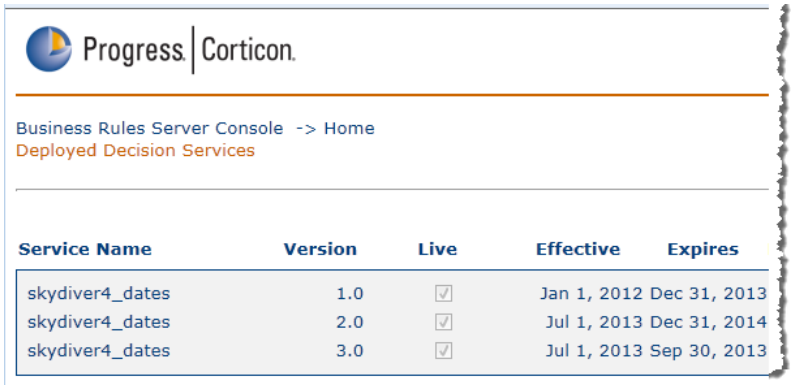
1. Open the Deployment Console, and then add the three Ruleflows we created, all under the same Decision Service name, as shown:

   Figure 101: Deployment Console with the Sample Ruleflows



2. Click **Pre-compile Decision Services**, and save the files in a local output folder.
3. Copy the three `.eds` files you created to, in this example, the subdirectory of the default Tomcat web server included in the Corticon Server installation, `[CORTICON_HOME]\Server\Tomcat\CcServer\cdd`.
4. Start the Corticon Tomcat Server.
5. In a browser, go the URL `http://localhost:8082/axis/`, and then log in with the user name and password `admin`.
6. Click **Deploy Decision Services**, and then browse to each of the `.eds` file you staged to add it to the deployment, as shown:

   Figure 102: Server Console deploying the Sample Decision Services



7. Confirm the deployment, by starting `testServerAxis.bat` (at `[CORTICON_HOME]\Server\Tomcat\`, and entering `120 - Get Decision Service Names`. The results are as shown:

   Figure 103: Confirming Deployment of Decision Service skydiver4_dates

```
Enter transaction number:
120
AllocateTrade;Candidates;ProcessOrder;skydiver4_dates;skydiver4_noDates

Transaction completed.
```

8. Notice that the Decision Service `skydiver4_dates` shows up only **once** in the list, even though there are actually *three* different versions of this Decision Service deployed and loaded on Corticon Server. This is normal. Corticon Server "summarizes" all deployed versions of the Decision Service as a single entry on the list, even though each entry on the list may have multiple versions. So rather than thinking of these as three *different* Decision Services – think of them as three *versions* of the *same* Decision Service.

Specifying Decision Service effective timestamp in a SOAP request message

As with `decisionServiceTargetVersion`, the `CorticonRequest` complexType also includes an optional `decisionServiceEffectiveTimestamp` attribute. This attribute (again, we're talking about attribute in the XML sense, not the Corticon Vocabulary sense) is included in all service contracts generated by the *`Deployment Console`* - refer to Service contract and message samples on page 159 for full details of the XML service contracts supported (XSD and WSDL).

The relevant section of the XSD is shown below:

```
<xsd:attribute name="decisionServiceEffectiveTimestamp" use="optional"
type="xsd:dateTime" />
```

Updating `CorticonRequest` with `decisionServiceEffectiveTimestamp` according to the XSD, our new XML payload looks like this:

```
<CorticonRequest xmlns="urn:decision:tutorial_example"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
decisionServiceName="skydiver4_dates"
decisionServiceEffectiveTimestamp="8/15/2012">
<WorkDocuments>
    <Person id="Person_id_2">
 <age>42</age>
 <skydiver>true</skydiver>
 <ssn>111-22-1111</ssn>
    </Person>
</WorkDocuments>
</CorticonRequest>
```

Sending this request message using `testServerAxis` as before, the response from Corticon Server is:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:CorticonResponse xmlns:ns1="urn:decision:tutorial_example"
xmlns="urn:decision:tutorial_example"
decisionServiceEffectiveTimestamp="8/15/2012"
decisionServiceName="skydiver4_dates">
      <ns1:WorkDocuments>
        <ns1:Person id="Person_id_2">
          <ns1:age>42</ns1:age>
          <ns1:riskRating>medium</ns1:riskRating>
          <ns1:skydiver>false</ns1:skydiver>
          <ns1:ssn>111-22-1111</ns1:ssn>
        </ns1:Person>
      </ns1:WorkDocuments>
      <ns1:Messages version="1.0">
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>A person over 35 years old that does not skydive is rated
 as medium risk.</ns1:text>
          <ns1:entityReference href="#Person_id_2" />
        </ns1:Message>
      </ns1:Messages>
    </ns1:CorticonResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Corticon Server executed this request message using Decision Service version `1.0`, which has the Effective/Expiration Date pair of 1/1/2012—12/31/2013. That is the only version of the Decision Service "effective" for the date specified in the request message's Effective Timestamp. The version that was executed shows in the `version` attribute of the `<Messages>` complexType.

To illustrate what happens when an Effective Timestamp falls in range of more than one Major Version of deployed Decision Services, let's modify our request message with a `decisionServiceEffectiveTimestamp` of `8/15/2013`, as shown:

```
<CorticonRequest xmlns="urn:decision:tutorial_example"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
decisionServiceName="skydiver4_dates"
decisionServiceEffectiveTimestamp="8/15/2013">
<WorkDocuments>
    <Person id="Person_id_2">
 <age>42</age>
 <skydiver>true</skydiver>
 <ssn>111-22-1111</ssn>
    </Person>
</WorkDocuments>
</CorticonRequest>
```

Send this request to Corticon Server, and then examine the response:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:CorticonResponse xmlns:ns1="urn:decision:tutorial_example"
xmlns="urn:decision:tutorial_example"
decisionServiceEffectiveTimestamp="8/15/2013"
decisionServiceName="skydiver4_dates">
      <ns1:WorkDocuments>
        <ns1:Person id="Person_id_2">
          <ns1:age>42</ns1:age>
          <ns1:riskRating>low</ns1:riskRating>
          <ns1:skydiver>false</ns1:skydiver>
          <ns1:ssn>111-22-1111</ns1:ssn>
        </ns1:Person>
      </ns1:WorkDocuments>
      <ns1:Messages version="3.0">
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>A person 55 years old or younger is rated as low
risk.</ns1:text>
          <ns1:entityReference href="#Person_id_2" />
        </ns1:Message>
      </ns1:Messages>
    </ns1:CorticonResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

This time Corticon Server executed the request with version `3`. It did so because whenever a request's `decisionServiceEffectiveTimestamp` value falls within range of more than one deployed Decision Service, Corticon Server chooses the Decision Service with the *highest* Major Version number. In this case, all three Decision Services were effective on 8/15/2013, so Corticon Server chose version `3` – the highest qualifying Version – to execute the request.

Specifying effective timestamp in a Java API call

Versions of the `execute()` method exist that contain an extra argument for a specified Decision Service Version:

```
ICcRulesMessages      execute(String astrDecisionServiceName,
                              Collection acolWorkObjs,
                              Date adDecisionServiceEffectiveTimestamp)

ICcRulesMessages      execute(String astrDecisionServiceName,
                              Map amapWorkObjs,
                              Date adDecisionServiceEffectiveTimestamp)
```

where:

- `astrDecisionServiceName` is the Decision Service Name String value.
- `acolWorkObjs` is the collection of Java Business Objects – the date `payload`.
- `adDecisionServiceEffectiveTimestamp` is the DateTime Effective Timestamp.

More information on this variant of the `execute()` method may be found in the *JavaDoc* installed in `[CORTICON_JAVA_SERVER_HOME]\Server\JavaDoc\Server`. See the package `com.corticon.eclipse.server.core` Interface `ICcServer` methods of modifier type `ICcRuleMessages`.

Specifying both major version and effective timestamp

Specifying both attributes in a single request message is allowed, only where the minor version identifier is not used.

```
ICcRulesMessages      execute(String astrDecisionServiceName,
                              Collection acolWorkObjs,
                              Date adDecisionServiceEffectiveTimestamp,
                              int aiDecisionServiceTargetMajorVersion)

ICcRulesMessages      execute(String astrDecisionServiceName,
                              Map amapWorkObjs,
                              Date adDecisionServiceEffectiveTimestamp,
                              int aiDecisionServiceTargetMajorVersion)
```

Default behavior

How does Corticon Server respond when *no* `decisionServiceEffectiveTimestamp` is specified in a request message? In this case, Corticon Server will assume that the value of `decisionServiceEffectiveTimestamp` is equal to the DateTime of invocation – the DateTime *right* now. Corticon Server then selects the Decision Service which is effective now. If more than one are effective then Corticon Server selects the Decision Service with the highest Major.Minor Version number (as we saw in the overlap example).

```
<CorticonRequest xmlns="urn:decision:tutorial_example"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
decisionServiceName="skydiver4_dates">
```

```
<WorkDocuments>
    <Person id="Person_id_2">
 <age>42</age>
 <skydiver>true</skydiver>
 <ssn>111-22-1111</ssn>
    </Person>
</WorkDocuments>
</CorticonRequest>
```

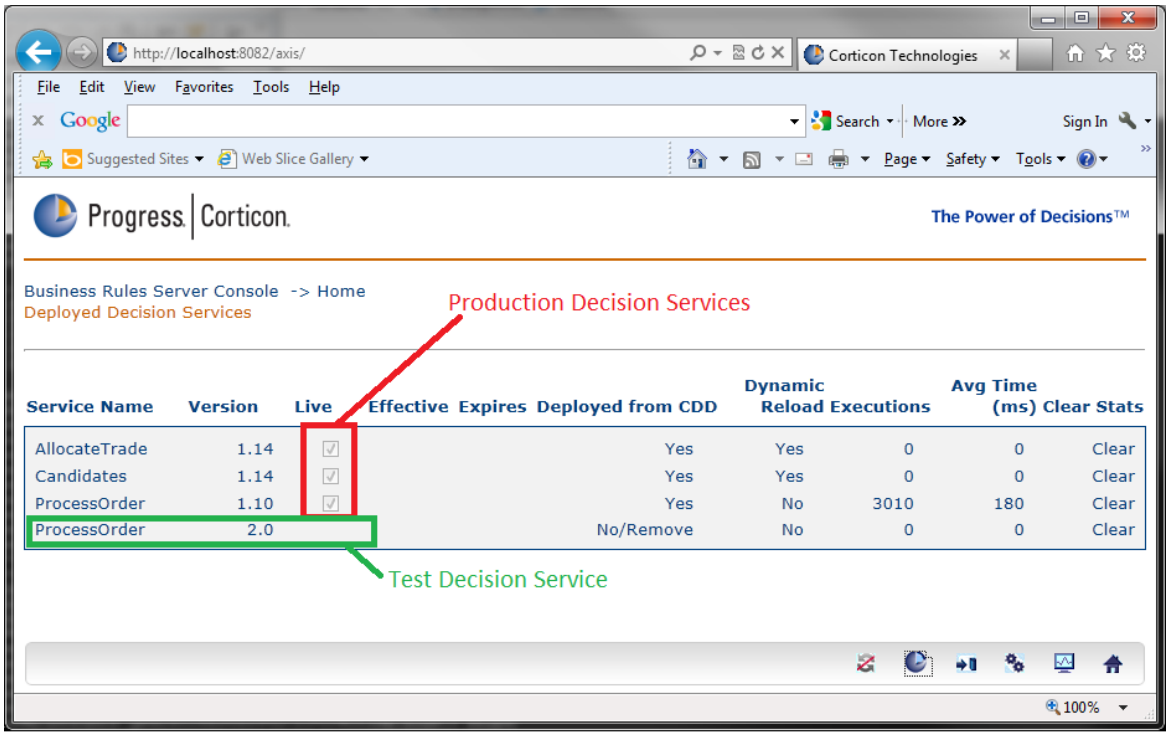
As expected, the current date (this document was drafted on 8/15/2013) was effective in all three versions. As such, the highest version applied and is noted in the reply:

```
<ns1:Messages version="3.0">
```

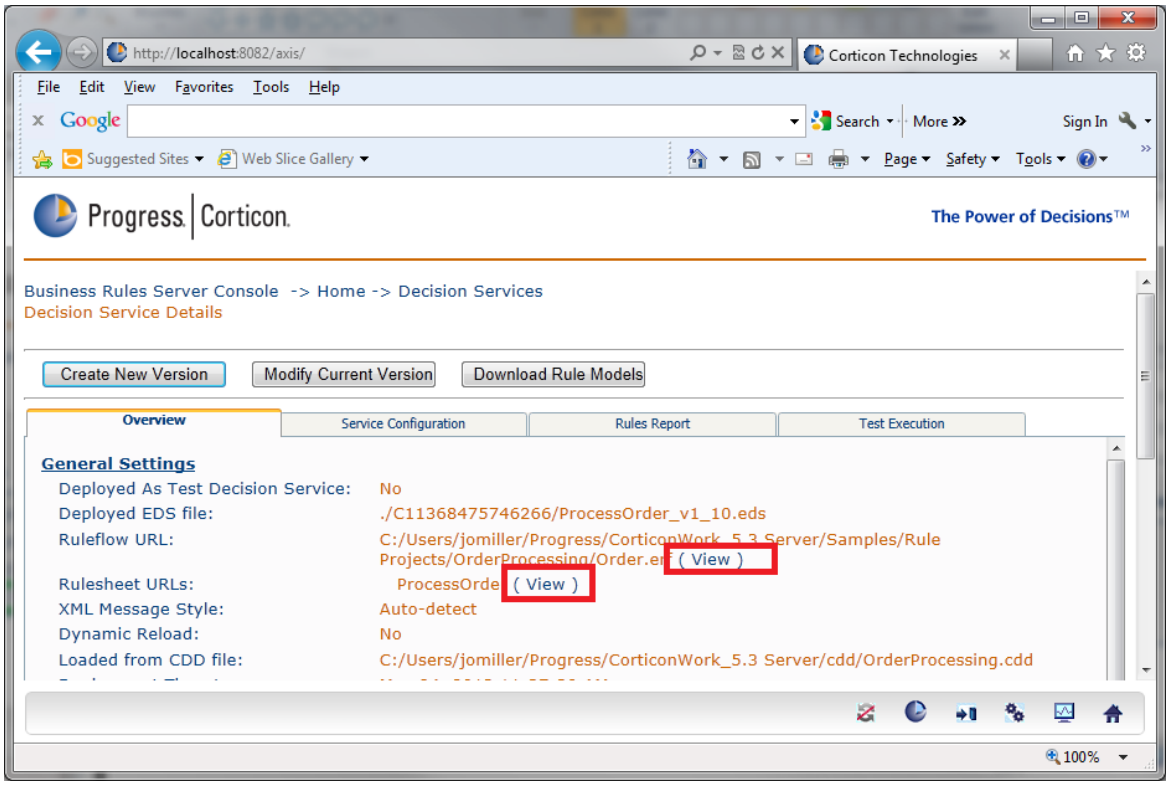
How Test Decision Services differ from Production (live) Decision Services

You can use the Corticon Server Console to modify Decision Services that have been deployed on the Corticon Server. Production Decision Services are differentiated from Test Decision Services - that's because only Test Decision Services can be modified. A Decision Service marked as Production has its “Edit” capabilities disabled.

For example:

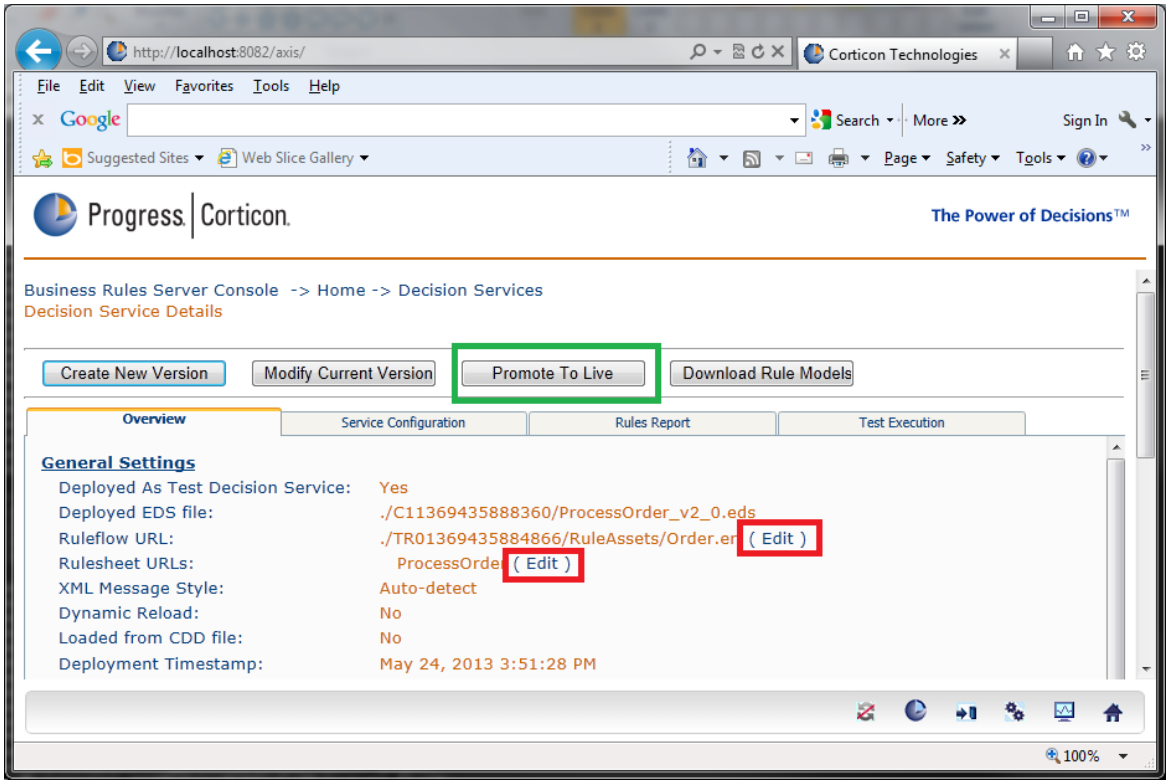


When we look at the Decision Service Detail Screen for ProcessOrder 1.10, we see that you can only “View” the Rulesheets:



When we look at the Decision Service Detail Screen for ProcessOrder 2.0, we see that you can only “Edit” the Rulesheets.

Also, after you complete your changes, you can commit the changes to Live (Production) by clicking **Promote to Live**.



Determining which Decision Service to execute against based on Versions and Effective/Expires Dates

When a user supplies non-specific information about which Decision Service they want to execute against, the Corticon Server applies an algorithm to determine what is most suitable. When this algorithm is applied, all Test Decision Services are filtered out. In a production setting, the user cannot execute against a Test Decision Service unless the CorticonRequest is very specific about the Decision Service Name, Major Version Number, and Minor Version Number. The following examples use a mixture of ProcessOrder deployments to evalaute requests, first for only versions (or none), and then for effective/expires dates that might have version specified as well.

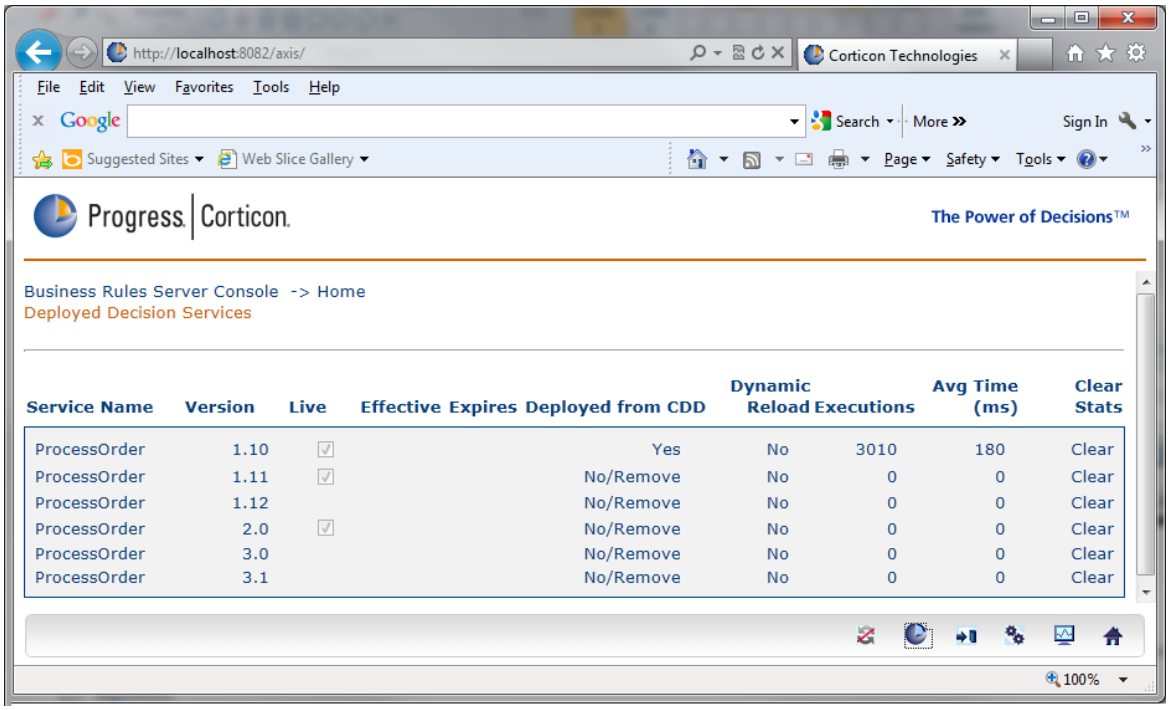
The algorithm evaluates versioning as follows:

1. If the user specifies the Decision Service Name (ProcessOrder), but does not specify the Major or Major/Minor Version Number, the Corticon Server tries to determine which version of ProcessOrder it should use.
2. If Decision Service Name only: Find the highest Production Major/Minor version for that Decision Service Name.
3. If Decision Service Name and Major Version Number: Find the highest Production Minor version for that Decision Service Name and Major Version Number.
4. If Decision Service Name and Major and Minor Version: Find that specific combination…regardless of whether it is Production or Test. If the user actually specifies the Major and Minor Version Number, assume that they know what they are doing and allow them to go against a Test Decision Service.
5. If no Decision Service is found based on this algorithm, then the Corticon Server throws a `CcServerDecisionServiceNotRegisteredException`.

The following two examples show how this algorithm applies in a variety of requests.

Example 1: Requests use versions but not effective dates

The deployed Decision Services are as shown:



Test 1: User only specifies the Decision Service Name in the payload

```
<CorticonRequest xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        decisionServiceName="ProcessOrder">
        <WorkDocuments>
```

The Algorithm will find the largest Live Major Version and then the largest Minor Version for that Major Version.

In this case, this would be version 2.0.

Test 2a: User specifies the Decision Service Name and the Major Version Number in the payload:

```
<CorticonRequest xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        decisionServiceName="ProcessOrder" decisionServiceTargetVersion="1">
        <WorkDocuments>
```

The Algorithm will find the largest Live Minor Version for that Major Version Number.

In this case, this would be version 1.11. Version 1.12 is ignored because it is not Live.

Test 2b: User specifies the Decision Service Name and the Major Version Number in the payload:

```
<CorticonRequest xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
decisionServiceName="ProcessOrder"
        decisionServiceTargetVersion="3">
        <WorkDocuments>
```

The Algorithm will find the largest Live Minor Version for that Major Version Number. However,

in this case, there is no Live Decision Service with Major Version 3. The Corticon Server will throw a `CcServerDecisionServiceNotRegisteredException`.

Test 3a: User specifies the Decision Service Name and the Major and Minor Version Number in the payload:

```
<CorticonRequest
        xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       decisionServiceName="ProcessOrder" decisionServiceTargetVersion="3.1">

        <WorkDocuments>
```

The Algorithm will try and find that specific version regardless if it is Production or Test.

In this case, there is a match and 3.1 would be used.

Test 3b: User specifies the Decision Service Name and the Major and Minor Version Number in the payload:

```
<CorticonRequest xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       decisionServiceName="ProcessOrder" decisionServiceTargetVersion="3.2">

        <WorkDocuments>
```

The Algorithm will try and find that specific version regardless if it is Production or Test.

In this case, there is Decision Service with Major and Minor Version 3.2. The Corticon Server will throw a `CcServerDecisionServiceNotRegisteredException`.

Example 2: Requests use effective dates and sometimes versions as well

The Test Cases above did not have Effective Dates in place. When Effective Dates are involved, the Algorithm to determine which Decision Service to use gets a more complex. This example describes how Effective and Expires Dates play a role in the Algorithm in determining the appropriate Decision Service to use during execution.

- When a Decision Service has no Effective or Expires Date specified, this means this Decision Service is valid from the start of time till the end of time…there is no lower or upper bounds on when this Decision Service is valid. (infinity in both directions)
- When a Decision Service has no Effective but does have an Expires Date specified, that means this Decision Service is valid from the start of time up to the specified Expires Date.
- When a Decision Service has an Effective Date but does not have an Expires Date specified, that means this Decision Service is valid from that specified Effective Date till the end of time.

In Example 1, all the Decision Services were unbounded (no Effective or Expires Date specified). In this case, none of the Decision Services were filtered out by the Algorithm based on the implied EffectiveDate value inside the CorticonRequest.

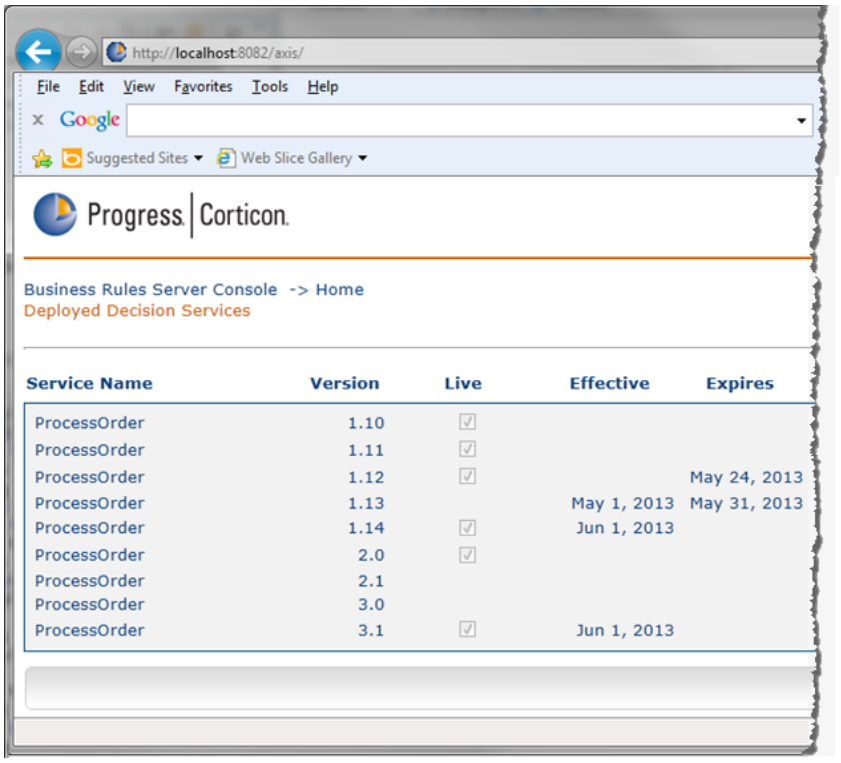
The algorithm is expanded as follows:

1. If the user specifies the Decision Service Name (ProcessOrder), but does not specify the Major or Major/Minor Version Number, or CorticonRequest TargetDate, the Corticon Server tries to determine which version of ProcessOrder it should use.
2. If Decision Service Name only: Find the highest Production Major/Minor version for that Decision Service Name that satisfies `Decision Service Effective Date < today() < Decision Service Expires Date`.
3. If Decision Service Name and CorticonRequest Target Date: Find the highest Production Major/Minor version for that Decision Service Name that satisfies `Decision Service Effective Date < CorticonRequest Target Date < Decision Service Expires Date`.
4. If Decision Service Name and Major Version Number: Find the highest Production Minor version for that Decision Service Name and Major Version Number that satisfies `Decision Service Effective Date < today() < Decision Service Expires Date`.
5. If Decision Service Name, Major Version Number, and CorticonRequest Target Date: Find the highest Production Minor version for that Decision Service Name and Major Version Number that satisfies `Decision Service Effective Date < CorticonRequest Target Date < Decision Service Expires Date`.

To simplify potential issues in determine what has priority between Major/Minor Version and Effective/Expiration Dates, an exception is thrown to the user if they specify Decision Service Name, Major Version, Minor Version, and Target Date.

Example 2

The deployed Decision Services are as shown:



Test 1a: User only specifies the Decision Service Name in the payload

```
<CorticonRequest xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        decisionServiceName="ProcessOrder">
```

```
        <WorkDocuments>
```

Since no Target Date was specified inside the CorticonRequest, `Target Date = today() = 8/1/2013`.

The Algorithm will first analyze all the Decision Services with name “ProcessOrder” and filter out those that don’t satisfy `Decision Service Effective Date < Target Date < Decision Service Expires Date`.

Decision Services that qualify include:

- Version 1.10
- Version 1.11
- Version 1.14
- Version 2.0
- Version 3.1

Test Decision Services are automatically excluded. These include Version 1.13, 2.1, and 3.0.

Now, find the largest Live Major Version and then the largest Minor Version for that Major Version from those versions that qualified.

In this case, this would be version 3.1.

Test 1b: User specifies the Decision Service Name in the payload and Effective Timestamp:

```
<CorticonRequest xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        decisionServiceName="ProcessOrder"
decisionServiceEffectiveTimestamp=”5/27/2013”>
        <WorkDocuments>
```

The Algorithm will first analyze all the Decision Services with name “ProcessOrder” and filter out those that don’t satisfy `Decision Service Effective Date < 5/27/2013 < Decision Service Expires Date`.

Decision Services that qualify include:

- Version 1.10
- Version 1.11
- Version 2.0

Version 1.12 expired 3 days prior.

Test Decision Services are automatically excluded. These include Version 1.13, 2.1, and 3.0.

Version 1.14 and 3.1 are not effective for another 4 days.

Now, find the largest Live Major Version and then the largest Minor Version for that Major Version from those versions that qualified.

In this case, this would be version 2.0.

Test 2a: User specifies the Decision Service Name and the Major Version Number in the payload:

```
<CorticonRequest xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        decisionServiceName="ProcessOrder" decisionServiceTargetVersion="1">
        <WorkDocuments>
```

Since no Target Date was specified inside the CorticonRequest, `Target Date = today() = 8/1/2013`.

The Algorithm will first analyze all the Decision Services with name “ProcessOrder” and filter out those that don’t satisfy `Decision Service Effective Date < Target Date < Decision Service Expires Date`.

Decision Services that qualify include:

- Version 1.10
- Version 1.11
- Version 1.14

All Decision Services that don’t have a Major Version Number = 1 are excluded from available list.

Test Decision Services are automatically excluded. This includes Version 1.13.

Now, find the largest Live Major Version and then the largest Minor Version for that Major Version from those versions that qualified.

In this case, this would be version 1.4.

Test 2b: User specifies the Decision Service Name, Major Version Number, and Effective Timestamp in the payload:

```
<CorticonRequest xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        decisionServiceName="ProcessOrder" decisionServiceTargetVersion="1"
          decisionServiceEffectiveTimestamp=”5/27/2013”>
          <WorkDocuments>
```

The Algorithm will first analyze all the Decision Services with name “ProcessOrder” and filter out those that don’t satisfy `Decision Service Effective Date < 5/27/2013 < Decision Service Expires Date.`

Decision Services that qualify include:

- Version 1.10
- Version 1.11

All Decision Services that don’t have a Major Version Number = 1 are excluded from available list.

Test Decision Services are automatically excluded. This includes Version 1.13.

Now, find the largest Live Major Version and then the largest Minor Version for that Major Version from those versions that qualified.

In this case, this would be version 1.11.

Test 2c: User specifies the Decision Service Name, Major Version Number, and Effective Timestamp in the payload:

```
<CorticonRequest xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        decisionServiceName="ProcessOrder"
decisionServiceTargetVersion="3”decisionServiceEffectiveTimestamp=”5/27/2013”>
```

```
<WorkDocuments>
```

The Algorithm will first analyze all the Decision Services with name “ProcessOrder” and filter out those that don’t satisfy `Decision Service Effective Date < 5/27/2013 < Decision Service Expires Date`.

Decision Services that qualify include:

- none

All Decision Services that don’t have a Major Version Number = 3 are excluded from available list.

Test Decision Services are automatically excluded. This includes Version 3.0.

Version 3.1 isn’t effective for another 4 days.

The Corticon Server will throw a `CcServerDecisionServiceNotRegisteredException`.

Test 3a: User specifies the Decision Service Name, Major and Minor Version Numbers, and Effective Timestamp in the payload:

```
<CorticonRequest xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        decisionServiceName="ProcessOrder" decisionServiceTargetVersion="3.1"

          decisionServiceEffectiveTimestamp=”5/27/2013”>
          <WorkDocuments>
```

The payload cannot have Major and Minor Version with an EffectiveTimestamp in the CorticonRequest. In this case, the Corticon Server will throw a `CcServerInvalidArgumentException`.

Summary of major version and effective timestamp behavior

Request Specifies Major Version?	Request Specifies Minor Version?	Request Specifies Timestamp?	Server Behavior
No	No	No	Execute the highest Major.Minor version Production Decision Service that is in effect based on the invocation timestamp
Yes	No	No	Execute the given Major version's highest minor version Production Decision Service that is in effect based on the invocation timestamp
Yes	Yes	No	Execute the given combined Major.Minor version *Production or Test* Decision Service that is in effect based on the invocation timestamp

Request Specifies Major Version?	Request Specifies Minor Version?	Request Specifies Timestamp?	Server Behavior
Yes	Yes	Yes	Server error, see the figure, **Server Error Due to Specifying Both Major.Minor Version and Timestamp**, above.
No	No	Yes	Execute the highest Major.Minor version Production Decision Service that is in effect based on the **specified** timestamp
Yes	No	Yes	Execute the given Major version's highest minor version Production Decision Service that is in effect based on the **specified** timestamp

11

Performance and tuning guide

This section discusses aspects of Corticon Server performance.
For details, see the following topics:

Rulesheet performance and tuning

In general, Corticon Studio includes many features that help rule authors write efficient rules. Because one of the biggest contributors to Decision Service (`Ruleflow`) performance is the number of rules (columns) in the component `Rulesheet`s, reducing this number may improve performance. Using the Compression tool to reduce the number of columns in a `Rulesheet` has the effect of reducing the number of rules, even though the underlying logic is unaffected. In effect, you can create smaller, better performing Decision Services by compressing your `Rulesheet`s prior to deployment. For more information, refer to the *Rule Modeling Guide's* chapter on "Rule Analysis and Optimization".

Server performance and tuning

Important: Before doing any performance and scalability testing when using an evaluation version of Corticon Server, check with Progress Corticon support or your Progress representative to verify that your evaluation license is properly enabled to allow unlimited concurrency. Failure to do so may lead to unsatisfactory results as the default evaluation license does not permit high-concurrency operation.

A Corticon Decision Service (*Ruleflow*) executes in the same thread as its caller. All Decision Services are stateless and have no latency; that is, they do not call out to other external services and await their response. Therefore, increasing the capacity for thread usage will increase performance. This can be done through:

- Using faster CPUs so threads are processed faster.
- Using more CPUs or CPU cores so more threads may be processed in parallel.
- Allocating more system memory to the JVM so there is more room for simultaneous threads.
- Distributing transactional load across multiple Corticon Server instances or multiple CPUs.

Optimizing pool settings for performance

When a Decision Service is deployed (via either the *Deployment Console* or API), the person responsible for deployment (typically an IT specialist) decides how many instances of the same Decision Service (Reactors) may run concurrently. This number establishes the pool size for that particular Decision Service. Different Decision Services may have different pool sizes on the same Corticon Server because consumer demand for different Decision Services may vary.

Choosing how large to make the pool depends on many factors, including the incoming arrival rate of requests for a particular Decision Service, the time required to process a request, the amount of other activity on the server box and the physical resources (number and speed of CPUs, amount of physical memory) available to the server box. A maximum pool size of one (1) implies no concurrency for that Decision Service. See Multi-threading and Concurrency or the Deployment Console's pool size section for more details

The recommendations that follow are not requirements. High-performing Corticon Server deployments may be achieved with varying configurations dictated by the realities of your IT infrastructure. Our testing and field experience suggests, however, that the closer your configuration comes to these standards, the better Corticon Server performance will be.

Configuring the runtime environment revolves around a few key quantities:

- The number of CPUs in the server box on which the Corticon Server is running.
- The number of wrappers deployed. The wrapper is the intermediary "layer" between the web/app server and Corticon Server, receiving all calls to Corticon Server and then forwarding the calls to the Corticon Server via the Corticon API set. The wrapper is the interface between deployment-specific details of an installation, and the fixed API set exposed by Corticon Server. A sample Servlet wrapper, `axis.war`, is provided as part of the default Corticon Server installation.
- The minimum and maximum pool size settings for each deployed Decision Service. These pool sizes are set in the Deployment Descriptor file (`.cdd`) created in the *`Deployment Console`*.

Single machine configuration

CPUs & Wrappers

For optimal performance, the number of wrappers (Session EJBs, Servlets, and such (Oracle WebLogic application server refers to wrappers as "Bean Pools")) deployed should never exceed the number of CPU cores on the server hardware, minus an allocation to support the OS and other applications resident on the server, including middleware. Typically, the number of these wrappers is controlled via a configuration file: the sample EJB code *`Progress Corticon`*'s default installation sets this number in the `weblogic-ejb-jar.xml file` (located in `meta-inf` of `[CORTICON_WORK_DIR]\Samples\Server\Containers\EAR\lib\CcServerAdminEJB.jar` and `CcServerExecuteEJB.jar`. Servlets are configured in a similar way. For example, a 4-core server box should have, at most, 4 wrappers deployed to it. Another example: a dedicated Corticon Server box with 16 cores should have at most 15 wrappers deployed, with 1 core of capacity reserved for OS and middleware platform.

Wrappers & pools

The number of wrappers should be greater than or equal to the highest pool setting for any deployed Decision Service. For example, take the following example deployment:

- *`Ruleflow`* #1 (Decision Service #1): min pool size = 5, max pool size = 5.
- *`Ruleflow`* #2 (Decision Service #2): min pool size = 4, max pool size = 4.
- *`Ruleflow`* #3 (Decision Service #3): min pool size = 9, max pool size = 9.

In this case, 9 deployed wrappers are optimum to ensure that unused or idle Reactors in the pool are minimized. This setting, however, may conflict with the wrapper number suggested by CPU core number. If we were starting with a fixed CPU core number, say 8, then we would want to reduce the pool size for Decision Service #3 to:

- *`Ruleflow`* #3 (Decision Service #3): min pool size = 8, max pool size = 8.

And deploy only 8 wrappers instead of 9. Had we retained the original 9/9 pool setting, the ninth Reactor in the pool would have gone unused and simply would have consumed additional memory with no benefit. On the other hand, increasing wrappers to 9 might cause a total of 9 threads of execution to be allocated to 9 reactors (any mix of Reactors for the 3 *`Ruleflow`*). Since only 8 threads can process simultaneously (only 8 physical CPU cores), then performance-robbing thread switching will occur.

Minimum and maximum pool sizes

Current testing suggests that setting minimum and maximum pool sizes equal to each other results in best performance. Although keeping a larger number of Reactors ready in the pool requires more memory, it also eliminates the time necessary to "spawn" new Reactors into the pool when transaction demand suddenly increases because the pool is already loaded with the maximum number of Reactors allowed. It is also important to note that higher minimum pool settings require more time to initialize during web/app server startup because more Reactors must be put into the pool.

Hyper-threading

Hyper-threading is an Intel-proprietary technology used to improve parallelization of computations (doing multiple tasks at once) performed on PC microprocessors. For each processor core that is physically present, the operating system addresses two virtual processors, and shares the workload between them when possible. Field experience suggests that Hyper-threading does not allow doubling of wrappers or Reactors for a given physical CPU core number. Doubling wrappers or Reactors with the expectation that Hyper-threading will double capacity will result in core under-utilization and poor performance. We recommend setting wrapper and Reactor parameters based on the assumption of one thread per CPU core.

Cluster configuration

The recommendations above also hold true in clustered environments, with the following clarifications:

CPUs & Wrappers

Because wrappers are typically located on the Main Cluster Instance and Reactors are located on the cluster machines, the direct relationship between CPUs and wrappers isn't so straightforward in clustered environments. The key relationship becomes number of CPUs on the cluster machine and the maximum pool size of any given Decision Service deployed to the *same* machine. If the number of CPUs in cluster machine A is 4, then the maximum pool size for any Decision Service deployed to cluster machine A should not exceed 4.

Wrappers and pools

Wrapper count on the Main Cluster Instance should be greater than or equal to the sum of the maximum pool sizes for any given Decision Service across all clustered machines. For example:

- Cluster machine A has Decision Service #1 deployed with min/max pool settings of 4/4.
- Cluster machine B has Decision Service #1 deployed with min/max pool settings of 6/6.
- Cluster machine C has Decision Service #1 deployed with min/max pool settings of 2/2.

Based on this example, the Main Cluster Instance should have *at* least 12 instances of the wrapper deployed to make most efficient use of the 12 available Reactors in Decision Service #1's clustered pool.

Minimum and maximum pool sizes

As with the Single Machine configuration, minimum and maximum pool size settings should also equal each other on each cluster machine

Shared directories and unique sandboxes

While sharing certain directories across multiple clustered machines is a good practice, the nodes in a cluster should not share the same `CcServerSandbox` directory. Different instances are likely to get out-of-sync with the `ServerState.xml`, thereby causing instability across all instances. Each cluster member should have its own `CcServerSandbox` with its own `ServerState.xml`, yet share the same Deployment Directory (`/cdd`) directory. Then, when there is a change to a `.cdd` or a RuleAsset, each node handles its own updates and its own `ServerState.xml` file.

Logging

Corticon Server log settings can have a major impact on performance. The log is set to `VIOLATION` mode by default, which only logs Corticon Server violations and exceptions. Configuring logging is described in Common properties (CcCommon.properties) on page 188.

In production deployments, Corticon Server logging should always be set to `VIOLATION` mode, as all other modes may increase transaction times by a factor of 10. `INFO` and `RULETRACE` modes are intended for testing environments and `DEBUG` mode should be enabled only on instruction from Progress technical support personnel.

Logging service name and requestor's IP address

In many cases, the Exception will contain new information about the Decision Service name and version information and the host name used to connect to the server. When this Exception is passed back to the Soap Servlet, the Client IP will be added to the log statement:

```
CorticonRequest
<CorticonRequest xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" decisionServiceEffectiveTimestamp="1/1/2013"
decisionServiceName="ProcessOrder" decisionServiceTargetVersion="1.1">


Exception
Cc|2013-06-24 15:08:19.784|Thread:http-bio-8082-exec-10|Version: 5.3.2.0 -b5059(5.3.5059)
|jomiller|ERROR|com.corticon.eclipse.soap.CcServerMessagingAxis|com.corticon.service.ccserver.exception.CcServerInvalidArgumentException:
CcServer.execute(Document, String, String) (DecisionServiceName=ProcessOrder[1, 1, 1/1/2013], HostLocation=192.168.1.7) -- Input Request
Document cannot contain both 'decisionServiceTargetVersion' containing a major and minor version number and
'decisionServiceEffectiveTimestamp' as attributes on Root Element or children on Root Element.
Object state for: com.corticon.eclipse.server.core.impl.CcServerImpl: com.corticon.eclipse.server.core.impl.CcServerImpl@1fdfe75
  at com.corticon.eclipse.server.core.impl.CcServerImpl.execute(CcServerImpl.java:870)
at com.corticon.eclipse.soap.CcServerMessagingAxis.execute(CcServerMessagingAxis.java:525) at
sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at sun.reflect.NativeMethodAccessorImpl.invoke
(NativeMethodAccessorImpl.java:57) at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) at
java.lang.reflect.Method.invoke(Method.java:601)
```

When a user accesses a Corticon Soap Servlet, then the IP of the Client and the Host Name of Server are added to each log statement generated inside the Soap Servlet. Each method in the Soap Servlet has a `Start` and an `End` log statement, as illustrated:

```
Cc|2013-06-24 14:54:35.618|Thread:http-bio-8082-exec-9|Version: 5.3.2.0 -b5059(5.3.5059)
|jomiller|INFO|com.corticon.eclipse.soap.CcServerMessagingAxis|getDecisionServiceNames -
Start :: (HostLocation=192.168.1.7, ClientIP=192.168.1.7) :: Arguments
Cc|2013-06-24 14:54:41.265|Thread:http-bio-8082-exec-9|Version: 5.3.2.0 -b5059(5.3.5059)
|jomiller|INFO|com.corticon.eclipse.soap.CcServerMessagingAxis|getDecisionServiceNames -
End :: (HostLocation=192.168.1.7, ClientIP=192.168.1.7) :: Results [ProcessOrder]
```

Logging History of a Rule Execution

You can see the details of every CorticonRequest and CorticonResponse in the Corticon Log when you set `loglevel=RULETRACE`. This setting should be used with discretion as it will impact performance.

Note: In earlier releases, payloads were only written to log when `loglevel=INFO`. That log level should not be used in production machine as it impacts performance dramatically.

To get the idea of what gets logged, review the following excerpt of a single service test where the `axis.war` is deployed on Tomcat, the `loglevel` is set to `RULETRACE`, and we use an XML payload to make an execution against a Decision Service.

```
Cc|2013-06-24 15:25:14.619|Thread:http-bio-8082-exec-4|Version: 5.3.2.0 -b5059(5.3.5059)
|jomiller|TIMING|com.corticon.eclipse.server.core.impl.CcServerImpl|CcServerImpl.execute(Document, String) -> Start
Cc|2013-06-24 15:25:14.619|Thread:http-bio-8082-exec-4|Version: 5.3.2.0 -b5059(5.3.5059)
|jomiller|RULETRACE|com.corticon.eclipse.server.core.impl.CcServerImpl|CcServer.execute(Document, String) INPUTS:
aDocCorticonRequest = <?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" decisionServiceName="ProcessOrder">
<WorkDocuments> <Order id="Order_id_1"> <dueDate>1/1/2005 12:00:00 AM</dueDate> <total xsi:nil="1" /> <myItems id="Item_id_1">
<price>10.000000</price> <product>Ball</product> <quantity>20</quantity> </myItems> <myItems id="Item_id_2"> <price>20.000000</price>
<product>Racket</product> <quantity>1</quantity> </myItems> <myItems id="Item_id_3"> <price>5.250000</price> <product>Wrist Band</product>
<quantity>2</quantity> </myItems> </Order> <Order id="Order_id_2"> <dueDate>1/1/2006 12:00:00 AM</dueDate> <myItems id="Item_id_4">
<price>1.000000</price> <product>Pencil</product> <quantity>10</quantity> </myItems> <myItems id="Item_id_5"> <price>3.000000</price>
<product>Notebook</product> <quantity>2</quantity> </myItems> <myItems id="Item_id_6"> <price>2.000000</price> <product>Eraser</product>
<quantity>1</quantity> </myItems> </Order> </WorkDocuments> </CorticonRequest>
Cc|2013-06-24 15:25:14.619|Thread:http-bio-8082-exec-4|Version: 5.3.2.0 -b5059(5.3.5059)
|jomiller|RULETRACE|com.corticon.eclipse.server.core.impl.CcServerImpl|CcServer Exection: Execute using DecisionService: ProcessOrder
Cc|2013-06-24 15:25:14.713|Thread:http-bio-8082-exec-4|Version: 5.3.2.0 -b5059(5.3.5059)
|jomiller|RULETRACE|com.corticon.reactor.engine.DataManager|DataManager.initializeDataObjects(...) :: End :: Total Time = 62
Cc|2013-06-24 15:25:14.775|Thread:http-bio-8082-exec-4|Version: 5.3.2.0 -b5059(5.3.5059)
|jomiller|RULETRACE|corticonrules_inmemory.Act_Order_13702924368121| Fire
rule:corticonrules_inmemory.Act_Order_13702924368121_nrule_0:@1372112714713 Fire
rule:corticonrules_inmemory.Act_Order_13702924368121_nrule_2:@1372112714729 Fire
rule:corticonrules_inmemory.Act_Order_13702924368121_nrule_3:@1372112714729 Fire
rule:corticonrules_inmemory.Act_Order_13702924368121_nrule_1:@1372112714775 Fire
rule:corticonrules_inmemory.Act_Order_13702924368121_rule_1:@1372112714775 Fire
rule:corticonrules_inmemory.Act_Order_13702924368121_rule_2:@1372112714775
Cc|2013-06-24 15:25:14.791|Thread:http-bio-8082-exec-4|Version: 5.3.2.0 -b5059(5.3.5059)
|jomiller|RULETRACE|corticonrules_inmemory.Act_Order_13702924368121subflow| Fire
rule:corticonrules_inmemory.Act_Order_13702924368121:@1372112714713 corticonrules_inmemory.Act_Order_13702924368121subflow - total execution
time is 78 milliseconds - Percentage of execution time [100%] from rulesheet corticonrules_inmemory.Act_Order_13702924368121
Cc|2013-06-24 15:25:14.791|Thread:http-bio-8082-exec-4|Version: 5.3.2.0 -b5059(5.3.5059)|jomiller|RULETRACE|corticonrules_inmemory.Act_Order| Fire
rule:corticonrules_inmemory.Act_Order_13702924368121subflow:@1372112714713 corticonrules_inmemory.Act_Order - total execution time is 78
milliseconds - Percentage of execution time [100%] from rulesheet corticonrules_inmemory.Act_Order_13702924368121subflow
Cc|2013-06-24 15:25:14.822|Thread:http-bio-8082-exec-4|Version: 5.3.2.0 -b5059(5.3.5059)
|jomiller|RULETRACE|com.corticon.eclipse.server.core.impl.CcServerImpl|CcServer.execute(Document, String, String)
(DecisionServiceName=ProcessOrder[-1, -1, null], HostLocation=192.168.1.7) -- RESULT CorticonResponseDocument = <?xml version="1.0"
encoding="UTF-8"?> <CorticonResponse xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
decisionServiceName="ProcessOrder"> <WorkDocuments> <Order id="Order_id_1"> <dueDate>1/1/2005 12:00:00 AM</dueDate> <note>This Order was
shipped late</note> <shipped>true</shipped> <shippedOn>2008-12-01T00:00:00.000-08:00</shippedOn> <total>230.500000</total> <myItems
id="Item_id_1"> <price>10.000000</price> <product>Ball</product> <quantity>20</quantity> <subtotal>200.000000</subtotal> </myItems> <myItems
id="Item_id_2"> <price>20.000000</price> <product>Racket</product> <quantity>1</quantity> <subtotal>20.000000</subtotal> </myItems> <myItems
id="Item_id_3"> <price>5.250000</price> <product>Wrist Band</product> <quantity>2</quantity> <subtotal>10.500000</subtotal> </myItems> </Order>
<Order id="Order_id_2"> <dueDate>1/1/2006 12:00:00 AM</dueDate> <note>This Order was shipped late</note> <shipped>true</shipped>
<shippedOn>2008-12-01T00:00:00.000-08:00</shippedOn> <total>18.000000</total> <myItems id="Item_id_4"> <price>1.000000</price>
<product>Pencil</product> <quantity>10</quantity> <subtotal>10.000000</subtotal> </myItems> <myItems id="Item_id_5"> <price>3.000000</price>
<product>Notebook</product> <quantity>2</quantity> <subtotal>6.000000</subtotal> </myItems> <myItems id="Item_id_6"> <price>2.000000</price>
<product>Eraser</product> <quantity>1</quantity> <subtotal>2.000000</subtotal> </myItems> </Order> </WorkDocuments> <Messages version="1.10">
<Message> <severity>Info</severity> <text>The subtotal of line item for Wrist Band is 10.500000.</text> <entityReference href="#Item_id_3" /> </Message>
<Message> <severity>Info</severity> <text>The subtotal of line item for Ball is 200.000000.</text> <entityReference href="#Item_id_1" /> </Message>
<Message> <severity>Info</severity> <text>The subtotal of line item for Racket is 20.000000.</text> <entityReference href="#Item_id_2" /> </Message>
<Message> <severity>Info</severity> <text>The subtotal of line item for Pencil is 10.000000.</text> <entityReference href="#Item_id_4" /> </Message>
<Message> <severity>Info</severity> <text>The subtotal of line item for Notebook is 6.000000.</text> <entityReference href="#Item_id_5" /> </Message>
<Message> <severity>Info</severity> <text>The subtotal of line item for Eraser is 2.000000.</text> <entityReference href="#Item_id_6" /> </Message>
<Message> <severity>Info</severity> <text>The total for the Order is 230.500000.</text> <entityReference href="#Order_id_1" /> </Message> <Message>
<severity>Info</severity> <text>The total for the Order is 18.000000.</text> <entityReference href="#Order_id_2" /> </Message> <Message>
<severity>Warning</severity> <text>This Order was shipped late. Ship date 12/1/2008 12:00:00 AM</text> <entityReference href="#Order_id_1" />
</Message> <Message> <severity>Warning</severity> <text>This Order was shipped late. Ship date 12/1/2008 12:00:00 AM</text> <entityReference
href="#Order_id_2" /> </Message> </Messages> </CorticonResponse>
Cc|2013-06-24 15:25:14.822|Thread:http-bio-8082-exec-4|Version: 5.3.2.0 -b5059(5.3.5059)
|jomiller|TIMING|com.corticon.eclipse.server.core.impl.CcServerImpl|CcServer.execute(Document, String, String) (DecisionServiceName=ProcessOrder[-1, -
1, null], HostLocation=192.168.1.7) -> End
```

Capacity planning

In a given JVM, the Corticon Server and its Decision Services occupy the following amounts of physical memory:

State of Corticon Server	RAM required
Basic *Corticon Server* overhead with no Decision Services (excludes memory footprint of the JVM which varies by JDK version and platform)	25 MB
Load a single Decision Service from the Deployment Descriptor or `addDecisionService()` method API.	~ 5 MB (this is the overhead for the Trade Allocation sample application with seven (7) Rulesheets, twenty-four (24) rules, five (5) associated entities)
Working memory to handle a single CorticonRequest	~ 1 MB (this is the overhead for the Trade Allocation sample application with seven (7) Rulesheets, twenty-four (24) rules, five (5) associated entities (steady-state usage)).

You may reduce the amount of memory required in a large system by dynamically loading and unloading specific Decision Services. This is especially relevant in resource-constrained handheld or laptop scenarios where only a single business transaction occurs at a time. After the first Decision Service is invoked, it is unloaded by the application and the second Decision Service is loaded (and so on). While this will be slower than having all Decision Services always loaded, it can address tight, memory-constrained environments. A compromise alternative would only dynamically load/unload infrequently used Decision Services.

The Java clock

Finally, whenever performance of Java applications needs to be measured in milliseconds, it should be remembered that Java is dependent upon the operating system's internal clock. And not all operating systems track time to equal degrees of granularity. The following excerpt from the Java *JavaDoc* explains:

```
public static long currentTimeMillis()
```

Returns the current time in milliseconds. Note that while the unit of time of the return value is a millisecond, the granularity of the value depends on the underlying operating system and may be larger. For example, many operating systems measure time in units of tens of milliseconds.

See the description of the class `Date` for a discussion of slight discrepancies that may arise between "computer time" and coordinated universal time (UTC).

Returns:

The difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.

12

Internationalization

Corticon Server can accept and generate data values in character sets other than English. This section describes the general capabilities of Corticon Server for use outside the English character set.

- Any attribute of type String can contain any character supported by the UTF-8 encoding standard. This means that characters in European and Asian languages are supported. All encoding of string values passed to Corticon Server is assumed to be UTF-8. Any CorticonResponse outputs, including Messages, will also follow UTF-8 character encoding.
- Vocabulary names (entities, attributes, associations) are restricted to A-Z, a-z, 0-9, and underscore.
- File names and their paths are restricted to A-Z, a-z, 0-9, and underscore.
- All tags in the XML payload must use English characters.
- All Java class and Java property names in any Java payload must follow Java English conventions.

In Corticon Studio, it is possible to use ISO 8859-1 encoding instead of UTF-8 (although this will mean that Asian languages are not supported) by setting this property in `CcStudio.properties`:

```
com.corticon.encoding.standard=ISO-8859-1
```

See the *Corticon Studio: Installation Guide* for more details on enabling localization in Corticon Studio.

A

Service contract and message samples

For details, see the following topics:

Annotated examples of XSD and WSDLS available in the Deployment Console

Section	Type	Level	Style
1	XSD	Vocabulary	Flat
2	XSD	Vocabulary	Hierarchical
3	XSD	Decision Service	Flat
4	XSD	Decision Service	Hierarchical
5	WSDL	Vocabulary	Flat
6	WSDL	Vocabulary	Hierarchical
7	WSDL	Decision Service	Flat
8	WSDL	Decision Service	Hierarchical

1 - Vocabulary-level XML schema, FLAT XML messaging style

This section formally defines and annotates the FLAT Vocabulary-level XSD. Annotations are shown *in this format*, while XML code is shown

```
in this format.
```

Header

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema "
xmlns:tns="urn:<namespace>" targetNamespace="urn:<namespace>"
elementFormDefault="qualified">
```

for details on <namespace> definition, see XML Namespace Mapping

CorticonRequestType and CorticonResponseType

The CorticonRequest element contains the required input to the Decision Service:

```
<xsd:element name="CorticonRequest" type="tns:CorticonRequestType" />
```

The CorticonResponse element contains the output produced by the Decision Service:

```
   <xsd:element name="CorticonResponse" type="tns:CorticonResponseType" />

   <xsd:complexType name="CorticonRequestType">
     <xsd:sequence>
```

Each CorticonRequestType must contain one WorkDocuments element:

```
   <xsd:element name="WorkDocuments" type="tns:WorkDocumentsType" />
      </xsd:sequence>
```

This attribute contains the Decision Service Name. Because a Vocabulary-level service contract can be used for several different Decision Services (provided they all use the same Vocabulary), a Decision Service Name will not be automatically populated here during service contract generation. Your request document must contain a valid Decision Service Name in this attribute, however, so the Server knows which Decision Service to execute…

```
   <xsd:attribute name="decisionServiceName" use="required" type="xsd:string"
/>
```

This attribute contains the Decision Service target version number. While every Decision Service created in Corticon Studio will be assigned a version number (if not manually assigned), it is not necessary to include that version number in the invocation unless you want to invoke a specific version of the named Decision Service.

```
     <xsd:attribute name="decisionServiceTargetVersion" use="optional"
type="xsd:decimal" />
```

This attribute contains the invocation timestamp. Decision Services may be deployed with effective and expiration dates, which allow the Corticon Server to manage multiple versions of the same Decision Service Name and execute the effective version based on the invocation timestamp. It is not necessary to include the invocation unless you want to invoke a specific effective version of the named Decision Service by date (usually past or future).

```
     <xsd:attribute name="decisionServiceEffectiveTimestamp" use="optional"
type="xsd:dateTime" />
 </xsd:complexType>
  <xsd:complexType name="CorticonResponseType">
  <xsd:sequence>
```

Each CorticonResponseType element produced by the Server will contain one WorkDocuments element:

```
   <xsd:element name="WorkDocuments" type="tns:WorkDocumentsType" />
```

Each CorticonResponseType element produced by the Server will contain one Messages element, but if the Decision Service generates no messages, this element will be empty:

```
   <xsd:element name="Messages" type="tns:MessagesType" />
</xsd:sequence>
```

Same as attribute in CorticonRequest. This means that every CorticonResponse will contain the Decision Service Name executed during the transaction.

```
    < xsd:attribute name="decisionServiceName" use="required"
type="xsd:string" />
```

Same as attribute in CorticonRequest.

```
    <xsd:attribute name="decisionServiceTargetVersion" use="optional"
type="xsd:decimal" />
```

Same as attribute in CorticonRequest.

```
  <xsd:attribute name="decisionServiceEffectiveTimestamp" use="optional"
type="xsd:dateTime" />
```

WorkDocumentsType

Entities within `WorkDocumentsType` may be listed in any order.

```
  <xsd:complexType name="WorkDocumentsType">
```

If you plan to use a software tool to read and use a Corticon-generated service contract, be sure it supports this <xsd:choice> tag…

```
  <xsd:choice maxOccurs="unbounded">
```

In a Vocabulary-level XSD, a WorkDocumentsType element contains all of the entities from the Vocabulary file specified in the Deployment Console. One will be a mandatory WDE and all the others will be optional (non-WDE) entities…

The WDE entity is mandatory in message instances that use this service contract. It has the form:

```
    <xsd:element name="VocabularyEntityName"
type="tns:VocabularyEntityNameType" maxOccurs="unbounded" />
```

Non-WDE entities are optional in message instances that use this service contract (minOccurs="0" indicates optional) and have the form:

```
    <xsd:element name="VocabularyEntityName" type="
tns:VocabularyEntityNameType" minOccurs="0" maxOccurs="unbounded" />
  </xsd:choice>
```

This element reflects the FLAT XML Messaging Style selected in the Deployment Console:

```
    <xsd:attribute name="messageType" fixed="FLAT" use="optional" />
     </xsd:complexType>
```

MessagesType

```
<xsd:complexType name="MessagesType">
```

If you plan to use a software tool to read and use a Corticon-generated service contract, be sure it supports this <xsd:sequence> tag (see important note below)…

```
 <xsd:sequence>
```

A Messages element includes zero or more Message elements.

```
    <xsd:element name="Message" type="tns:MessageType" minOccurs="0"
maxOccurs="unbounded" />
  </xsd:sequence>
```

This version number corresponds to the responding Decision Service's version number, which is set in Corticon Studio.

```
   <xsd:attribute name=version type=xsd:string />
     </xsd:complexType>
```

A Message element consists of several items – see the Rule Language Guide for more information on the post operator, which generates the components of a Messages element.

```
  <xsd:complexType name="MessageType">
    <xsd:sequence>
```

These severity levels correspond to those of the posted Rule Statements…

```
<xsd:element name="severity">
       <xsd:simpleType>
          <xsd:restriction base="xsd:string">
             <xsd:enumeration value="Info" />
             <xsd:enumeration value="Warning" />
             <xsd:enumeration value="Violation" />
          </xsd:restriction>
       </xsd:simpleType>
 </xsd:element>
```

The text element corresponds to the text of the posted Rule Statements…

```
  <xsd:element name="text" type="xsd:string" />
  <xsd:element name="entityReference">
        <xsd:complexType>
```

The href association corresponds to the entity references of the posted Rule Statements…

```
            <xsd:attribute name="href" type="xsd:anyURI" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
```

The XML tag `<xsd:sequence>` is used to define the attributes of a given element. In an XML Schema, `<sequence>` requires the elements that follow to appear in exactly the order defined by the schema within the corresponding XML document.

If CcServer.properties `com.corticon.ccserver.ensureComplianceWithServiceContract` is:

- `true`, the Server will return the elements in the same order as specified by the service contract, even for elements created during rule execution and not present in the incoming message.
- `false`, the *`Server`* may return elements in any order. Consuming applications should be designed accordingly. This setting results in slightly better *`Server`* performance.

VocabularyEntityNameType

```
 <xsd:complexType name="VocabularyEntityNameType">
     <xsd:sequence>
```

A VocabularyEntityNameType contains zero or more VocabularyAttributeNames, but any VocabularyAttributeName may appear at most once per VocabularyEntityNameType…

```
     <xsd:element name="VocabularyAttributeName"
type="xsd:VocabularyAttributeNameType" nillable="false" minOccurs="0" />
```

Associations between VocabularyEntityNames are represented as follows. This particular association is optional and has one-to-one or many-to-one cardinality:

```
   <xsd:element name="VocabularyRoleName" type="tns:ExtURIType"
minOccurs="0" />
```

This particular association is optional and has one-to-many or many-to-many cardinality:

```
     <xsd:element name="VocabularyRoleName" type="tns:ExtURIType"
minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
```

Every VocabularyEntityNameType will contain a unique id number – if an id is not included in the CorticonRequest element, the Server will automatically assign one and return it in the CorticonResponse

```
 <xsd:attribute name="id" type="xsd:ID" use="optional" />
```

The ExtURIType is used by all associations in messages having FLAT XML Message Style…

```
     <xsd:complexType name="ExtURIType">
        <xsd:attribute name="href" type="xsd:anyURI" />
      </xsd:complexType>
    </xsd:schema>
```

VocabularyAttributeNameTypes

Every attribute in a *Corticon* Vocabulary has one of five datatypes – Boolean, String, Date, Integer, or Decimal. Thus when entities are passed in a CorticonRequest or CorticonResponse, their attributes must be one of these five types. In addition, the `ExtURIType` type is used to implement associations between entity instances. The `href` attribute in an entity "points" to another entity with which it is associated.

2 - Vocabulary-level XML schema, HIER XML messaging style

This section formally defines and annotates the HIER Vocabulary-level XSD. Most elements are the same or have only minor differences from the FLAT XSD described above.

Header

This section of the XSD is identical to the FLAT version, described in Header on page 160.

CorticonRequestType and CorticonResponseType

This section of the XSD is identical to the FLAT version, described in CorticonRequestType and CorticonResponseType on page 160.

WorkDocumentsType

One line in this section differs from the FLAT version (described in WorkDocumentsType on page 162):

This attribute value indicates the HIER XML Messaging Style selected in the Deployment Console:

```
<xsd:attribute name="messageType" fixed="HIER" use="optional" />
```

MessagesType

This section of the XSD is identical to the FLAT version, described in MessagesType on page 162.

VocabularyAttributeNameTypes

This section of the XSD is the same as the FLAT version, described above.

3 - Decision-service-level XML schema, HIER XML messaging style

When **Decision Service** is selected in section **16** of Figure: Deployment Console with Input Options Numbered, the XML Messaging Style option in section **20** becomes inactive ("grayed out"). This occurs because the XML Messaging Style option at the Decision Service level, (selected in section **15** of Figure: Right Portion of Deployment Console, with Deployment Descriptor File Options Numbered) becomes the governing setting.

Header

This section of the XSD is identical to the Vocabulary-level FLAT version, described in Header on page 160.

CorticonRequestType and CorticonResponseType

This section of the XSD is identical to the Vocabulary-level FLAT version, described in CorticonRequestType and CorticonResponseType on page 160, with the exception of the following lines in each complexType:

```
<xsd:attribute name="decisionServiceName" use="required"
fixed="DecisionServiceName" type="xsd:string" />
```

Notice that the name of the Decision Service you entered in section 2 of Left Portion of Deployment Console, with Deployment Descriptor File Options Numbered is automatically inserted in `fixed="DecisionServiceName"`.

The use of the `fixed` element is optional – in some cases, `fixed` may give XML parsers difficulty. See Extended Service Contracts: Fixed Attribute for details on removing the `fixed` element from your service contracts.

WorkDocumentsType

This section of the XSD is identical to the Vocabulary-level FLAT version, described in WorkDocumentsType on page 165.

MessagesType

This section of the XSD is identical to the Vocabulary-level FLAT version, described in MessagesType on page 162.

VocabularyEntityNameType and VocabularyAttributeNameTypes

The *structure* of this section of the XSD is identical to the Vocabulary-level FLAT version (described here). However, a Decision-Service-level service contract will contain *only* those entities and attributes from the Vocabulary that are actually used by the rules in the Decision Service. This means that a Decision-Service-level contract will typically contain a *subset* of the entities and attributes contained in the Vocabulary-level service contract.

4 - Decision-service-level XML schema, HIER XML messaging style

When **Decision Service** is selected in section **8**, the XML Messaging Style option in section **12** becomes inactive ("grayed out"). This occurs because the XML Messaging Style option at the Decision Service level, selected in **SECTION 6** becomes the governing setting.

Header

This section of the XSD is identical to the Vocabulary-level FLAT version, described in Header on page 160.

CorticonRequestType and CorticonResponseType

This section of the XSD is identical to the Decision-Service-level FLAT version, described in CorticonRequestType and CorticonResponseType on page 160.

WorkDocumentsType

This section of the XSD is identical to the Vocabulary-level HIER version, described in WorkDocumentsType on page 162.

MessagesType

This section of the XSD is identical to the Vocabulary-level FLAT version, described in MessagesType on page 162.

VocabularyEntityNameType and VocabularyAttributeNameTypes

The *structure* of this section of the XSD is identical to the Vocabulary-level HIER version (described above). However, a Decision-Service-level service contract will contain *only* those entities and attributes from the Vocabulary that are actually used by the rules in the Decision Service. This means that a Decision-Service-level service contract will typically contain *some subset* of the entities and attributes contained in the Vocabulary-level service contract.

5 - Vocabulary-level WSDL, FLAT XML messaging style

SOAP Envelope

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="urn:<namespace>" xmlns:cc="urn:<namespace>"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
targetNamespace="urn:<namespace>">
```

for details on <namespace> definition, see XML Namespace Mapping

Types

```
<types>
     <xsd:schema xmlns:tns="urn:Corticon" targetNamespace="urn:<namespace>"
elementFormDefault="qualified">
```

This <type> section contains the entire Vocabulary-level XSD, FLAT-style service contract, minus the XSD Header section…

or details on <namespace> definition, see XML Namespace Mapping

```
</types>
```

Messages

The SOAP service supports two messages, each with a single argument. See portType

```
<message name="CorticonRequestIn">
     <part name="parameters" element="cc:CorticonRequest" />
</message>
<message name="CorticonResponseOut">
     <part name="parameters" element="cc:CorticonResponse" />
</message>
```

PortType

```
<portType name="VocabularyNameDecisionServiceSoap">
```

Indicates service operation: one message in and one message out…

```
    <operation name="processRequest">
       <input message="tns:CorticonRequestIn" />
       <output message="tns:CorticonResponseOut" />
    </operation>
</portType>
```

Binding

Use HTTP transport for SOAP operation defined in <portType>

```
<binding name="VocabularyNameDecisionServiceSoap" type="tns:
VocabularyNameDecisionServiceSoap">
```

All WSDLs generated by the Deployment Console use Document-style messaging:

```
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document" />
     <operation name="processRequest">
```

Identifies the SOAP binding of the Decision Service:

```
      <soap:operation soapAction="urn:Corticon" style="document" />
    <input>
        <soap:body use="literal" namespace="urn:Corticon" />
     </input>
     <output>
        <soap:body use="literal" namespace="urn:Corticon" />
      </output>
    </operation>
  </binding>
```

Service

```
<service name="VocabularyNameDecisionService">
```

*Any text you enter in a Rulesheet's comments window (accessed via **Rulesheet** > **Properties** > **Comments** tab on the Corticon Studio menubar) will be inserted here:*

```
     <documentation>optional Rulesheet comments</documentation>
     <port name="VocabularyNameDecisionServiceSoap"
 binding="tns:VocabularyNameDecisionServiceSoap">
```

Corticon Server Servlet URI contained in section 22 of the Deployment Console will be inserted here:

```
      <soap:address location="http://localhost:8082/axis/services/Corticon"
/>
     </port>
   </service>
</definitions>
```

6 - Vocabulary-level WSDL, HIER XML messaging style

SOAP Envelope

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in SOAP Envelope on page 167.

Types

```
<types>
      <xsd:schema xmlns:tns="urn:<namespace>"
targetNamespace="urn:<namespace>" elementFormDefault="qualified">
```

This <type> section contains the entire Vocabulary-level XSD, FLAT-style service contract, minus the XSD Header section…

or details on <namespace> definition, see XML Namespace Mapping

```
</types>
```

Messages

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in Messages on page 168.

PortType

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in PortType on page 168.

Binding

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in Binding on page 168.

Service

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in Service on page 169.

7 - Decision-service-level WSDL, FLAT XML messaging style

SOAP Envelope

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in SOAP Envelope on page 167.

Types

```
<types>
      <xsd:schema xmlns:tns="urn:<namespace>"
targetNamespace="urn:<namespace>" elementFormDefault="qualified">
```

This <type> section contains the entire Decision Service-level XSD, FLAT-style service contract, minus the XSD Header section…

or details on <namespace> definition, see XML Namespace Mapping

```
</types>
```

Messages

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in Messages on page 168.

PortType

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in PortType on page 168, with the exception of the following line:

```
<portType name="DecisionServiceNameSoap">
```

Binding

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in Binding on page 168, with the exception of the following line:

```
<binding name="DecisionServiceNameSoap" type="tns: DecisionServiceNameSoap">
```

Service

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in Service on page 169, with the exception of the following lines:

```
    <service name="DecisionServiceName">

    <port name="DecisionServiceNameSoap" binding="tns:
DecisionServiceNameSoap">
```

8 - Decision-service-level WSDL, HIER XML messaging style

SOAP Envelope

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in SOAP Envelope on page 167.

Types

```
<types>
     <xsd:schema xmlns:tns="urn:<namespace>" targetNamespace="urn:<namespace>"

elementFormDefault="qualified">
```

This <type> section contains the entire Decision Service-level XSD, HIER-style service contract, minus the XSD Header section…

or details on <namespace> definition, see XML Namespace Mapping

```
</types>
```

Messages

This section of the WSDL is identical to the Decision Service-level FLAT version, described in Messages on page 168.

PortType

This section of the WSDL is identical to the Decision Service-level FLAT version, described in PortType on page 168.

Binding

This section of the WSDL is identical to the Decision Service-level FLAT version, described in Binding on page 168.

Service

This section of the WSDL is identical to the Decision Service-level FLAT version, described in Service on page 169.

Extended service contracts

NewOrModified attribute

Corticon service contract structures may be extended with an optional `newOrModified` attribute that indicates which parts of the payload have been changed by the *Corticon Server* during execution.

```
    <xsd:attribute name="newOrModified" type="xsd:boolean" use="optional" />
</xsd:complexType>
```

Any attribute (the Vocabulary attribute) whose value was changed by the *Corticon Server* during rule execution will have the `newOrModified` attribute set to `true`. Also,

In FLAT messages, the `newOrModified` attribute of an entity is `true` if:

- Any contained attribute is modified.
- Any association to that entity is added or removed.

In HIER messages, the `newOrModified` attribute of an entity is `true` if the entity, *or any of its associated entities*:

- Any contained attribute is modified.
- Any association to that entity is added or removed.

This attribute (XML attribute, not Vocabulary attribute) is enabled and disabled by the `enableNewOrModified` property in `CcCommon.properties` (see Configuring Corticon properties and settings on page 185 for details).

In order to make use of the `newOrModified` attribute, your consuming application must be able to correctly parse the response message. Because this attribute adds additional complexity to the service contract and its resultant request and response messages, be sure your SOAP integration toolset is capable of handling the increased complexity before enabling it.

Extended datatypes

If the `newOrModified` attribute is enabled, then the base XML datatypes must be extended to accommodate it. The following `complexTypes` are included in service contracts that make use of the `newOrModified` attribute.

ExtBooleanType

```
  <xsd:complexType name="ExtBooleanType">
    <xsd:simpleContent>
         <xsd:extension base="xsd:boolean">
          <xsd:attribute name="newOrModified" type="xsd:boolean"
use="optional" />
         </xsd:extension>
       </xsd:simpleContent>
    </xsd:complexType>
```

ExtStringType

```
  <xsd:complexType name="ExtStringType">
       <xsd:simpleContent>
         <xsd:extension base="xsd:string">
           <xsd:attribute name="newOrModified" type="xsd:boolean"
use="optional" />
         </xsd:extension>
       </xsd:simpleContent>
   </xsd:complexType>
```

ExtDateTimeType

```
   <xsd:complexType name="ExtDateTimeType">
        <xsd:simpleContent>
          <xsd:extension base="xsd:dateTime">
            <xsd:attribute name="newOrModified" type="xsd:boolean"
use="optional" />
          </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
```

If CcServer.properties `com.corticon.ccserver.ensureComplianceWithServiceContract` is

- `False`: the Server will return an attribute of type `Date` in the same form as it was received.
- `True`: the Server will return an attribute of type Date using the date mask `yyyy-MM-dd'T'HH\:mm\:ss z`

ExtIntegerType

```
   <xsd:complexType name="ExtIntegerType">
        <xsd:simpleContent>
          <xsd:extension base="xsd:integer">
            <xsd:attribute name="newOrModified""type="xsd:boolean"
use="optional" />
          </xsd:extension>
        </xsd:simpleContent>
     </xsd:complexType>
```

ExtDecimalType

```
  <xsd:complexType name="ExtDecimalType">
      <xsd:simpleContent>
           <xsd:extension base="xsd:decimal">
          <xsd:attribute name="newOrModified"
           type="xsd:boolean" use="optional" />
        </xsd:extension>
      </xsd:simpleContent>
  </xsd:complexType>
```

Fixed attribute

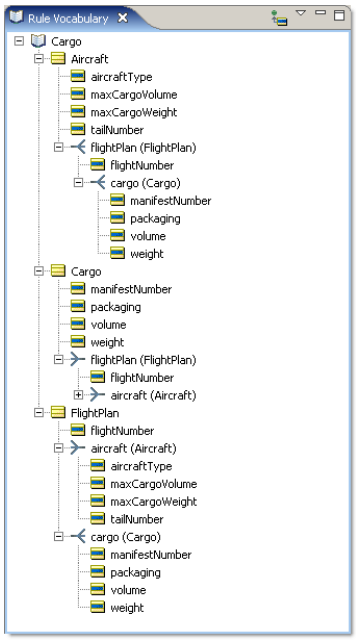
As with the `newOrModified` attribute, some XML or SOAP toolsets have difficulty parsing service contracts and messages which include the `fixed` attribute, seen in the `CorticonRequest` and `CorticonResponse` complexType sections of all service contracts. As of this manual's release date, it is known that versions of Microsoft's .NET SOAP toolkit have difficulty with `fixed`. To improve compatibility, CcDeployment.properties includes a property named `com.corticon.deployment.includeFixedTaskValueInServiceContract`, which if set to `false`, removes the `fixed` attribute from all service contracts generated by the *`Deployment Console`*.

Examples

This section illustrates with an example how the service contract is generated and what the input and output payload looks like.

The example used is from the *Corticon Studio Tutorial: Basic Rule Modeling*. A `FlightPlan` is associated with a `Cargo`. A `FlightPlan` is also associated with an `Aircraft`.

The Vocabulary is shown below.



`FlightPlan` is the Work Document Entity (WDE).

In this section, both WSDL and XML Schema service contracts are shown. Some annotations are provided. The WSDL example uses FLAT XML messaging style; the XML Schema example uses HIER messaging style.

Vocabulary-Level WSDL, FLAT XML Messaging Style

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns=
"http://localhost:8082/axis/services/Corticon/CargoDecisionService"
xmlns:cc= "urn:decision:CargoDecisionService"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" targetNamespace=
"http://localhost:8082/axis/services/Corticon/CargoDecisionService"<types>
    <xsd:schema xmlns:tns= "urn:decision:CargoDecisionService"
targetNamespace= "urn:decision:CargoDecisionService"
elementFormDefault="qualified">
  <xsd:element name="CorticonRequest" type="tns:CorticonRequestType" />

  <xsd:element name="CorticonResponse" type="tns:CorticonResponseType" />

  <xsd:complexType name="CorticonRequestType">
  <xsd:sequence>
    <xsd:element name="WorkDocuments" type="tns:WorkDocumentsType" />

  </xsd:sequence>
  <xsd:attribute name="decisionServiceName" use="required"
type="xsd:string" />
  </xsd:complexType>
  <xsd:complexType name="CorticonResponseType">
    <xsd:sequence>
      <xsd:element name="WorkDocuments" type="tns:WorkDocumentsType" />
```

```
      <xsd:element name="Messages" type="tns:MessagesType" />
    </xsd:sequence>
```

Even though this is a Vocabulary-level WSDL, the Decision Service Name is still required:

```
   <xsd:attribute name="decisionServiceName" use="required"
type="xsd:string" />
  </xsd:complexType>
  <xsd:complexType name="WorkDocumentsType">
    <xsd:choice maxOccurs="unbounded">
```

This is a Vocabulary-level service contract, so all entities in the Vocabulary are included here:

```
  <xsd:element name="Aircraft"
type="tns:AircraftType" minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="Cargo" type="tns:CargoType" minOccurs="0"
maxOccurs="unbounded" />
```

FlightPlan is the WDE, so it is a mandatory element – notice no minOccurs attribute:

```
  <xsd:element name="FlightPlan"
type="tns:FlightPlanType" maxOccurs="unbounded" />
</xsd:choice>
```

FLAT style specified here:

```
        <xsd:attribute name="messageType" fixed="FLAT" use="optional" />
     </xsd:complexType>
        <xsd:element name="Aircraft" type="tns:AircraftType" minOccurs="0"
maxOccurs="unbounded" />
         <xsd:element name="Cargo" type="tns:CargoType"\ minOccurs="0"
maxOccurs="unbounded" />
        <xsd:element name="FlightPlan" type="tns:FlightPlanType" minOccurs="0"

maxOccurs="unbounded" />
          </xsd:choice>
       </xsd:complexType>
      <xsd:complexType name="MessagesType">
        <xsd:sequence>
           <xsd:element name="Message" type="tns:MessageType" minOccurs="0"
maxOccurs="unbounded" />
 </xsd:sequence>
       <xsd:attribute name="version" type="xsd:string" />
      </xsd:complexType>
      <xsd:complexType name="MessageType">
        <xsd:sequence>
          <xsd:element name="severity">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:enumeration value="Info"" />
                <xsd:enumeration value="Warning" />
                <xsd:enumeration value="Violation" />
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="text" type="xsd:string" />
          <xsd:element name="entityReference">
            <xsd:complexType>
              <xsd:attribute name="href" type="xsd:anyURI" />
            </xsd:complexType>
          </xsd:element>
       </xsd:sequence>
</xsd:complexType>
      <xsd:complexType name="AircraftType">
        <xsd:sequence>
          <xsd:element name="aircraftType" type="xsd:string" nillable="false"
```

```
 minOccurs="0" />
         <xsd:element name="maxCargoVolume" type="xsd:decimal" nillable="false"
 minOccurs="0" />
         <xsd:element name="maxCargoWeight" type="xsd:decimal" nillable="false"
 minOccurs="0" />
          <xsd:element name="tailNumber" type="xsd:string" nillable="false"
minOccurs="0" />
          <xsd:element name="flightPlan" type="tns:ExtURIType" minOccurs="0"
maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID" use="optional" />
      </xsd:complexType>
      <xsd:complexType name="CargoType">
        <xsd:sequence>
         <xsd:element name="manifestNumber" type="xsd:string" nillable="false"
 minOccurs="0" />
          <xsd:element name="volume" type="xsd:decimal" nillable="false"
minOccurs="0" />
          <xsd:element name="weight" type="xsd:decimal" nillable="false"
minOccurs="0" />
         <xsd:element name=""flightPlan"" type="tns:"ExtURIType" minOccurs="0"
 />
       </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID" use="optional" />
      </xsd:complexType>
      <xsd:complexType name="FlightPlanType">
       <xsd:sequence>
 <xsd:element name="flightNumber" type="xsd:integer" nillable="false"
minOccurs="0" />
          <xsd:element name="flightRange" type="xsd:integer" nillable="false"
 minOccurs="0" />
```

This is a FLAT-style message, so all associations are represented by the ExtURIType:

```
         <xsd:element name="aircraft" type="tns:ExtURIType" minOccurs="0" />
         <xsd:element name="cargo" type="tns:ExtURIType" minOccurs="0"
maxOccurs="unbounded" />
       </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID" use="optional" />
     </xsd:complexType>
     <xsd:complexType name="ExtURIType">
       <xsd:attribute name="href" type="xsd:anyURI" />
     </xsd:complexType>
    </xsd:schema>
   </types>
   <message name="CorticonRequestIn">
     <part name="parameters" element="cc:CorticonRequest" />
   </message>
   <message name="CorticonResponseOut">
     <part name="parameters" element="cc:CorticonResponse" />
   </message>
   <portType name="CargoDecisionServiceSoap">
     <operation name="processRequest">
      <input message="tns:CorticonRequestIn" />
       <output message="tns:CorticonResponseOut" />
     </operation>
   </portType>
   <binding name="CargoDecisionServiceSoap"
type="tns:CargoDecisionServiceSoap">
```

All Web Services service contracts must be document-style!

```
     <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document" />
     <operation name="processRequest">
       <soap:operation soapAction="urn:Corticon" style="document" />
       <input>
         <soap:body use="literal" namespace="urn:Corticon" />
       </input>
       <output>
         <soap:body use="literal" namespace="urn:Corticon" />
       </output>
     </operation>
   </binding>
   <service name="CargoDecisionService">
     <documentation>InsertDecisionServiceDescription</documentation>
     <port name="CargoDecisionServiceSoap"
binding="tns:CargoDecisionServiceSoap">
       <soap:address location="http://localhost:8082/axis/services/Corticon"
/>
     </port>
 </service>
 </definitions>
```

Decision-Service-Level XSD, HIER XML Messaging Style

```
 <?xml version="1.0" encoding="UTF-8"?>
 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://localhost:8082/axis/services/Corticon/tutorial_example"
 xmlns:cc="urn:decision:tutorial_example"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

targetNamespace="http://localhost:8082/axis/services/Corticon/tutorial_example">

   <types>
     <xsd:schema xmlns:tns="urn:decision:tutorial_example"
 targetNamespace="urn:decision:tutorial_example"
 elementFormDefault="qualified">
       <xsd:element name="CorticonRequest" type="tns:CorticonRequestType" />
       <xsd:element name="CorticonResponse" type="tns:CorticonResponseType"
/>
       <xsd:complexType name="CorticonRequestType">
         <xsd:sequence>
           <xsd:element name="WorkDocuments" type="tns:WorkDocumentsType" />
         </xsd:sequence>
```

The Decision Service Name has been automatically included here:

```
   <xsd:attribute name="decisionServiceName" use="required"
fixed="tutorial_example" type="xsd:string" />
         <xsd:attribute name="decisionServiceTargetVersion" use="optional"
type="xsd:decimal" />
         <xsd:attribute name="decisionServiceEffectiveTimestamp"
use="optional" type="xsd:dateTime" />
       </xsd:complexType>
       <xsd:complexType name="CorticonResponseType">
         <xsd:sequence>
           <xsd:element name="WorkDocuments" type="tns:WorkDocumentsType" />
           <xsd:element name="Messages" type="tns:MessagesType" />
         </xsd:sequence>
```

The Decision Service Name has been automatically included here:

```
      <xsd:attribute name="decisionServiceName" use="required"
fixed="tutorial_example" type="xsd:string" />
        <xsd:attribute name="decisionServiceTargetVersion"
use="optional" type="xsd:decimal" />
        <xsd:attribute name="decisionServiceEffectiveTimestamp"
use="optional" type="xsd:dateTime" />
      </xsd:complexType>
      <xsd:complexType name="WorkDocumentsType">
        <xsd:choice maxOccurs="unbounded">
          <xsd:element name="Cargo" type="tns:CargoType" />
        </xsd:choice>
```

Cargo is the WDE, so it is mandatory – notice no minOccurs attribute is present.

HIER message style:

```
      <xsd:attribute name="messageType" fixed="HIER" use="optional" />
      </xsd:complexType>
      <xsd:complexType name="MessagesType">
        <xsd:sequence>
          <xsd:element name="Message" type="tns:MessageType" minOccurs="0"
maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="version" type="xsd:string" />
      </xsd:complexType>
      <xsd:complexType name="MessageType">
        <xsd:sequence>
          <xsd:element name="severity">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:enumeration value="Info" />
                <xsd:enumeration value="Warning" />
                <xsd:enumeration value="Violation" />
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="text" type="xsd:string" />
          <xsd:element name="entityReference">
             <xsd:complexType>
              <xsd:attribute name="href" type="xsd:anyURI" />
             </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="CargoType">
        <xsd:sequence>
          <xsd:element name="container" type="xsd:string" nillable="false"
minOccurs="0" />
          <xsd:element name="needsRefrigeration" type="xsd:boolean"
nillable="true"
minOccurs="0" />
          <xsd:element name="volume" type="xsd:long" nillable=""false"
minOccurs="0" />
          <xsd:element name="weight" type="xsd:long" nillable=""false"
minOccurs="0" />
        </xsd:sequence>
       <xsd:attribute name="id" type="xsd:ID" use="optional" />
       <xsd:attribute name="href" type="xsd:anyURI" use=""optional" />
     </xsd:complexType>
    </xsd:schema>
  </types>
  <message name="CorticonRequestIn">
    <part name="parameters" element="cc:CorticonRequest" />
</message>
  <message name="CorticonResponseOut">
    <part name="parameters" element="cc:CorticonResponse" />
</message>
```

```
  <portType name="tutorial_exampleSoap">
<operation name="processRequest">
      <input message="tns:CorticonRequestIn" />
     <output message="tns:CorticonResponseOut" />
    </operation>
  </portType>
  <binding name="tutorial_exampleSoap" type="tns:tutorial_exampleSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
 style="document" />
    <operation name="processRequest">
      <soap:operation soapAction="urn:Corticon" style="document" />
      <input>
       <soap:body use="literal" namespace="urn:Corticon" />
      </input>
      <output>
        <soap:body use="literal" namespace="urn:Corticon" />
      </output>
    </operation>
  </binding>
  <service name="tutorial_example">
    <documentation />
    <port name="tutorial_exampleSoap" binding="tns:tutorial_exampleSoap">
      <soap:address location="http://localhost:8082/axis/services/Corticon"
/>
    </port>
  </service>
</definitions>
```

Sample CorticonRequest Content

A sample `CorticonRequest` payload is shown below. It is a Decision-Service-level message which means that only those Vocabulary terms used in the Decision Service are contained in the `CorticonRequest`. It is also HIER XML messaging style.

Notice the Decision Service Name in the CorticonRequest:

```
<CorticonRequest xmlns=" urn:decision:tutorial_example "
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
decisionServiceName="tutorial_example">
```

Notice the unique ids for every entity. If not provided by the client, Corticon Server will add them automatically to ensure uniqueness:

```
<WorkDocuments>
     <Cargo id="Cargo_id_1">
```

Attribute data is inserted as follows:

```
          <volume>40</volume>
       <weight>16000</weight>
     </Cargo>
  </WorkDocuments>
</CorticonRequest>
```

Sample CorticonResponse Content

Notice the Decision Service Name in the CorticonResponse – this informs the consuming application (which may be consuming several Decision Services asynchronously) which Decision Service is responding in this message:

```
<CorticonResponse decisionServiceName="tutorial_example"
xmlns="urn:Corticon" xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance">
   <WorkDocuments>
     <Cargo id="Cargo_id_1">
       <volume>40.000000</volume>
       <weight>16000.000000</weight
```

Notice that the optional newOrModified attribute has been set to true, indicating that container was modified by the Corticon Server. The value of container, oversize, is the new data derived by the Decision Service.

```
<container newOrModified="true">oversize</container>
     </Cargo>
  </WorkDocuments>
</CorticonResponse>
```

The data contained in the CorticonRequest is returned in the CorticonResponse:

```
         <volume>400.000000</volume>
         <weight>160000.000000</weight>
       </cargo>
    </FlightPlan>
  </WorkDocuments>
<Messages version="1">
```

Notice the message generated and returned by the Server:

```
<Message>
       <severity>Info</severity>
       <text>Cargo weighing between 150,000 and 200,000 lbs must be carried
            by a 747.</text>
```

The entityReference contains an href that associates this message with the FlightPlan that caused it to be produced

```
      <entityReference href="#FlightPlan_id_1"/>
     </Message>
   </Messages>
 </CorticonResponse>
```

B

API summary

The Corticon API set is fully defined in the *JavaDoc* provided separately from the installation kit.

Important: The *JavaDoc* is the official documentation of the API set. It may be updated from time to time in point releases. Readers are encouraged to consult the *JavaDoc* for the latest details.

C

Configuring Corticon properties and settings

Corticon uses property files to control most user-configurable behaviors in Corticon Studio and Corticon Server. All property files are essentially just text files that are automatically read when Corticon Studio and Corticon Server are started, even though they are typically packaged into JAR archives.

The property files are loaded in a specific sequence as listed:

Property File Name	Usage
CcCommon.properties	Modifies the behavior of elements common to both the Corticon Studio and Corticon Server .
CcStudio.properties	Controls behaviors of Corticon Studio functions.
CcServer.properties	Controls behaviors of Corticon Server functions.
CCDeployment.properties	Controls behaviors of Corticon Deployment Console functions.
`CcOem.properties`	Reserved for licensed OEM customers to override vendor-specific properties.
`CcDebug.properties`	Reserved for internal use.
`brms.properties`	A user created text file that provides name-value pairs that override specified default

Note: Once an instance of Corticon Server is running, some properties can be modified via API method calls, as discussed in the Administrative API section.

Properties used by Corticon Studios

Default property settings are built-in to Corticon Studio. See the Common and Studio properties topics for details. You are encouraged to create and maintain a `brms.properties` to add your overrides.

Properties used by Corticon Servers

Corticon Server Property files are collected into a single file named `CcConfig.jar`, located in `[CORTICON_HOME]\Server\lib\CcConfig.jar`. The properties within these files can be changed by editing them directly with a text editor, or, in some cases, with API calls (see the API reference in API summary on page 183 of this document). When the files themselves are changed, the settings remain in effect between Corticon Server sessions. When property settings are changed using APIs, the changes only remain in effect for that Corticon Server session. When Corticon Server starts a new session, it will use the settings in `CcConfig.jar`.

Properties used by Deployment Console

The Deployment Console, installed with each of the Corticon Servers, uses its own set of properties.

For details, see the following topics:

- Specifying custom Business Rules Management System properties
- Properties often set in brms.properties to override defaults
- Common properties (CcCommon.properties)
- Corticon Studio properties (CcStudio.properties)
- Corticon Server properties (CcServer.properties)
- Corticon Deployment Console properties (CcDeployment.properties)

Specifying custom Business Rules Management System properties

The override file, `brms.properties`, lets you specify properties you want to modify without dealing with the mechanics of maintaining a file in a JAR, as well as ensuring that you can revert to the original behavior by just removing your custom properties file.

In the following sections, list your preferred changes as `name=value` pairs in a text file, and then save it as `brms.properties`.

For Studios, copy the file to the folder:

- Studio for Analysts: `[CORTICON_HOME]\Studio for Analysts\configuration\com.corticon.brms`
- Studio: `[CORTICON_HOME]\Studio\eclipse\configuration\com.corticon.brms`

For Servers, add the file into the `CcConfig.jar` appropriate for your deployment strategy.

When Corticon Studio starts up, it will read your properties file last, and thus override corresponding default settings with your settings.

Note: Property settings you list in your `brms.properties` *replace* corresponding properties that have default settings. They do not *append* to an existing list. For example, if you want to add a new `DateTime` mask to the built-in list, be sure to include *all* the masks you intend to use, not just the new one. If your `brms.properties` file contains only the new mask, then it will be the only mask Corticon uses.

Be sure to shut down or exit Corticon Studio or Corticon Server before changing properties, and then restart for the changes to take effect. The Properties in these files are described in detail in the following topics.

Properties often set in brms.properties to override defaults

The properties that users often want to change are:

Log Location

```
logpath=%CORTICON_WORK_DIR%/logs
```

Set in CcCommon.properties. This path determines where Corticon Server log file is stored. In a default Windows installation, the, `%CORTICON_WORK_DIR%` is `C:\Users\{username}\Progress\CorticonWork_5.3`.

Log Level

```
loglevel=VIOLATION
```

Set in CcCommon.properties. This property controls the amount of information included in *`Corticon Server`*'s log file. Options:

- `VIOLATION` – Only Exceptions are logged. Recommended for deployment because it maximizes Server performance. This is the default setting.
- `INFO` – All messages except debugging messages are logged. Never deploy Corticon Server in production at the `INFO` log level. Log files created in this mode may become very large and slow down Corticon Server's performance dramatically.
- `RULETRACE` – Logs rule firings and performance data. Do not deploy Corticon Server in production at the `RULETRACE` log level. While log files are not nearly as large as those generated at the `DEBUG` or `INFO` levels, they still require time to write, and therefore slow Corticon Server's performance.
- `DEBUG` – To be used only at the direction of Progress technical support. Never deploy Corticon Server in production with at the `DEBUG` log level. Log files created in this mode are very detailed; they impact performance and threaten disk overflow.

Automatic Decision Service Reload

Set in CcCommon.properties. Corticon Server runs a background maintenance thread which checks for changes to deployed Decision Services or the Deployment Descriptor files that control their deployment settings. For each Decision Service whose Dynamic Reload deployment setting is **Yes**, the maintenance thread will periodically inspect that *Ruleflow*'s timestamp and determine if a newer version is available (based on the last-saved timestamp assigned by the OS). If it is, then the maintenance thread will flush idle Reactors from Corticon Server's pool and reload it with updated Reactors.

The maintenance thread checks for:

- Changes to any `.cdd` loaded to Corticon Server by a `loadFromCdd` or `loadFromCddDir` API call, including new or removed Decision Services and changes to settings of deployed Decision Services.
- Changes to any `.cdd` file (including new `.cdd's` within the Deployment Descriptor directory) by a `loadFromCddDir` API call, including new or removed Decision Services and changes to settings of deployed Decision Services.
- Changes in any of the Decision Services loaded to Corticon Server. This is done through a compilation timestamp check.

This maintenance thread is enabled by default in CcServer.properties, but it can be disabled using

```
com.corticon.ccserver.dynamicupdatemonitor.autoactivate=false
```

The periodicity of the maintenance thread's inspection is controlled in CcServer.properties by:

```
com.corticon.ccserver.serviceIntervals=30000
```

The maintenance thread can be shutdown and restarted dynamically using the Administrative APIs:

- `CcServerAdminInterface.stopDynamicaUpdateMonitoringService()`
- `CcServerAdminInterface.startDynamicUpdateMonitoringService()`

Common properties (CcCommon.properties)

The following properties are used by both Corticon Studio and Corticon Server:

```
logpath=%CORTICON_HOME%/Log
logverbosity=VERBOSE
loglevel=VIOLATION
com.corticon.logging.thirdparty.logger.class=
```

These four properties deal with logging performed by Corticon Server. `logpath` assigns the directory to which logs are written and saved. By default the value of

```
%CORTICON_HOME%=[CORTICON_HOME];
```

`logverbosity` options:

- `VERBOSE`: Logging is on. Logged Exceptions include stack trace. Default.
- `BRIEF`: Logging is on. Logged Exceptions do NOT include stack trace.
- `SILENT`: Turns off all logging. Maximizes Corticon Server performance.

`loglevel` options:

- `DEBUG`: All messages are logged, including debugging info. To be used only at the direction of Progress technical support. Never deploy Corticon Server in production with `loglevel` set to `DEBUG`. Log files created in this mode may become very large and slow down Corticon Server's performance by orders of magnitude.
- `INFO`: All messages except debugging messages are logged. Never deploy Corticon Server in production with `loglevel` set to `INFO`: Log files created in this mode may become very large and slow down Corticon Server's performance by orders of magnitude.
- `RULETRACE`: Logs performance statistics (such as total time for rule execution, percentage of time on each Rulesheet) and information on what specific rules were fired (not just evaluated). Do not deploy Corticon Server in production with `loglevel` set to `RULETRACE`. While log files are not nearly as large as those generated in `DEBUG` or `INFO` modes, they still require time to write, and therefore slow Corticon Server performance.
- `VIOLATION`: Only Exceptions are logged. Recommended for deployment because it maximizes Corticon Server performance. Default.

`com.corticon.logging.thirdparty.logger.class` Enables a third-party logger to be used for all logging. The new Logger would need to implement the `com.corticon.log.ICcThirdPartyLogger` Interface. If the property is empty, then logging will default back to the Corticon Logger. To use Corticon's Log4JLogger, specify the following logger class for this property: `com.corticon.log.CcLog4JLogger`

```
decimalscale= 6
```

This property handles the default precision for Decimal values in Corticon Studio and Corticon Server . All Decimal values are rounded to the specified number of places to the right of the decimal point. Default is `6` (for example, 4.6056127 will be rounded, displayed, and/or returned as 4.605613).

```
enableNewOrModified= false
```

Determines whether XML responses from Corticon Server include `newOrModified` attributes indicating which elements of the XML document are new or have been modified. This flag also impacts the generation of service contracts (XSD, WSDL). Setting the flag to `false` results in more mainstream XML messaging without the `newOrModified` attributes. Default value is `false`.

```
com.corticon.ccserver.ensureComplianceWithServiceContract= true
```

Determines whether the returning XML CorticonResponse document must be valid with respect to the generated XSD/WSDL file. Ensuring compliance may require dynamic sorting which, if necessary, will slow performance. Default value is `true` (ensure compliance and perform the sorting, if necessary).

```
com.corticon.crml.OclDate.defaultDateForTimeValues=1970-01-01
com.corticon.crml.OclDate.defaultDateFormatForTimeValues= yyyy-MM-dd
```

Determines the "Default Date" to be used when instantiating or converting to Time data values. It is important that this property matches the database date and date format so that there is consistency between Time values inserted into the database directly and those inserted into the database by rules. Default Date value: `1970-01-01`. Default DateFormat value: `yyyy-MM-dd`

```
com.corticon.jdom.translation.textmode= NORMALIZE
```

Determines the type of Corticon translation from JDOM to String. Different settings will yield different results.

- `NORMALIZE`: Mode for text normalization (left and right trim plus internal whitespace is normalized to a single space.
- `TRIM_FULL_WHITE`: Mode for text trimming of content consisting of nothing but whitespace but otherwise not changing output.
- `TRIM`: Mode for text trimming (left and right trim).
- `PRESERVE`: Mode for literal text preservation.

Default is `NORMALIZE`

```
com.corticon.validate.on.load= true
```

Determines whether the Foundation APIs will perform automatic validation of assets when they are loaded. API-controlled validation can help improve performance by validating assets only when necessary. If this flag is set to true, the APIs will validate assets at load time if:

- New validation rules have been added to the APIs.
- Related assets have been changed in a manner that justifies revalidation.

For example, if a *`Rulesheet`*'s Vocabulary has been changed, the API will automatically revalidate the *`Rulesheet`* to ensure that all *`Rulesheet`* expressions are valid with respect to the Vocabulary changes.

If this flag is set to false, the APIs will not perform any validation when the asset is loaded; thus, the GUI is required to explicitly call the `validate` API during editor initialization. Default is `true`.

```
com.corticon.validate.on.activation= false
```

Determines whether the Foundation APIs will defer interdependency validation until asset activation. Normally, the APIs immediately revalidate assets within the editing domain as related assets are modified; however, when a large number of interrelated assets are open simultaneously, this instantaneous revalidation may degrade performance.

If this flag is set to true, the APIs will defer revalidation of assets until they are explicitly "activated" via method `IModelAPI.activate()`, typically when a GUI editor gains the focus. The activate method causes the API to trigger deferred validations if any are pending due to changes in related assets. Default is `true`.

```
com.corticon.localization.setsupportedlocales.swap= true
```

Determines whether foundation API method `setSupportedLocales` will automatically rearrange localizations, potentially changing the base locale of the asset.

The default setting (`true`) will cause the API to automatically swap localized values into the base slot if the base locale in the input array is different from the asset base language. This allows the client program to designate what was originally an added (non-base) locale as the new base locale of the asset; however, this setting also imposes a precondition: when setSupportedLocales is called, the asset must contain complete localizations for every localizable element, or the API will throw an exception. This precondition is imposed because the API contract always requires a base value for every localizable element; while localizations are optional, a base value must never be null.

If this flag is set to `false`, the system will allow the client program to indiscriminately change the set of supported locales without preconditions. In this mode, the system will arbitrarily update the asset's language legend and will remove any localizations while leaving the base values unchanged. While this ensures that API contract is not violated (because the base values remain unaffected), it puts the onus on the client program to "manually" update all base values and localizations to match the specified locale array; failure to do so may leave the language legend and localizations out of synch. Default is `true`.

```
com.corticon.vocabulary.cache= true
```

Determines whether the Vocabulary API will use a hash map to speed up Vocabulary element lookups. This can improve Rulesheet parsing performance, particularly for applications with larger Vocabularies. Default is `true`.

```
com.corticon.validation.checksum.propertylist= com.corticon.crml.OclDate.date
format;com.corticon.crml.OclDate.datetimeformat;com.corticon.crml.OclDate.ti
meformat;com.corticon.crml.OclDate.permissive;com.corticon.crml.OclDate.mask
literals;decimalscale;com.corticon.crml.OclDate.defaultDateForTimeValues;com
.corticon.crml.OclDate.defaultDateFormatForTimeValues;com.corticon.validate.
on.load;com.corticon.validate.on.activation;com.corticon.localization.setsup
portedlocales.swap
```

This is a list of Corticon Properties that will be used in the Checksum Calculations during the Post Load of the Models. This will help determine if we need to revalidate the Model.

```
com.corticon.migration.vocabulary.mandatory.flag= false
```

In this release, the value of an attribute's mandatory property has an impact on engine behavior at runtime, whereas in v4 this was not the case. As a result, migration of v4 assets to this version may result in rule execution behavior that is inconsistent with v4. This property can be used to control this behavior. Here are the allowed values:

- `preserve`: Preserves mandatory property values in migrated asset
- `true`: Unconditionally set all mandatory property values to true
- `false`: Unconditionally set all mandatory property values to false

Default value is `false`.

```
com.corticon.reactor.rulebuilder.maxloops= 100
```

Set max loop interation. Determines what constitutes an endless loop. For `.ers` files with the **Process Logical Loops** setting on, it is necessary to have a safety net to prevent endless loops. This is done by designating the maximum number of iterations allowed for any loop. Default is `100`.

```
com.corticon.reactor.rulebuilder.exception= raise
```

Set maxloop exception handling {raise, bury}. Specifies whether the rule engine will raise a MaxLoopsExceededException if the maximum number of loop iterations is exceeded. Default value is `raise`.

```
com.corticon.reactor.engine.CheckForAssociationDuplicates= true
```

Specifies whether the rule engine conducts an integrity check when adding to an existing association. This integrity check ensures that rules do not add redundant associations between the same two entities. Although, this is a rare that occurrence, it is possible. The downside of this integrity check is that Decision Services that create a significant number of new associations can experience a performance degradation. Such Decision Services would require this configuration property to be set to false. Default value is `true`.

Date/time formats in CcCommon.properties

Corticon Studio uses the `DateTime` datatype to contain both data and time data. The `Date` datatype handles only date information, and the `Time` datatype handles only time information.

It is also noteworthy that the Corticon XML Translator will maintain the consistency of `DateTime`, `Date`, and `Time` values from input to output documents so long as the masks used are contained in the lists.

The first entry for each `dateformat`, `datetimeformat`, and `timeformat` is the default mask. For example, the built-in operator `today` always returns the current date in the default `dateformat` mask. The function `now` returns the current date in the default `datetimeformat`.The entries can be altered but must conform to the patterns/masks supported by the Java class `SimpleDateFormat` in the `java.text` package.

```
com.corticon.crml.OclDate.dateformat=
MM/dd/yy
MM/dd/yyyy
M/d/yy
M/d/yyyy
yyyy/MM/dd
yyyy-MM-dd
yyyy/M/d
yy/MM/dd
yy/M/d
MMM d, yyyy
MMMMM d, yyyy

com.corticon.crml.OclDate.datetimeformat=
MM/dd/yy h:mm:ss a;
MM/dd/yyyy h:mm:ss a;
M/d/yy h:mm:ss a;
```

```
M/d/yyyy h:mm:ss a;
yyyy/MM/dd h:mm:ss a;
yyyy/M/d h:mm:ss a;
yy/MM/dd h:mm:ss a;
yy/M/d h:mm:ss a;
MMM d, yyyy h:mm:ss a;
MMMMM d, yyyy h:mm:ss a;

MM/dd/yy H:mm:ss;
MM/dd/yyyy H:mm:ss;
M/d/yy H:mm:ss;
M/d/yyyy H:mm:ss;
yyyy/MM/dd H:mm:ss;
yyyy/M/d H:mm:ss;
yy/MM/dd H:mm:ss;
yy/M/d H:mm:ss;
MMM d, yyyy H:mm:ss;
MMMMM d, yyyy H:mm:ss;

MM/dd/yy hh:mm:ss a;
MM/dd/yyyy hh:mm:ss a;
M/d/yy hh:mm:ss a;
M/d/yyyy hh:mm:ss a;
yyyy/MM/dd hh:mm:ss a;
yyyy/M/d hh:mm:ss a;
yy/MM/dd hh:mm:ss a;
yy/M/d hh:mm:ss a;
MMM d, yyyy hh:mm:ss a;
MMMMM d, yyyy hh:mm:ss a;

MM/dd/yy HH:mm:ss;
MM/dd/yyyy HH:mm:ss;
M/d/yy HH:mm:ss;
M/d/yyyy HH:mm:ss;
yyyy/MM/dd HH:mm:ss;
yyyy/M/d HH:mm:ss;
yy/MM/dd HH:mm:ss;
yy/M/d HH:mm:ss;
MMM d, yyyy HH:mm:ss;
MMMMM d, yyyy HH:mm:ss;

MM/dd/yy h:mm:ss a z;
MM/dd/yyyy h:mm:ss a z;
M/d/yy h:mm:ss a z;
M/d/yyyy h:mm:ss a z;
yyyy/MM/dd h:mm:ss a z;
yyyy/M/d h:mm:ss a z;
yy/MM/dd h:mm:ss a z;
yy/M/d h:mm:ss a z;
MMM d, yyyy h:mm:ss a z;
MMMMM d, yyyy h:mm:ss a z;

MM/dd/yy H:mm:ss z;
MM/dd/yyyy H:mm:ss z;
M/d/yy H:mm:ss z;
M/d/yyyy H:mm:ss z;
yyyy/MM/dd H:mm:ss z;
yyyy/M/d H:mm:ss z;
yy/MM/dd H:mm:ss z;
yy/M/d H:mm:ss z;
MMM d, yyyy H:mm:ss z;
MMMMM d, yyyy H:mm:ss z;

MM/dd/yy hh:mm:ss a z;
MM/dd/yyyy hh:mm:ss a z;
M/d/yy hh:mm:ss a z;
M/d/yyyy hh:mm:ss a z;
yyyy/MM/dd hh:mm:ss a z;
yyyy/M/d hh:mm:ss a z;
```

```
yy/MM/dd hh:mm:ss a z;
yy/M/d hh:mm:ss a z;
MMM d, yyyy hh:mm:ss a z;
MMMMM d, yyyy hh:mm:ss a z;

MM/dd/yy HH:mm:ss z;
MM/dd/yyyy HH:mm:ss z;
M/d/yy HH:mm:ss z;
M/d/yyyy HH:mm:ss z;
yyyy/MM/dd HH:mm:ss z;
yyyy/M/d HH:mm:ss z;
yy/MM/dd HH:mm:ss z;
yy/M/d HH:mm:ss z;
MMM d, yyyy HH:mm:ss z;
MMMMM d, yyyy HH:mm:ss z

com.corticon.crml.OclDate.timeformat=
h:mm:ss a
h:mm:ss a z
H:mm:ss
H:mm:ss z
hh:mm:ss a
hh:mm:ss a z
HH:mm:ss
HH:mm:ss z

------------------------------------------------------------
com.corticon.crml.OclDate.permissive =true
```

If `permissive` is true (default), then the Corticon date/time parser will be lenient when handling incoming or entered date/times, trying to find a match even if the pattern is not contained in the mask lists. If false, then any incoming or entered date/time must strictly adhere to the patterns defined by `dateformat`, `datetimeformat`, `timeformat`.

Default patterns are for United States and other countries that follow the US conventions on date/times.

```
com.corticon.crml.OclDate.maskliterals =true
```

If `maskliterals` is true (default), the system will parse strings and dates more quickly by checking for the presence of mask literals (for example, "/", "-", ":" or ",") before consulting the date masks (an expensive process). If a string does not contain any of the mask literal characters, it can be immediately deemed a string (as opposed to a date).

To take advantage of this feature, all user-specified date masks must contain at least one literal character. If any user-specified masks contain exclusively date pattern characters (for example, "MMddyy"), `maskliterals` must be set to false in order to prevent the system from misinterpreting date literals (for example, '123199') as simple strings.

These properties deal with the way Corticon Studio and Corticon Server handle date/time formats. Preset formats, or "masks" are used to:

- Process incoming date/times on request XML payloads.
- Insert date/times into output response XML payloads.
- Parse entries made in the Corticon Studio Rulesheets, Vocabulary, and Tests.
- To display any date/time in Corticon Studio.

Masks are divided into 3 categories: `dateformat`, `datetimeformat`, `timeformat`.

Use the following chart to decode the date mask formats:

The following symbols are used in date/time masks:

Symbol	Meaning	Presentation	Patterns
G	Era Designator	Text	G = {AD, BC}
y	Year	Number	yy = {00..99} yyyy = {0000..9999}
M	Month of the year	Text or Number	M = {1..12} MM = {01..12} MMM = {Jan..Dec} MMMM = {January..December}
d	day of the month	Number	d = {1..31} dd = {01..31}
h	hour in AM or PM	Number	h = {1..12} hh = {01..12}
H	hour in 24-hour format (0-23)	Number	H = {0..23} HH = {00..23}
m	minute of the hour	Number	m = {0..59} mm = {00..59}
s	second of the minute	Number	s = {0..59} ss = {00..59}
S	millisecond of the minute	Number	S = {0..999} SSS = {000..999}
E	day of the week	Text	E, EE, or EEE = {Sun..Sat} EEEE = {Sunday..Saturday}
D	day of the year	Number	D = {0..366} DDD = {000..366}
F	day of week in the month	Number	F = {0..6}
w	week of the year	Number	w = {1..53} ww = {01..53}

Symbol	Meaning	Presentation	Patterns
W	week of the month	Number	W = {1..6}
a	AM/PM marker	Text	a = {AM, PM}
k	hour of the day (1-24)	Number	k = {1..24} kk = {01..24}
K	hour in AM/PM	Number	K = {1..12} KK = {01..12}
z	time zone	Text	z, zz, or zzz = abbreviated time zone zzzz = full time zone
`	escape character used to insert text	Delimiter	
'	single quote	Literal	‘

Any characters in the pattern that are not in the ranges of [a..z] and [A..Z] will be treated as quoted text. For instance, characters like {:, ., <space>, #, @} will appear in the resulting time text even they are not embraced within single quotes. A pattern containing any invalid pattern letter will result in a thrown exception during formatting or parsing.

Examples:

Sample Pattern	Resulting Formatted Date
yyyy.MM.dd G 'at' hh:mm:ss z	2013.07.10 AD at 15:08:56 PDT
EEE, MMM d, ''yy	Wed, Jul 10, '13
h:mm a	12:08 PM
hh 'o''clock' a, zzzz	12 o'clock PM, Pacific Daylight Time
K:mm a, z	0:00 PM, PST
yyyy.MMMM.dd G h:mm a	2013.July.10 AD 12:08 PM

Corticon Studio properties (CcStudio.properties)

The following properties are used by the Corticon Studio:

```
com.corticon.eclipse.tester.message.use.swt.table=true
```

Not user configurable.

```
com.corticon.designer.undoredo.stack.size=3
```

Specifies the size of the undo/redo stack. This number corresponds to the number of undo/redo operations the system will permit. Default is `3`.

```
com.corticon.designer.corticon.insertrowstoend=10
```

Determines the number of rows that are added to the end of a Rulesheet section when **Rulesheet** > **Add Rows to End** is selected from the Corticon Studio menubar or popup menu. Default is `10`.

```
com.corticon.designer.corticon.insertcolumnstoend=10
```

Determines the number of columns that are added to the end of a Rulesheet section when **Rulesheet** > **Add Columns to End** is selected from the Corticon Studio menubar or popup menu. Default is `10`.

```
fileHistoryCount=10
```

Determines the number of opened files that are persisted in the file history list. This determines the maximum number of files that appear when **File** > **Reopen** is selected from the Corticon Studio menubar. Default value is `10`.

```
dir.uri.Report=%corticonbase%/Report
```

Determines the directory where the XML-based reporting files (XSL, XSD, HTML, XML) are located. By default, the normal value of `%corticonbase%` is `[CORTICON_HOME]/Report`

```
com.corticon.designer.vocabulary.autofill.objectPropertyAutoFillCollectionType=Vector
com.corticon.designer.vocabulary.autofill.useIsMethodForBoolean=false
```

Determines the parameters used by by Vocabulary|Auto-Fill Object Properties function. `objectPropertyAutoFillCollectionType` determines Object Collection Type when a x-to-many association end is encountered. There are three possible values:

- `Vector` for java.util.Vector
- `Array` for Java arrays (for example, A[])
- `ArrayList` for java.util.ArrayList

If `useIsMethodForBoolean` is `false` then the system generates a getXYZ for a property of type Boolean. If it is true then the system generates a isXYZ in the Getter field for a property of type Boolean. Note: the Setter is not impacted by this flag.

Default value for `objectPropertyAutoFillCollectionType` is `Vector`

Default value for `useIsMethodForBoolean` is `false`

```
com.corticon.encoding.standard=UTF-8
```

Default character encoding for Corticon Studio objects, such as Vocabulary, *`Rulesheet`* and *`Ruletest`* XML files. Examples: UTF-8, UTF-16, ISO-8859-1, US-ASCII. Default value is `UTF-8`.

```
com.corticon.designer.useTextualRulesheetTestsheetStatusIndicator=false
```

Determines whether `[Invalid]` and `[Disable]` text will be added to *`Rulesheet`* and *`Ruletest`* tab names to help users who have difficulty seeing the red and gray colors. Default value is `false`.

```
com.corticon.designer.valuesets.compressLargeSetsUsingNot=false
```

Determines whether Conditional rule column value sets are automatically converted to their logically equivalent negated form upon *`Rulesheet`* collapse or compression. This is done in order to compress the value set down to a more manageable size. If the flag is set to true and a value set contains at least 2/3 of the possible Values for the condition then the system converts it to the negated form.

```
xmi.import.ecore.readonlyflag.default=true
```

Determines whether Vocabulary property values are read-only when the Vocabulary is in edit mode. Read-only values are displayed with a light gray background to differentiate them from modifiable values. Read-only true means values are not modifiable. Read-only false means values are modifiable. Default value: `true`.

```
com.corticon.eclipse.platform.io.useuuids=false
```

Determines whether serialized versions of emf resources use Universally Unique IDs (UUIDs AKA GUIDs) or URI strings to reference other serialized elements. One reason to use UUIDs is to keep related resources in sync if they are moved to different locations. Use of URI strings will only work if the elements are always kept in the same relative locations. The value `true` means, yes, use UUIDs, while the value false means use URI strings. Default value: `false`.

```
com.corticon.tester.ccserver.maxdecisionservices=15
```

Specifies the max number of Decision Services generated through Tester API that can reside on the Server at one time. Default value is `15`.

```
com.corticon.localize.expressions=true
```

Specifies whether the logic used to localize rule expressions will be invoked (true) or not (false). Default value is `true`.

```
com.corticon.tester.associations.includedomainandtype=true
```

Specifies whether test tree node association text will display domain and target entity type information. Default value is `true`.

```
com.corticon.studio.migration.datesubtype.default=Full DateTime
```

Specifies what current Date Data that will be mapped to a 4.1 Date Datatype, which does not contain a Subtype. Default value is `Full DateTime`. Other options include `Date Only` and `Time Only`.

```
com.corticon.eclipse.ui.dropdowns.visiblerows=10
```

Specifies the number of visible rows in a Drop Down Combo Box. Default value is `10`.

```
com.corticon.studio.swt.virtualization=false
```

Specifies whether SWT virtualization will be enabled. SWT virtualization can improve the initial load times of larger *`Ruletest`* assets by deferring the creation of tree items until they are actually needed. Default value is `false`.

```
com.corticon.designer.tested.xmlmessagingstyle=hier
```

This property sets the Studio Test's XML messaging style to one of the following:

- Hier (hierarchical)
- Flat
- Autodetect

These are exactly equivalent to the messaging styles selectable in the Deployment Descriptor file, described here.

```
com.corticon.crml.CrmlGraphVisualizer.fontname=arial.ttc
com.corticon.crml.CrmlGraphVisualizer.fontsize=9
```

This properties set the font type and size used by the Graphic Visualizer. Default values are `arial.ttc` and `9`, respectively.

```
com.corticon.eclipse.ui.completeness.check.autosize=true
```

Specifies whether columns added via the Completeness Checker will be automatically sized based on the data in the columns. Default value is `true`.

Corticon Server properties (CcServer.properties)

The following properties are used by the Corticon Server:

Settings that restrict each of the three types of Rule Messages (info, warning, and violation) from being posted to the output of an execution. The default value for each of the properties is `false` -- that message type is not restricted.

```
com.corticon.server.restrict.rulemessages.info=false
com.corticon.server.restrict.rulemessages.warning=false
com.corticon.server.restrict.rulemessages.violation=false
```

```
com.corticon.ccserver.sandboxDir=%CORTICON_HOME%/CcServerSandbox
```

Determines the path to an existing directory used exclusively by Corticon Server to persist and retrieve deployment assets. If the path does not exist, the Corticon Server attempts to automatically create it. If this fails then the Corticon Server is unable to startup. Default is `%CORTICON_HOME%/CcServerSandbox`

```
com.corticon.ccserver.serviceIntervals=30000
```

Corticon Server has a maintenance service that is tasked with keeping the state of the Decision Service pools up-to-date. The `serviceIntervals` property determines the number milliseconds elapsed in between service cycles during which Corticon Server checks for:

- Changes in any `.cdd` loaded to Corticon Server by a `loadFromCdd` or `loadFromCddDir` call
- Changes in any `.cdd` file including new cdds within the directory of cdds from a `loadFromCddDir` call
- Changes in any of the Decision Services (`.erf` files) loaded to the server. This is done via a timestamp check.

If any changes are detected, Corticon Server's state is dynamically updated to reflect the changes.

Note: The maintenance service can be shutdown and restarted using:

- `ICcServer.stopDynamicaUpdateMonitoringService()`
- `ICcServer.startDynamicUpdateMonitoringService()`

Default for serviceIntervals is 30 secs (`30000` ms)

```
com.corticon.ccserver.inactivity=60000
```

Amount of time (in milliseconds) that a Reactor can remain idle before it is eligible for reclamation by the JVM's garbage collection mechanism. Each Decision Service has pooling parameters for minimum & maximum pool sizes. The pool is initialized to the minimum number of Reactors and grows to meet demand until maximum pool size is reached. When the level of demand drops, any Reactors in the pool that remain idle for the specified period are removed from the pool until the minimum size is reached. This process releases resources and reduces the memory footprint of Corticon Server. Default is `60000` milliseconds (60 seconds).

```
com.corticon.ccserver.dynamicupdatemonitor.autoactivate=true
```

Determines whether the Dynamic Update Monitor Service should be started automatically when Corticon Server is initialized. Default is `true` (Starts the Update Monitor Service automatically).

```
com.corticon.ccserver.servermessages=false
```

Determines whether to display Corticon Server messages to `System.out`, reflecting the state of the Decision Service pools in Corticon Server at initialization and update times. The state of the "wait" queues as well as the Reactor allocations are also reflected. This is primarily a debugging property to be used under instructions from Progress technical support. Default is `false` (no messages).

```
com.corticon.reactor.rulebuilder.maxloops=100
```

Set max loop iteration. Determines what constitutes an endless loop. For `.ers` files with the **Process Logical Loops** setting on, it is necessary to have a safety net to prevent endless loops. This is done by designating the maximum number of iterations allowed for any loop. Default is `100`.

```
com.corticon.reactor.rulebuilder.exception=raise
```

Set maxloop exception handling {raise, bury}. Specifies whether the rule engine will raise a `MaxLoopsExceededException` if the maximum number of loop iterations is exceeded. Default value is `raise`.

```
com.corticon.ccserver.compiler.javahome.location=
```

Specifies the location of the JRE that will be used by the Corticon Server to compile the *`Ruleflow`*s into Decision Services. If not specified, the Corticon Server will use the same JRE that started the Corticon Server by calling into `System.getProperty("java.home")`. Default value is looked up using `System.getProperty("java.home")`.

```
com.corticon.cdolistener.collectionmapping=java.util.Vector
com.corticon.cdolistener.listmapping=java.util.ArrayList
com.corticon.cdolistener.setmapping=java.util.HashSet
```

Specify which implementation class to be used when a supported interface is used for an association inside the user's mapped Business Object. This is needed by the BusinessObject Listener class, which is compiled during Decision Service deployment. Supported interfaces include:

- java.util.Collection
- java.util.List
- java.util.Set

Default values are:

- java.util.Collection=java.util.Vector
- java.util.List=java.util.ArrayList
- java.util.Set=java.util.HashSet

```
com.corticon.server.compile.classpath.include.bos=true
```

Specify whether the Decision Service compile process should dynamically detect the location of the Jars where the Java Business Objects reside. Primary focus is to incorporate customer Java Business Objects in the Ant Classpath so that Listener Generation will succeed. Default value is `true`.

```
com.corticon.server.compile.classpath.include.alljarsunderccserver=false
```

Specify whether the Decision Service compile process should include all the Jars that are in the same directory as the `CcServer.jar` in the Ant Compile Classpath. This may need to be set to true dependent on the type of Application Server the Decision Services are deployed on. Primary focus is to incorporate customer Java Business Objects in the Ant Classpath so that Listener Generation will succeed. Default value is `false`.

```
com.corticon.reactor.engine.CheckForAssociationDuplicates=true
```

Specifies whether the rule engine conducts an integrity check when adding to an existing association. This integrity check ensures that rules do not add redundant associations between the same two entities. Although, this is a rare that occurance, it is possible. The downside of this integrity check is that Decision Services that create a significant number of new associations can experience a performance degradation. Such Decision Services would require this configuration property to be set to false. Default value is `true`.

```
com.corticon.reactor.rulebuilder.UseLoopContainerStrategy=false
```

Specifies whether the rule engine uses Loop Container Strategy. Loop Container Strategy will create a Rule container object for rules that form a loop, just as when loops are enabled, so that when sequential rules are executed they are executed as if they are in a loop, but without looping. Default value is `true`.

```
com.corticon.BuildWaitTime=300000
```

Specifies the amount of time in milliseconds that the Ant build processor will wait before automatically timing out. Default value is `300000` (5 minutes).

```
com.corticon.server.monitoring.decisionservice.record.times=false
com.corticon.server.monitoring.decisionservice.record.times.total=100
```

These properties are related to Decision Service/Version level monitoring.
`com.corticon.server.monitoring.decisionservice.record.times` specifies whether the Server will auto-start recording time measurements.

Note:

The performance monitoring service can also be shutdown and restarted using the following methods, which will override this setting.

- `ICcServer.stopServerPerformanceMonitoringService()`
- `ICcServer.startServerPerformanceMonitoringService()`

Default value is `false`

`com.corticon.server.monitoring.decisionservice.record.times.total` is the number of execution times to be stored for each Decision Service/Version

Default value is `100`

```
com.corticon.server.monitoring.decisionservice.record.date=false
com.corticon.server.monitoring.decisionservice.record.data.registration.delimiter=;
com.corticon.server.monitoring.decisionservice.record.data.bucket.registration.delimiter=,
com.corticon.server.monitoring.decisionservice.record.data.bucket.results.delimiter=:
```

Properties related to Decision Service/Version level monitoring.

`com.corticon.server.monitoring.decisionservice.record.data` specifies whether *`Corticon Server`* will auto-start recording time measurements.

Note: The data recording monitoring service can be shutdown and restarted using the following API methods, which will override this setting.

- `ICcServer.stopServerResultsDistributionMonitoringService()`
- `ICcServer.startServerResultsDistributionMonitoringService()`

Default value is `false`.

`com.corticon.server.monitoring.decisionservice.record.data.registration.delimiter` specifies the delimiter to use when registering default Tracking Attributes. Default value is `;`

`com.corticon.server.monitoring.decisionservice.record.data.bucket.registration.delimiter` specifies the delimiter to use when registering range buckets for the monitoring service. Default value is `,`

`com.corticon.server.monitoring.decisionservice.record.data.bucket.results.delimiter` specifies the delimiter to use between bucket definition and bucket counter when reporting results from the monitoring service. Default value is `:`

```
com.corticon.server.monitoring.decisionservice.interval.record.times=false
com.corticon.server.monitoring.decisionservice.interval.record.sleep=10000
com.corticon.server.monitoring.decisionservice.interval.record.total=50
```

These properties control monitoring execution times of Decision Service/Versions over defined interval periods.

`com.corticon.server.monitoring.decisionservice.interval.record.times` specifies whether *`Corticon Server`* will auto-start recording time interval measurements.

Note: The time interval monitoring service can be shutdown and restarted using the following API methods, which will override this setting.

- `ICcServer.stopServerExecutionTimesIntervalService()`
- `ICcServer.startServerExecutionTimesIntervalService()`

Default value is `false`

`com.corticon.server.monitoring.decisionservice.interval.record.sleep` indicates the number of milisconds that the interval results will be recorded. Default value is `10000` (10 seconds)

`com.corticon.server.monitoring.decisionservice.interval.record.total` indicates the number of past intervals that will be stored in memory. Default value is `50`

```
com.corticon.server.soap.collection.results.delimiter=;
```

Specifies the delimiter to be used when results from an RPC need to be converted from a Collection to a String. Default value is *`;`*

```
com.corticon.ccserver.autoloaddir.enable=true
com.corticon.ccserver.autoloaddir=
```

Specify if and from where `.cdd` files get "auto loaded" into Corticon Server when it starts up.

Note: This property changed using following method, which will override this setting.

- `ICcServer.setDeploymentDescriptorDirectoryPath(String)`

Default value: If `autoloaddir.enable` is `true`, then Corticon Server will automatically read the path specified in `autoloaddir` and attempt to reload any Decision Services referenced in any `.cdd` files it finds there. If `false`, Corticon Server will not try to reload Decision Services deployed via `.cdd` files.

If `autoloaddir` is empty, the following path is used: `<user.dir>\cdd` where `<user.dir>` is the value of environment variable "user.dir" which in Windows and Unix returns the directory where the container application was started in.

```
com.corticon.ccserver.appendservertimes=false
```

Specifies whether the Server Execution start and stop times are appended to the CorticonResponse document after ICcServer.execute(String) or ICcServer.execute(Document) is performed. Default value is `false`.

```
com.corticon.ccserver.cloneAssociationHashSets=false
```

Determines whether CDO association accessor ("getter") methods will return clones of their association HashSets. Normally, an association getter will return a direct reference to the association HashSet.

The default value (`false`) provides the best performance, because cloning an association HashSet can trigger unnecessary database I/O due to lazy-loading.

You can use this property to overcome ConcurrentModificationException errors which may arise when a Rulesheet has two aliases assigned to the same association, and that Rulesheet contains action statements that modify the association collection.

Note that this property only applies to many-to-many associations; for many-to-1 associations, the CDOs will always return a direct reference to the "singleton" HashMap. Default is `false`

```
com.corticon.server.serverstate.load=true
```

Determines whether Corticon Server will initially load the `ServerState.xml` file to restore the Corticon Server to its previous state. Default is `true`

```
com.corticon.server.serverstate.persistchanges=true
```

Determines whether Corticon Server will persist its state inside of the `ServerState.xml`. By default this feature is turned on. Default is `true`

```
com.corticon.reactor.rulebuilder.DisableNullCheckingOnExtensions=false
```

By default, attributes are checked for null values to prevent further rule evaluation. This property will disable the null checks on attributes used in an extension call out, thereby allowing null values to be passed into an extended operator call.

Corticon Deployment Console properties (CcDeployment.properties)

The following properties are used by the Deployment Console:

```
com.corticon.deployment.supportBPELinWSDLgeneration=false
```

Support BPEL in WSDL generation. Used to add a partnerlink section to the generated WSDL to make them BPEL compliant. Default is `false`.

```
com.corticon.xml.addDefaultNamespace=true
```

Adds the default namespace declaration to WSDL generation.

```
com.corticon.schemagenerator.addDefaultNamespace=true
```

Adds the default namespace declaration to XSD generation.

```
com.corticon.deployment.soapbindingurl_1=http://localhost:8082/axis/services/Corticon
```

This setting determines the default value of URL used by the Deployment Console. It does NOT determine where Corticon Studio looks when testing remote Decision Services (see Controlling Corticon Studio for instructions on changing remote Server locations in Corticon Studio). Other listed options include default port settings for other common web or application servers.

Defaults are `http://localhost:8082/axis/services/Corticon` (typically used by the bundled Tomcat Server).

`com.corticon.deployment.soapbindingurl_2=http://localhost:9080/axis/services/Corticon` (typically used by IBM WebSphere)

`com.corticon.deployment.soapbindingurl_3=http://localhost:7001/axis/services/Corticon` (typically used by Oracle/BEA WebLogic)

```
com.corticon.deployment.schema.useChoice=true
```

This property controls whether <choice> or <sequence> tags are used for the <WorkDocuments> section of the generated XSD/WSDL. When `useChoice` is set to true, <choice> tags are used which results in more flexibility in the order in which entity instances appear in the XML/SOAP message. When `useChoise` is set to false, <sequence> tags are used which requires that entity instances appears in the same order as they appear in the <WorkDocuments> section of the XSD/WSDL. Some Web Services platforms do not properly support <choice> tags. For these platforms, this property should be set to false. Default is `true`.

```
com.corticon.servicecontracts.ensureComplianceWithDotNET_WCF=false
```

Determines whether generated service contracts (WSDL/XSD) are compliant with Microsoft .NET WCF. This property must be set to `true` when *`Corticon Server`* is deployed inside a Microsoft WCF container. Note: WSDLs meant for .NET consumption should be generated in Hier XML Messaging Style . Default is `false`.

```
com.corticon.ccdeployment.sandboxDir=%CORTICON_HOME%/CcDeploymentSandbox
```

Determines the path to an existing directory used exclusively by the Deployment Console to pre-compile *`Ruleflow`* files into `.eds` files. Default is `%CORTICON_HOME%/DecisionServerSandbox`. Note, this is not the same property as the `sandboxDir` used in CcServer.properties.

```
com.corticon.deployment.ensureUniqueTargetNamespace=true
```

This property will tell the XSD and WSDL Generators to create unique Target Namespaces inside the output document.

If the property is set to `true`, the following template will be used to create the Target Namespaces for the XSD and WSDL Documents:

- XSD: `urn:decision/<Decision Service Name>`
- WSDL: `<soap binding uri>/<Decision Service Name>`

If the property is set to `false`, the following template will be used to create the Target Namespaces for the XSD and WSDL Documents:

- XSD: `urn:Corticon`
- WSDL: `urn:CorticonService`

Default is `true`. For backwards compatibility with Corticon Server version 5.1 and earlier, use `false`.

D

Third party acknowledgments

One or more products in the Progress Corticon v5.3.3 release includes third party components covered by licenses that require that the following documentation notices be provided:

Progress Corticon v5.3.3 incorporates Apache Commons Discovery v0.2 from The Apache Software Foundation. Such technology is subject to the following terms and conditions: The Apache Software License, Version 1.1 - Copyright (c) 1999-2001 The Apache Software Foundation. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end-user documentation included with the redistribution, if any, must include the following acknowlegement: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgement may appear in the software itself, if and wherever such third-party acknowledgements normally appear.

4. The names "The Jakarta Project", "Commons", and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.

5. Products derived from this software may not be called "Apache" nor may "Apache" appear in their names without prior written permission of the Apache Group.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation. For more information on the Apache Software Foundation, please see <http://www.apache.org/>.

Progress Corticon v5.3.3 incorporates Apache SOAP v2.3.1 from The Apache Software Foundation. Such technology is subject to the following terms and conditions: The Apache Software License, Version 1.1 Copyright (c) 1999 The Apache Software Foundation. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "SOAP" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org. 5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation. THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation. For more information on the Apache Software Foundation, please see <http://www.apache.org/>.

Progress Corticon v5.3.3 incorporates DOM4J v1.6.1. Such technology is subject to the following terms and conditions: Project License BSD style license Copyright 2001-2005 (C) MetaStuff, Ltd. All Rights Reserved.

Redistribution and use of this software and associated documentation ("Software"), with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain copyright statements and notices. Redistributions must also contain a copy of this document.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name "DOM4J" must not be used to endorse or promote products derived from this Software without prior written permission of MetaStuff, Ltd. For written permission, please contact dom4j-info@metastuff.com.

4. Products derived from this Software may not be called "DOM4J" nor may "DOM4J" appear in their names without prior written permission of MetaStuff, Ltd. DOM4J is a registered trademark of MetaStuff, Ltd.

5. Due credit should be given to the DOM4J Project - http://www.dom4j.org

THIS SOFTWARE IS PROVIDED BY METASTUFF, LTD. AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL METASTUFF, LTD. OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT,

INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Progress Corticon v5.3.3 incorporates Jaxen v1.0. Such technology is subject to the following terms and conditions: JAXEN License - $Id: LICENSE,v 1.3 2002/04/22 11:38:45 jstrachan Exp $ - Copyright (C) 2000-2002 bob mcwhirter and James Strachan. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.

3. The name "Jaxen" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@jaxen.org.

4. Products derived from this software may not be called "Jaxen", nor may "Jaxen" appear in their name, without prior written permission from the Jaxen Project Management (pm@jaxen.org).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgement equivalent to the following: "This product includes software developed by the Jaxen Project (http://www.jaxen.org/)." Alternatively, the acknowledgment may be graphical using the logos available at http://www.jaxen.org/. THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE Jaxen AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the Jaxen Project and was originally created by bob mcwhirter <bob@werken.com> and James Strachan <jstrachan@apache.org>. For more information on the Jaxen Project, please see <http://www.jaxen.org/>.

Progress Corticon v5.3.3 incorporates JDOM v1.0 GA. Such technology is subject to the following terms and conditions: $Id: LICENSE.txt,v 1.11 2004/02/06 09:32:57 jhunter Exp $ - Copyright (C) 2000-2004 Jason Hunter and Brett McLaughlin. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.

3. The name "JDOM" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact <request_AT_jdom_DOT_org>.

4. Products derived from this software may not be called "JDOM", nor may "JDOM" appear in their name, without prior written permission from the JDOM Project Management <request_AT_jdom_DOT_org>.

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgement equivalent to the following: "This product includes software developed by the JDOM Project (http://www.jdom.org/)."

Alternatively, the acknowledgment may be graphical using the logos available at http://www.jdom.org/images/logos. THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JDOM AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the JDOM Project and was originally created by Jason Hunter <jhunter_AT_jdom_DOT_org> and Brett McLaughlin <brett_AT_jdom_DOT_org>. For more information on the JDOM Project, please see <http://www.jdom.org/>.

Progress Corticon v5.3.3 incorporates Saxpath v1.0. Such technology is subject to the following terms and conditions: Copyright (C) 2000-2002 werken digital. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.

3. The name "SAXPath" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@saxpath.org.

4. Products derived from this software may not be called "SAXPath", nor may "SAXPath" appear in their name, without prior written permission from the SAXPath Project Management (pm@saxpath.org).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgement equivalent to the following: "This product includes software developed by the SAXPath Project (http://www.saxpath.org/)." Alternatively, the acknowledgment may be graphical using the logos available at http://www.saxpath.org/ THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE SAXPath AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the SAXPath Project and was originally created by bob mcwhirter <bob@werken.com> and James Strachan <jstrachan@apache.org>. For more information on the SAXPath Project, please see <http://www.saxpath.org/>.