Progress®

Corticon Server:
Integration and Deployment Guide

Progress® Corticon®

Notices

Copyright agreement

© 2016 Progress Software Corporation and/or one of its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Business Making Progress, Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, Deliver More Than Expected, Icenium, Kendo UI, Making Software Work Together, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Business Making Progress, Progress Software Developers Network, Rollbase, RulesCloud, RulesWorld, SequeLink, Sitefinity (and Design), SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. AccelEvent, AppsAlive, AppServer, BravePoint, BusinessEdge, DataDirect Spy, DataDirect SupportLink, Future Proof, High Performance Integration, OpenAccess, ProDataSet, Progress Arcade, Progress Profiles, Progress Results, Progress RFID, Progress Software, ProVision, PSE Pro, SectorAlliance, Sitefinity, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, WebClient, and Who Makes Progress are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

Please refer to the Release Notes applicable to the particular Progress product release for any third-party acknowledgements required to be provided in the documentation associated with the Progress product.

Table of Contents

Chapter 1: Introduction to Corticon Server deployment 11

Choose the deployment architecture.....11

Installation option 1: Web services.....13

Installation option 2: Java services with XML message payloads.....13

Installation option 3: Java services with Java object payloads.....14

Installation option 4: In-process Java classes with Java object or XML payloads.....14

Chapter 2: Types of Corticon Servers.....15

Chapter 3: Preparing Studio files for deployment.....17

Mapping the Vocabulary.....17

XML mapping.....18

Java object mapping.....20

Entity mapping.....23

Attribute mapping.....24

Association mapping.....26

Java generics.....26

Java enumerations.....26

Verifying Java object mapping.....29

Listeners.....29

Chapter 4: Packaging and deploying Decision Services 31

Deployment related files 32

Rule asset (ECORE, ERS, ERF) files 32

Test asset (ERT) files 32

Database access properties files 32

Corticon Deployment Descriptor (CDD) files.....33

Decision Service (EDS) files.....33

Schema (XSD, WSDL) files.....33

Using Studio to compile and deploy Decision Services 34

Compiling and deploying to a Corticon Server 35

Compiling and saving to Studio disk for later deployment.....37

Deploying Decision Services into Web Console Applications from Studio.....39

Using Web Console to deploy Decision Services.....42

Using Deployment Descriptors to deploy Decision Services.....42

Structure of a Deployment Descriptor (.cdd) file.....43

Setting properties in a CDD file.....43

Setting deployment properties in a CDD file through APIs....44
Example of a complete CDD file....46
Using the Server Deployment Console....47
Setting the autoloaddir property....50
Using command line utilities to compile Decision Services....50
Syntax of the compile and test commands....51
Creating a build process in Ant....58
Using Server API to compile and deploy Decision Services....60

Chapter 5: Integrating Corticon Decision Services....63

Components of a call to Corticon Server....63
The Decision Service Name....64
The Data....64
Service contracts: Describing the call....64
XML workDocument....65
Java business objects....65
Creating XML service contracts with Corticon Deployment Console....65
Types of XML service contracts....69
XML Schema (XSD)....69
Web Services Description Language (WSDL)....69
Annotated Examples of XSD and WSDLs Available in the Deployment Console....69
Passing null values in messages....70

Chapter 6: Secure servers with authentication and encryption....73

Implementing deployment security....73
Setting up authentication for secure server access....74
Setting up encryption between servers and clients....83

Chapter 7: Invoking Corticon Server....85

Methods of calling Corticon Server....85
SOAP call....86
Java call....86
REST call....87
Request/Response mode....88
Administrative APIs....88

Chapter 8: Relational database concepts in the Enterprise Data Connector (EDC)....91

Identity strategies....92
Advantages of using Identity Strategy rather than Sequence Strategy....94
Key assignments....94

Conditional entities....96
Support for catalogs and schemas....97
Support for database views....97
Fully-qualified table names....98
Dependent tables....98
Inferred property values....99
Join expressions....100
Java Data Objects....102

Chapter 9: Implementing EDC....105

Overview of the Enterprise Data Connector....106
Working with EDC in Corticon Studio....107
How Corticon Vocabulary terms relate to a database....107
Connecting a Vocabulary to a database....108
Specifying entity properties for the database schema....113
Creating a schema in the database....115
Importing database metadata into a Vocabulary....115
Mapping and validating database metadata....118
Metadata for Datastore Identity in XML and JSON Payloads....123
Data synchronization....124
Importing an attribute's possible values from database tables....134
Working with EDC in Corticon Server....141
Managing User Access in EDC....141
How EDC handles transactions and exceptions....141
Working with database caches....142
Creating and updating a database schema from a Vocabulary....144

Chapter 10: Inside Corticon Server....145

The basic path....146
About Working Memory....146
Ruleflow compilation into an EDS file....146
Multi-threading, concurrency reactors, and server pools....147
State....149
Reactor state....149
Corticon Server state....150
Dynamic discovery of new or changed Decision Services....151
Replicas and load balancing....151
Exception handling....151

Chapter 11: Decision Service versioning and effective dating....153

Deploying Decision Services with identical Decision Service names....154
Invoking a Decision Service by version number....155
Creating samples of versioned Ruleflows....155

Specifying a version in a SOAP request message....159
Specifying version in a Java API call....161
Default behavior with no target version....162
Invoking a Decision Service by date....162
Modifying the sample Rulesheets and Ruleflows....163
Specifying Decision Service effective timestamp in a SOAP request message....164
Specifying effective timestamp in a Java API call....166
Specifying both major version and effective timestamp....167
Default behavior with no timestamp....167
Summary of major version and effective timestamp behavior....167

Chapter 12: Using Corticon Server logs....169
How users typically use logs....169
Changing logging configuration....170
Configuring log content....170
Configuring log files....171
Troubleshooting Corticon Server problems....172

Chapter 13: Performance and tuning guide....179
Rulesheet performance and tuning....180
Server performance and tuning....180
Optimizing pool settings for performance....180
Single machine configuration....181
Cluster configuration....182
Capacity planning....183
The Java clock....183
Diagnosing runtime performance of server and Decision Services....184

Chapter 14: Enabling Server handling of locales, languages, and timezones....191
Character sets supported....192
Handling requests and replies across locales....193
Examples of cross-locale processing....193
Example of cross-locale literal dates....197
Example of requests that cross timezones....202

Chapter 15: Request and response examples....205
JSON/RESTful request and response messages....205
About creating a JSON request message for a Decision Service....205
Sample JSON request and response messages....211
XML requests and responses....222

Sample XML CorticonRequest content....222
Sample XML CorticonResponse content....223

Chapter 16: Implementing Rule Execution Recording in a database...225

Appendix A: Service contract examples....231

Examples of XSD and WSDLS available in the Deployment Console....232
1 Vocabulary-level XML schema, FLAT XML messaging style....232
2 Vocabulary-level XML schema, HIER XML messaging style....239
3 Decision-service-level XML schema, FLAT XML messaging style....239
4 Decision-service-level XML schema, HIER XML messaging style....240
5 Vocabulary-level WSDL, FLAT XML messaging style....243
6 Vocabulary-level WSDL, HIER XML messaging style....245
7 Decision-service-level WSDL, FLAT XML messaging style....246
8 Decision-service-level WSDL, HIER XML messaging style....247
Extended service contracts....248
Extended datatypes....248

Appendix B: Corticon API reference....251

Java API....251
REST Management API....252
API ListDecisionServices....255
API Deploy Decision Service....256
API Undeploy Decision Service....257
API Get Decision Service Properties....257
API Set Decision Service Properties....258
API Ping Server....260
API Retrieve Metrics....260
API Get Server Log....261
API Get Server Info....262
API Get Server Properties....264
API Set Server Properties....264
API Set Server License....265

Appendix C: Configuring Corticon properties and settings....267

Using the override file, brms.properties....268
Setting override properties in the brms.properties file....269
Common properties....272
Date/time formats in CcCommon.properties....276
Corticon Studio properties....281
Corticon Server properties....284
Corticon Deployment Console properties....293

Appendix D: Supported RDBMS brands and features for Corticon EDC.297

DataDirect Cloud: Rollbase....298
DataDirect Cloud: Salesforce....299
IBM DB2....300
Microsoft SQL Server....300
MySQL....301
Postgres....302
Oracle Database....302
Progress OpenEdge....303
Progress OpenAccess....304

Appendix E: Access to Corticon knowledge resources....305

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.” Rulesheets are not directly deployable to Corticon Server. They must be “packaged” as Ruleflows 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 Ruleflows 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 this Corticon product documentation, offer some help in coping with these constraints.

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

In-process Corticon Server for Java

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 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.

In-process Corticon Server for .NET

Corticon Server for .NET is an implementation of Corticon Server for the .NET framework.

The key component in Corticon Server for .NET is a set of core Java classes packaged as Java Archive (JAR) files. The core Java classes contain all the functionality of Corticon Server including the framework to host and manage Decision Services. Corticon Server for .NET is bundled with a set of libraries, collectively named IKVM.NET, which enable the Java classes to integrate with .NET clients.

Corticon Server for .NET provides deployment tools that enable you to install it on Microsoft Internet Information Services (IIS) so that Decision Services can be invoked as Web Services.

To learn more about installing Corticon Server for .NET on IIS and exposing Decision Services as Web Services, refer to the course Deploying Progress Corticon Decision Services as Web Services with .NET

To learn more about Progress Corticon in process deployment, refer to the course Integrating a Progress Corticon Decision Service with a .NET Client using Object Messaging.

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
- XML mapping
- Java object mapping
- Entity mapping
- Attribute mapping
- Association mapping
- Java generics
- Java enumerations
- Verifying Java object mapping
- Listeners

Mapping the Vocabulary

Part of the integration process involves mapping the Vocabulary terms to the structure of the data that will be sent to the deployed Ruleflows at runtime. This ensures that when the Decision Service is invoked, the data included in the invocation will be understood, translated, and processed correctly.

The Corticon Studio tasks in this section require that you set the Vocabulary to its **Advanced View** to expose the properties related to the mappings.

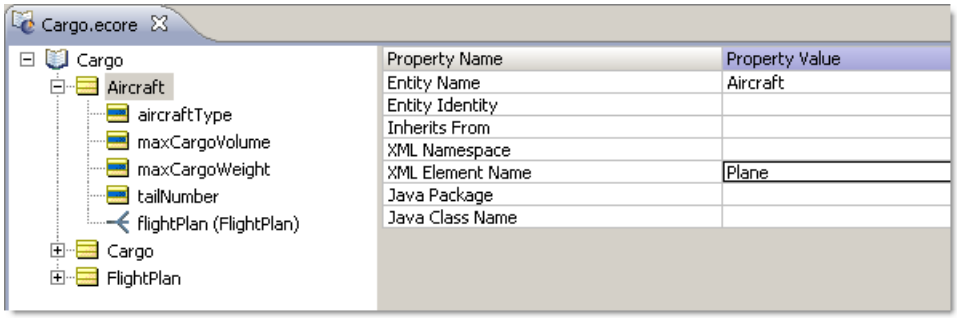
XML mapping

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



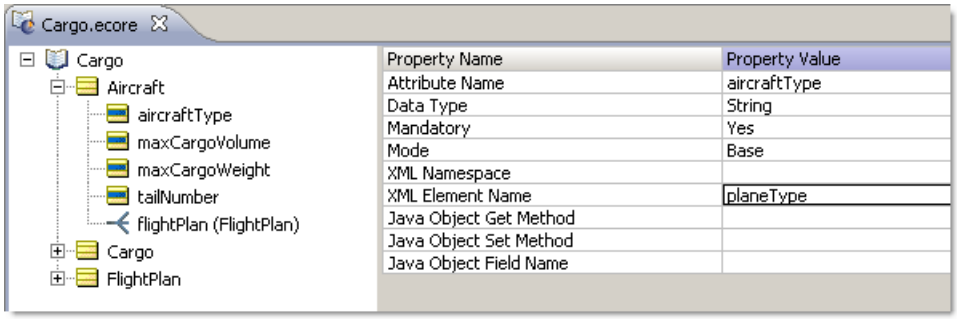
In the example shown in this figure, the Vocabulary entity name (`Aircraft`) does not *exactly* match the name of the external XML Class (`Plane`), so the mapping entry is required. If the two names were identical, then no mapping entry would be necessary.

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

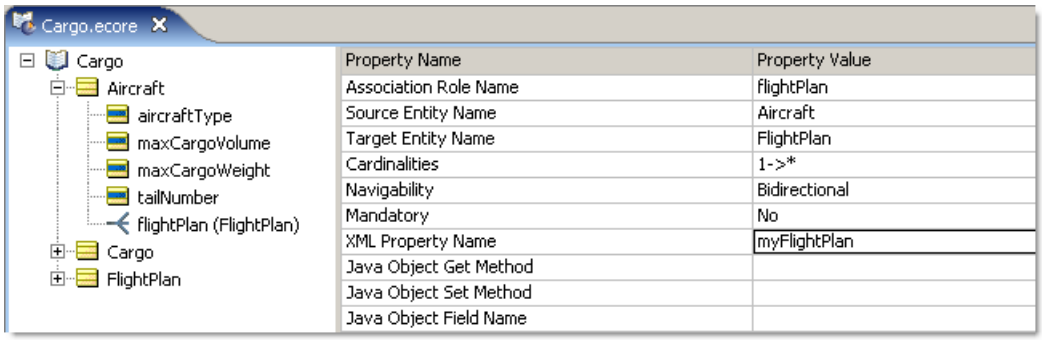


If XML Namespaces vary within the document, then use the **XML Namespace** field to enter the full namespace of the XML Element Name. If no XML Namespace value is entered, then it is assumed that all XML Elements use the same namespace.

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

Corticon Server assumes that incoming XML requests are loosely compliant with the XSD/WSDL generated for a particular Decision Service (by the Deployment Console, for example) so the Corticon XSD/WSDLs that are generated have a generic `targetNamespace` of `urn:Corticon`, as illustrated:

Figure 4: XSD with generic Namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="urn:Corticon"
targetNamespace="urn:Corticon" elementFormDefault="qualified">
  <xsd:element name="CorticonRequest" type="tns:CorticonRequestType" />
  <xsd:element name="CorticonResponse" type="tns:CorticonResponseType" />
```

Figure 5: WSDL with generic Namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="urn:CorticonService"
xmlns:cc="urn:Corticon" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap=
"http://schemas.xmlsoap.org/wsdl/soap/" targetNamespace="urn:CorticonService">
  <types>
    <xsd:schema xmlns:tns="urn:Corticon" targetNamespace="urn:Corticon"
    elementFormDefault="qualified">
      <xsd:element name="CorticonRequest" type="tns:CorticonRequestType" />
      <xsd:element name="CorticonResponse" type="tns:CorticonResponseType" />
```

Setting XML Namespace Mapping preference for unique target namespaces

Systems that are particular about XML validation might require a unique `targetNamespace` -- ideally globally unique.

You can choose to have unique names by setting the deployment propertn a server's `brms.properties` file. The changes will apply after a restart of the Server and the Deployment Console. 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.

The following images are examples of unique namespaces:

Figure 6: XSD with unique Namespace

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

Figure 7: WSDL with unique Namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns=
"http://localhost:8080/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:8080/axis/services/Corticon/tutorial_example">
  <types>
    <xsd:schema xmlns:tns="urn:decision:tutorial_example" targetNamespace=
    "urn:decision:tutorial_example" elementFormDefault="qualified">
```

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.

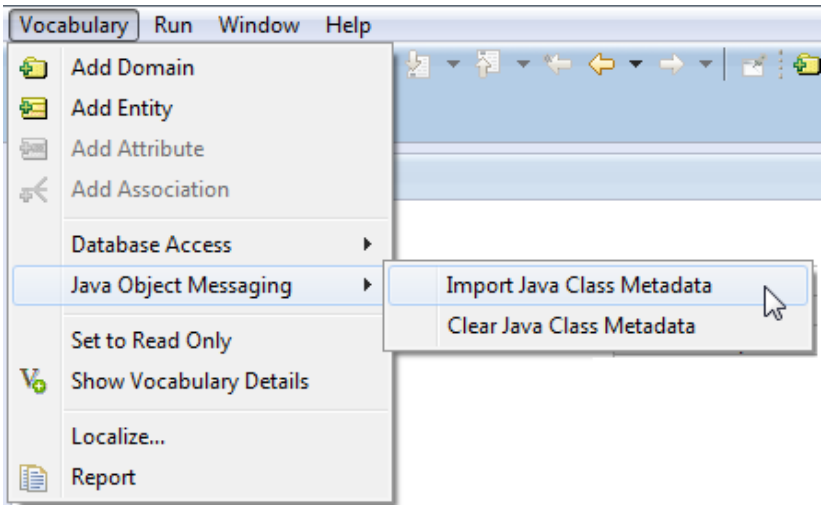
Corticon Studio can import a package of classes and automatically match the object structure with the Vocabulary structure. In other words, it will try to determine which objects match which Vocabulary entities, which properties match which Vocabulary attributes, and which object references match which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

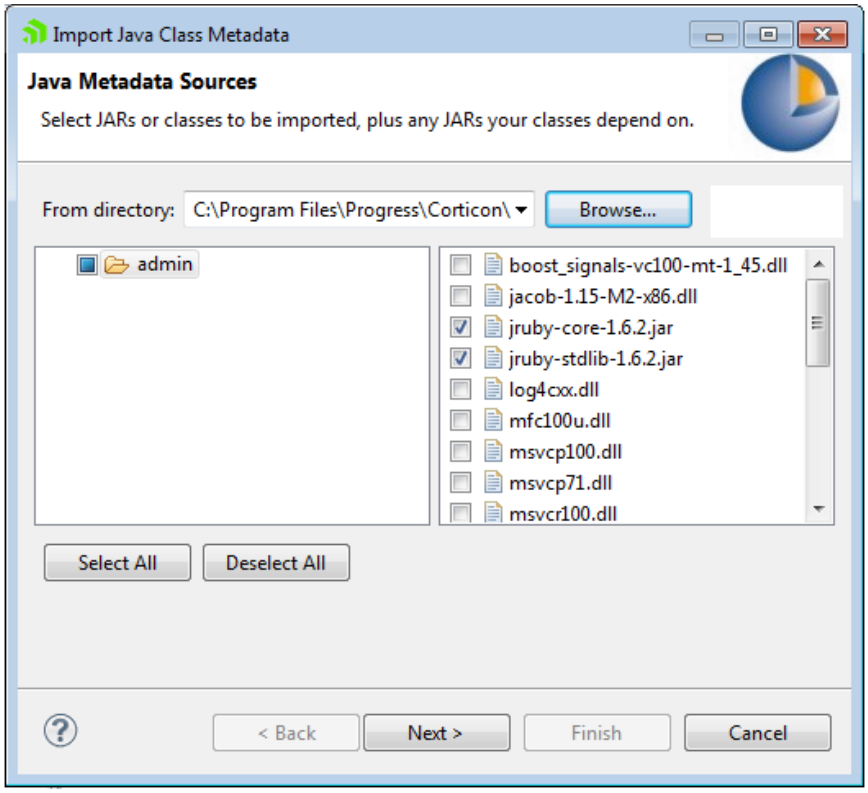
1. Open your Vocabulary in Corticon Studio's **Vocabulary Edit** mode.
2. From the menubar, select **Vocabulary** > **Java Object Messaging** > **Import Java Class Metadata**, as shown in the following figure:

Figure 8: Importing Java Class Metadata for Mapping



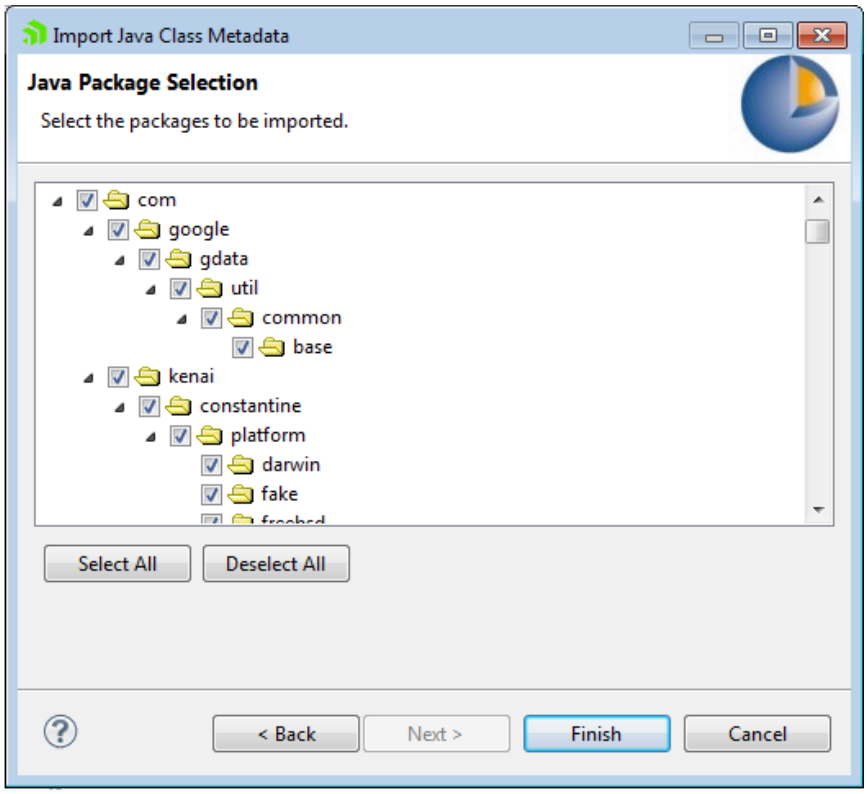
3. Use the **Browse** button to select the location of your Java Business Objects. They should be compiled `class` files or Java archives (`.jar` files).

Figure 9: Browsing to your Java Class files



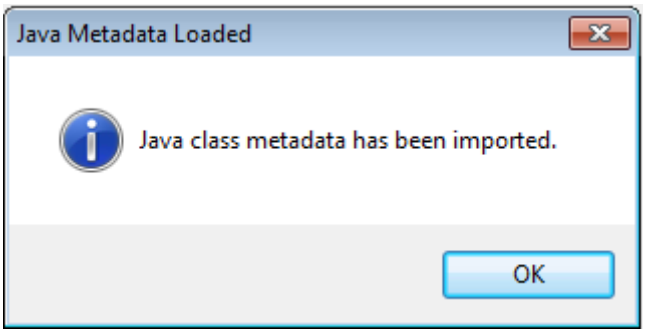
4. Select the package containing the Java business objects as shown:

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 mapping

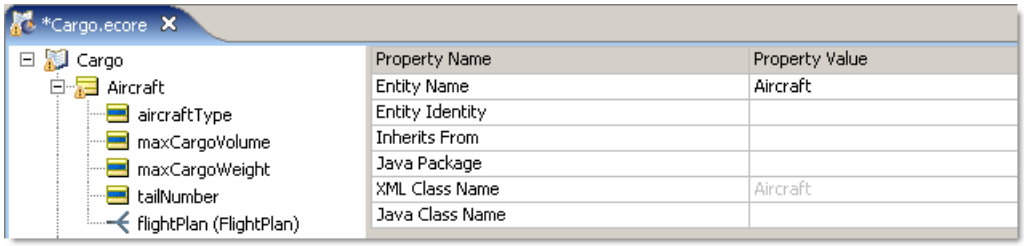
Let's take a look at a sample class that we might have wanted to map to the `Aircraft` entity.

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() {}
```

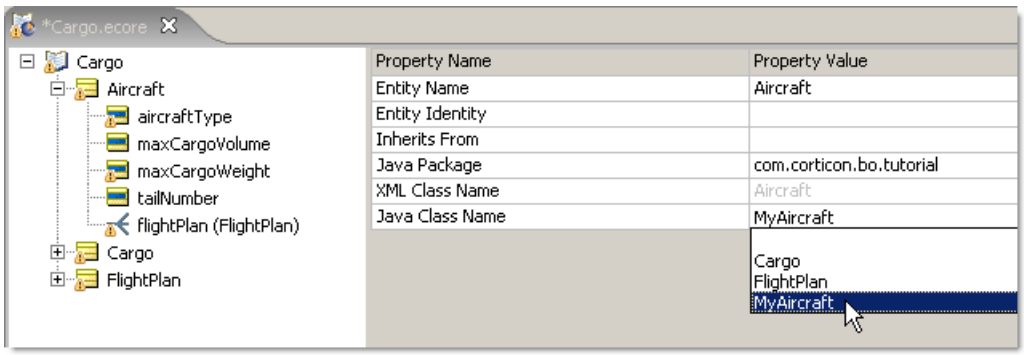
We can see in line 6 of this figure that this class is not actually named `Aircraft` – it is named `MyAircraft`. The automatic mapper attempts to locate a class by the same name as each entity. In the case of `Aircraft`, it looks for a class named `Aircraft`. Not finding one, it leaves the field empty,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

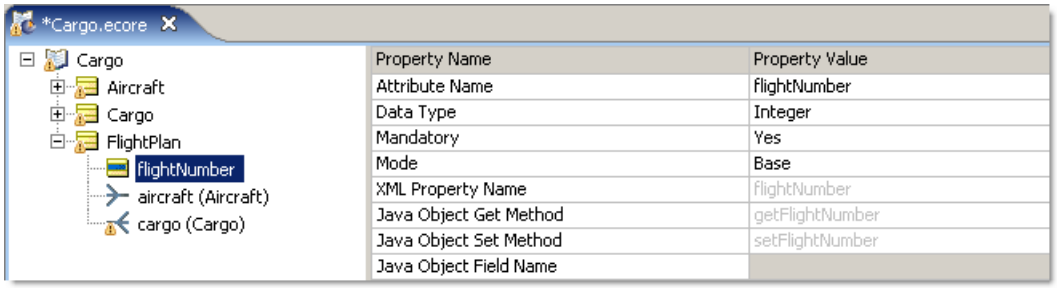
When attempting to map attributes, the mapper looks for class properties which are exposed using public get and set methods by the same name. For example, if mapping attribute `flightNumber`, the mapper looks for public `getFlightNumber` and `setFlightNumber` methods in the mapped class.

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
```

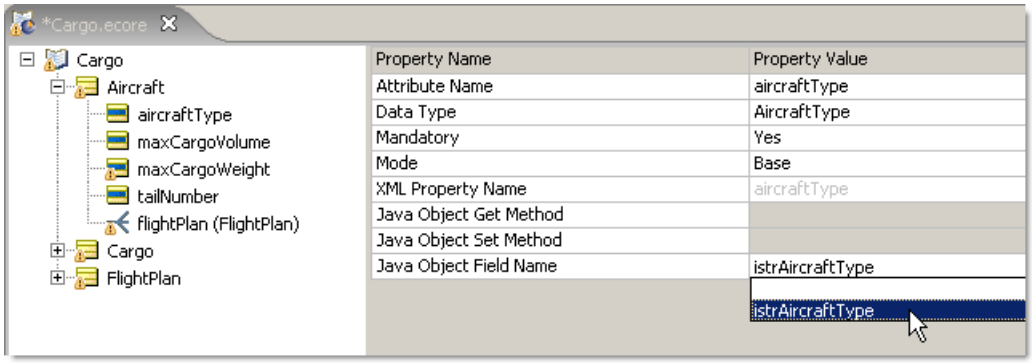
In the case of attribute `tailNumber`, the mapper finds get and set methods that conform to this naming convention, so the method names are inserted into the fields in gray type, as shown in the following figure.

Figure 16: Auto-Mapped Attribute Method Names



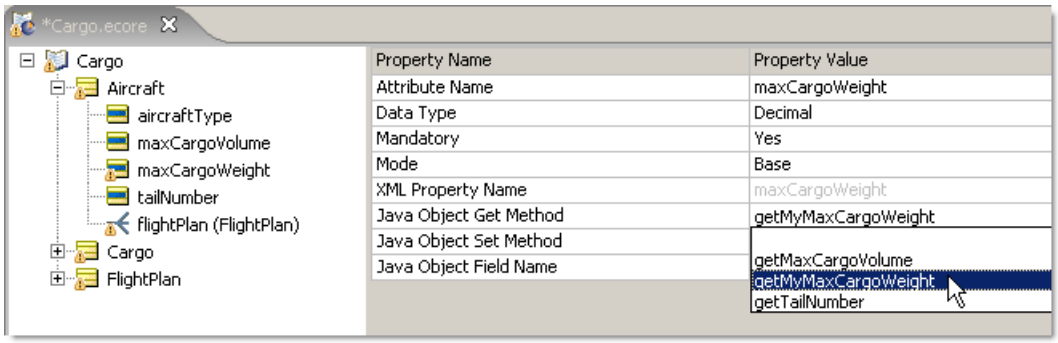
In those cases where the mapper cannot locate the corresponding methods, you will need to select them manually. Notice in the `MyAircraft` class shown in First Portion of MyAircraft Class, no get and set methods exist for `istrAircraftType` since it is a public instance variable. Therefore, we need to 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.

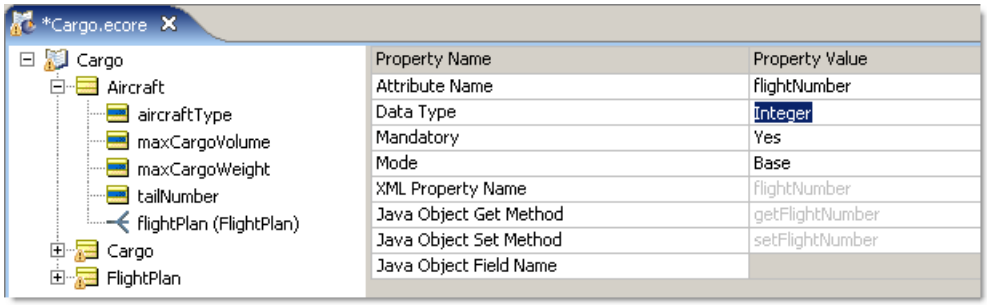
Figure 18: Manually Mapped Property Get and Set Method Names



Note: Java Object Messaging and mapping in versions of Corticon Studio prior to 5.2 required manual mapping of external data types as well.

A property's data type is detected by the auto-mapper, so there is 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.

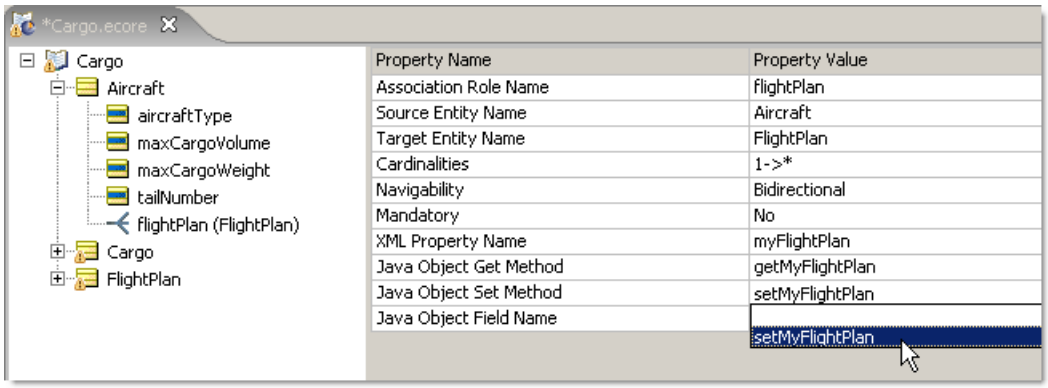
Association mapping

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;}
}
```

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.

Note: 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.

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)

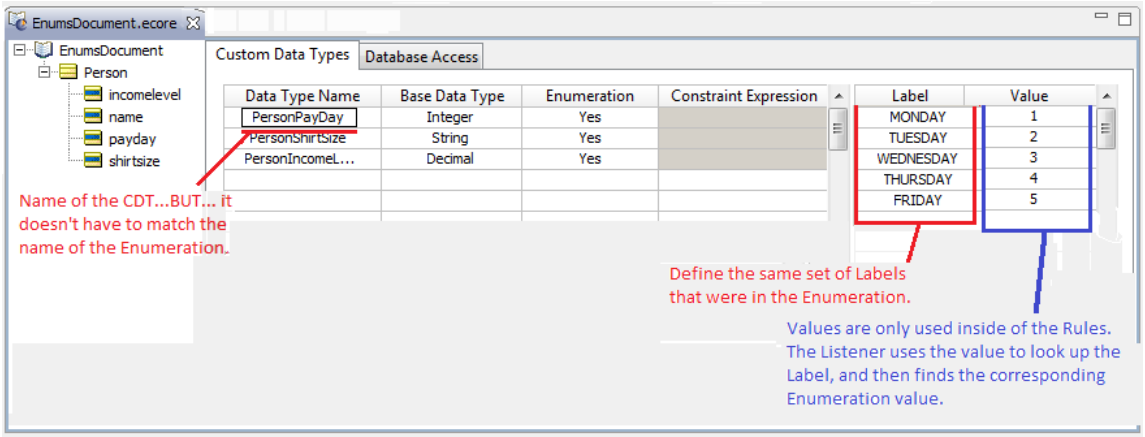
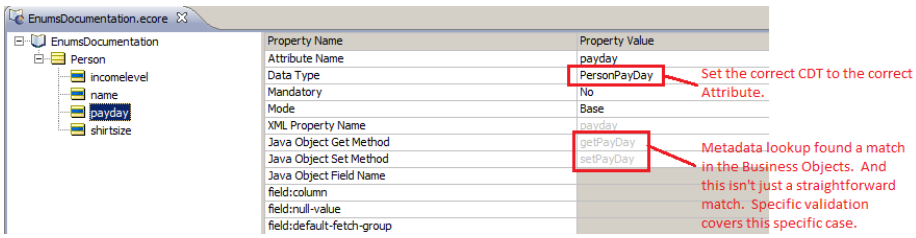


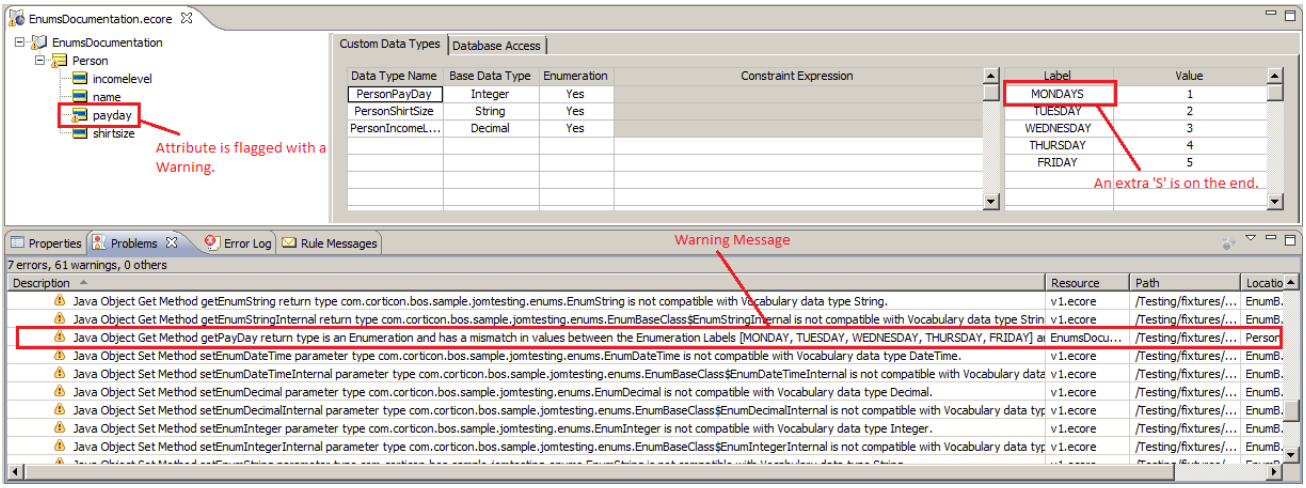
Figure 23: Vocabulary Mapper found correct Metadata based on matching enumeration labels



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



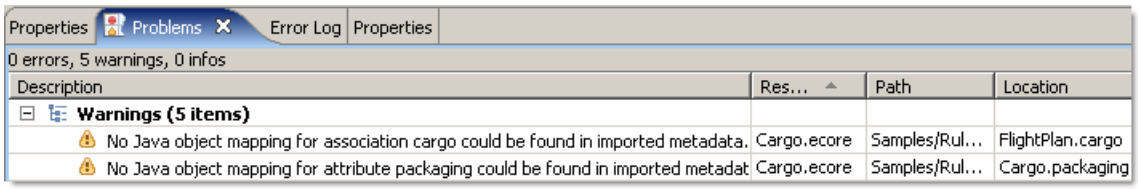
Notice the Vocabulary attribute is “flagged” with the small orange warning icon (shown in the upper left of the figure above). The associated warning message states:

```
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

In several of the preceding illustrations, orange triangle “warning” markers are shown next to Vocabulary nodes whose mappings have not yet been selected. Each warning will have a corresponding message entered in the **Problems** window, as shown:

Figure 25: Problem window showing list of current mapping problems



Note: The **Problems** view typically opens in the lower section of the Corticon Studio window -- if you do not see it, choose **Window > Show View > Problems**.

When all mappings are complete (either automatically or manually), the warning markers are removed.

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

```
CcServer.execute(String, Collection, Integer, Date)
Decision Service DecisionServiceName is not enabled to run
Object Execution.  Vocabulary and Ruleset need to be
regenerated with proper Java Object mappings in place
```

4

Packaging and deploying Decision Services

This section discusses the different approaches for packaging and deploying rules for use in test and production environments. Depending on your experience and your production status, you should start with the fastest and easiest way, and -- as your solution moves toward production -- refine your approach to better manage your deployed rules and Corticon Servers.

When you are developing rules in Corticon Studio, within Studio you can:

- Package and deploy Decision Services directly to a Corticon Server, a good idea for developer integration testing.
- Create deployable Decision Service files that can be delivered to other Corticon servers for later deployment through Server tools.

When you are managing and administering a Corticon Server, you can:

- Deploy Decision Service files which can be deployed with the Web Console or Server APIs.
- Deploy through a Deployment Descriptor file, a text file that identifies one or more Ruleflow or Decision Service files to be deployed and their respective properties to be set on the Decision Service. This is a good idea when you want a file manifest of the deployment.
    - Ruleflow files listed in a Deployment Descriptor file are great in a collaborative test environment, but not recommended for production because it requires the server to compile the rule assets into a Decision Service.
    - Decision Service files listed in a Deployment Descriptor file are recommended for production.

When you want to run Corticon Server in-process, you can:

- Use the Server API to add and manage Decision Services. See the JavaDoc APi for more details.

The next section reviews the file types that are involved in deployment.

For details, see the following topics:

- Deployment related files
- Using Studio to compile and deploy Decision Services
- Using Web Console to deploy Decision Services
- Using Deployment Descriptors to deploy Decision Services
- Using command line utilities to compile Decision Services
- Using Server API to compile and deploy Decision Services

Deployment related files

The path from creating your first Vocabulary to deploying a Decision Service on a production Corticon Server involves several types of files. This section takes a quick overview of the files created in a project to build and test rules all the way through to the deployment files and associated schemas. As the section gets into deployment, it provides links to relevant topics in this guide.

Rule asset (ECORE, ERS, ERF) files

In Corticon, *rule assets* are the essential files that meld the Corticon Rule Language with the structure and typing you created in a vocabulary onto a canvas that sequences the rules, the rule sets, and embeds other canvases into a single Ruleflow that can be packaged and deployed. Some designs have hundreds of rules in dozens of Ruleflows that use an elaborate Vocabulary of entities, atttributes, and associations to define a single Decision Service.

A Corticon Decision Service has all its rule assets available, whether loosely assembled around a Deployment Descriptor, or embedded in a compiled Decision Service.

Test asset (ERT) files

Testing a project is a key aspect of the Corticon Studio's tools. Once a project is packaged and prepared for deployment, it is a good practice to run the tests after building your Decision Service to identify any anomalies and to confirm that the Decision Service behaves correctly.

Database access properties files

When using the Enterprise Data Connector (EDC), you must supply a database access properties file the provides connection information for the database. That file is typically created in Corticon Studio after you have defined and tested a Vocabulary's database connection. See *"Creating a database access properties file"*.

When using CDD deployment, the database access properties file is identified within the CDD file. When using the Web Console or APIs for deployment you specify the file at the time of deployment.

Corticon Deployment Descriptor (CDD) files

Corticon Deploymemt Descriptor (CDD) files let you package Ruleflows, pre-compiled Decision services, and their deployment parameters in an XML-formatted text file.

A Corticon Server loads Deployment Descriptor files. When the Server reads a CDD file, it reads in each instance defined in the file to load its Ruleflow or Decision Service, and then sets its execution and configuration parameters.

(See Setting the autoloaddir property on page 50 for additional information.)

Note: If you are using the bundled Progress Application Server to test and deploy, copy the Deployment Descriptor file to the Corticon Server installation's `[CORTICON_WORK_DIR]\cdd` directory. When Corticon Server starts, it reads all `.cdd` files in that default location.

Deployment Descriptor files are created and managed by:

- Using the Server Deployment Console on page 47 - The graphical Deployment Console is included in both Corticon Server installations.
- Using Deployment Descriptors to deploy Decision Services on page 42 - Authoring lets you extend a CDD with additional available options.

Decision Service (EDS) files

A Decision Service file (`.eds`) is a self-contained, complete deployment asset that includes compiled versions of all its component rule assets. This has the following important consequences:

- Only the `.eds` file needs to be accessible to Corticon Server. The related rule asset files are not needed.
- The `.eds` files are already compiled, so Corticon Server can load them quickly upon deployment, without the lag time required by Ruleflow files in a Deployment Descriptor that requires 'on-the-fly' compilation.
- Corticon Server's dynamic monitoring update service will only check for updates to the `.eds` file's timestamp to know it has been updated and needs to be reloaded. It does not need to monitor for updates to changes to any of the rule assets used to build the .eds file.

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

Note: If your Ruleflow contains Service Call-Outs (SCOs), be sure to add the SCO classes as described in *"Precompiling Ruleflows with service call-outs" in the Extensions Guide.*

Every part of this section shows how its toolset can produce a Decision Service file.

Schema (XSD, WSDL) files

Schema files define a *service contract* -- the interface to a service for clients applications, telling them what can be sent and in what format. Two service contract formats are the Web Services Description Language (WSDL), and the XML Schema (.xsd). The topic Service contracts: Describing the call on page 64 discusses these concepts in greater depth, and the topics in Service contract examples on page 231 show each of the various types.

This section includes Generating XSD and WSDL schema files on page 53 as part of the command line utilities, while Creating XML service contracts with Corticon Deployment Console on page 65 discusses its usage as part of the section "Integrating Corticon Decision Services."

Using Studio to compile and deploy Decision Services

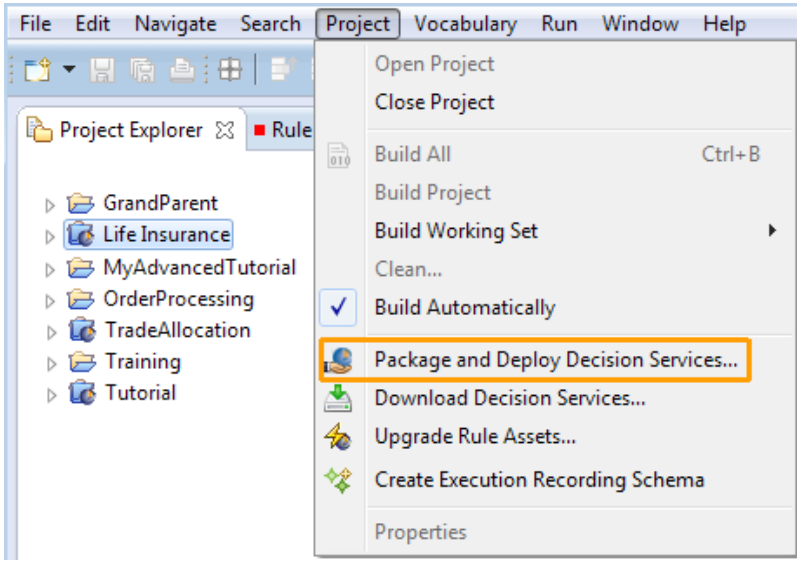
Within Corticon Studio you can package and deploy Decision Services. This is particularly useful during development and testing. In production, you typically would not deploy from Studio.

This fast deployment technique uses the Studio's Package and Deploy Decision Services wizard.

The projects that will be presented in the wizard are determined by the Project Explorer selections and the project of the current file in an editor. To set a context for the wizard:

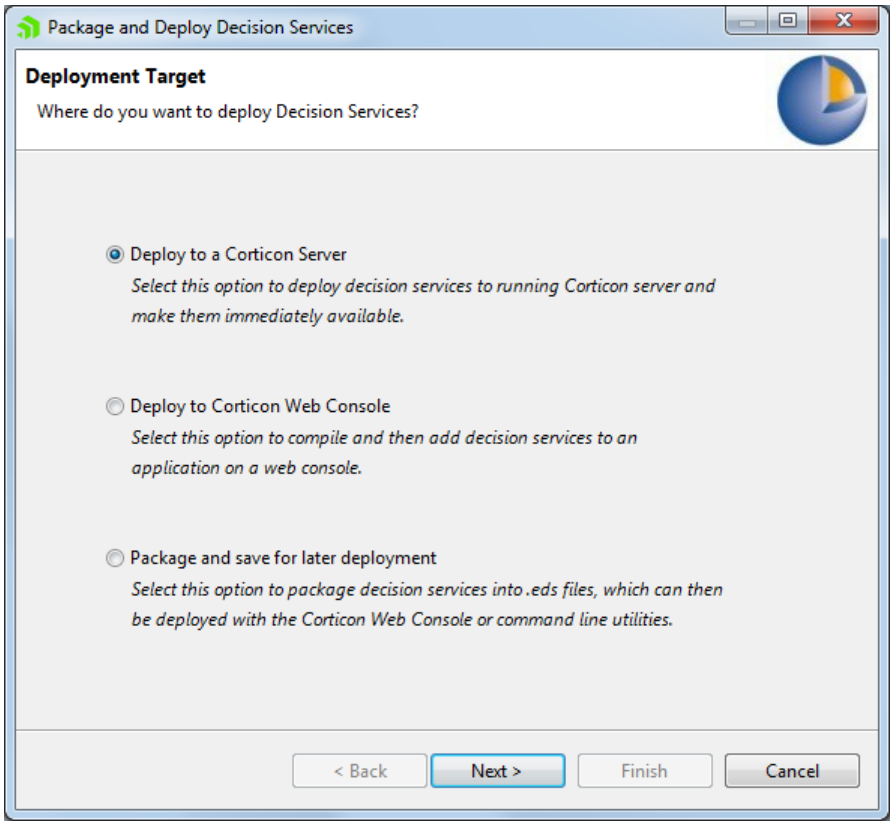
- In the Project Explorer, click one (or Ctrl-click to choose several) Projects. All the Ruleflows in those projects will be listed, as well as projects related to files in open Corticon editors.
- If no projects are selected, the Ruleflows in the project of the active Corticon editor file will be selected and listed.
- If no projects are selected and no Corticon editors are open, Ruleflows in *all* projects in the current workspace will be selected and listed.

Once you have set the wizard's context, choose the **Project** menu's **Package and Deploy Decision Services** action, as shown:



Note: You can choose the same action in the right-click menu of the Project Explorer.

The **Package and Deploy Decision Services** wizard opens:

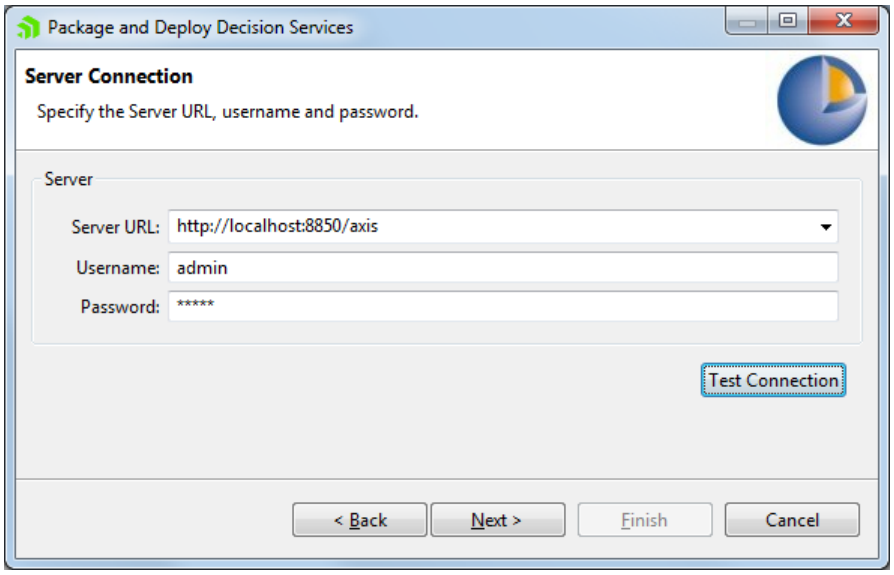


The packaged Decision Services can be saved in the file system, or deployed directly into an available Corticon Server at a specified location and port. Select your preference, and then click **Next**.

Compiling and deploying to a Corticon Server

When you choose to deploy to a Corticon Server, you first define a valid server connection, and then select Ruleflows to compile and deploy to that server.

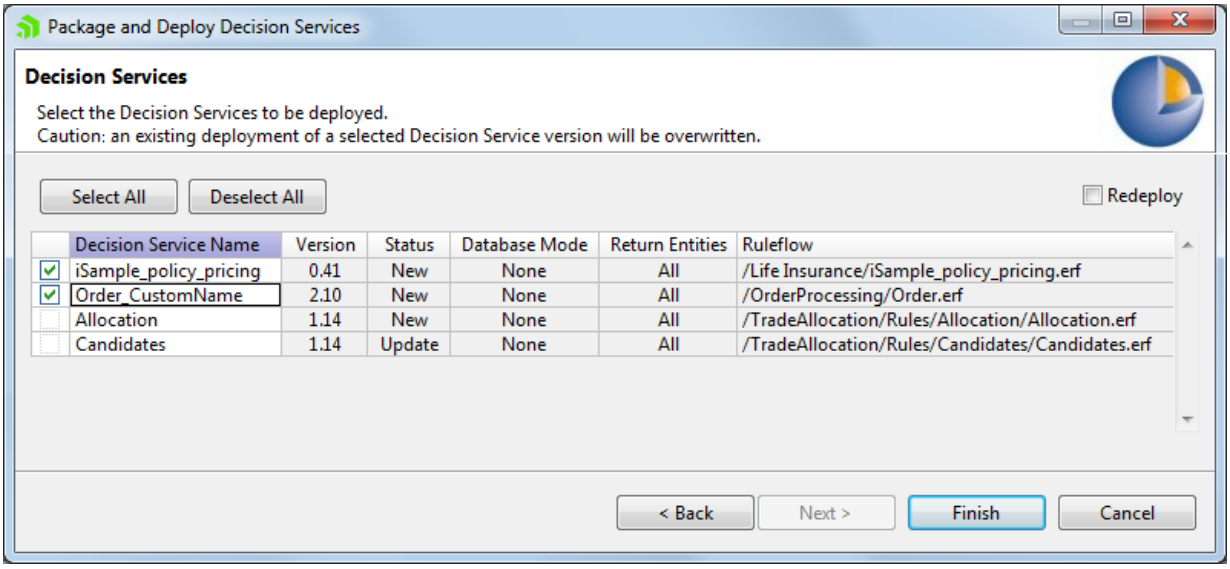
To connect to a Corticon Server from the Server Connection panel:



1. Enter the **Server URL** of the Corticon Server.
2. Enter the **Username** and **Password** for the server. For administrative permissions on Progress Application Server, try the default credentials `admin` and `admin`.
3. Click **Test Connection**.
   - On successful connection, the system displays: **Server connection test was successful.**
   - If the username or password is invalid, the system displays: **User does not have rights to upload/download content to/from the server.**
   - On errors such as the server being unavailable, the system displays: **Server connection test failed. Server may be off-line, unreachable, not listening on specified port, or incorrect Server URL, username, or password.**
   - On an unexpected 'Hard' failure, the system unwraps the **Axis Fault** and finds the underlying cause, such as **404** when the user specifies URL incorrectly.

   Once the connection test is successful, you are able to proceed.
4. Click **Next**.

   The **Decision Services** panel lists the Ruleflows in the context you specified.



The columns on this panel show the following about each Ruleflow:

- The wizard copies the **Decision Service Name** from the Ruleflow file name. You can change any **Decision Service Name** to publish the Decision Service with a preferred name. When you do that, the wizard might toggle the **Status** field between New and Update depending on whether that name is already deployed.
- **Version** is read from the *"Ruleflow" properties (see the Quick Reference Guide).* It is not modifiable here.
- **Status** indicates whether the Decision Service version is New or Update (that is, whether that Decision Service name with that version identity is already deployed on the server).
- **Database mode** for a Ruleflow that will have a database connection.
- **Return entities** for a Ruleflow that will have a database connection.
- **Ruleflow** location within the current workspace.

5. Click the check box for each Ruleflow to be packaged and deployed to the server as Decision Services.

   The wizard does not enable the **Finish** button if any selected Decision Service has the Status 'Update'. You can override this condition by renaming each such Decision Service, or by selecting the **Redeploy** checkbox to override and redeploy all such Ruleflows under the existing name and version.

6. Click **Finish**.

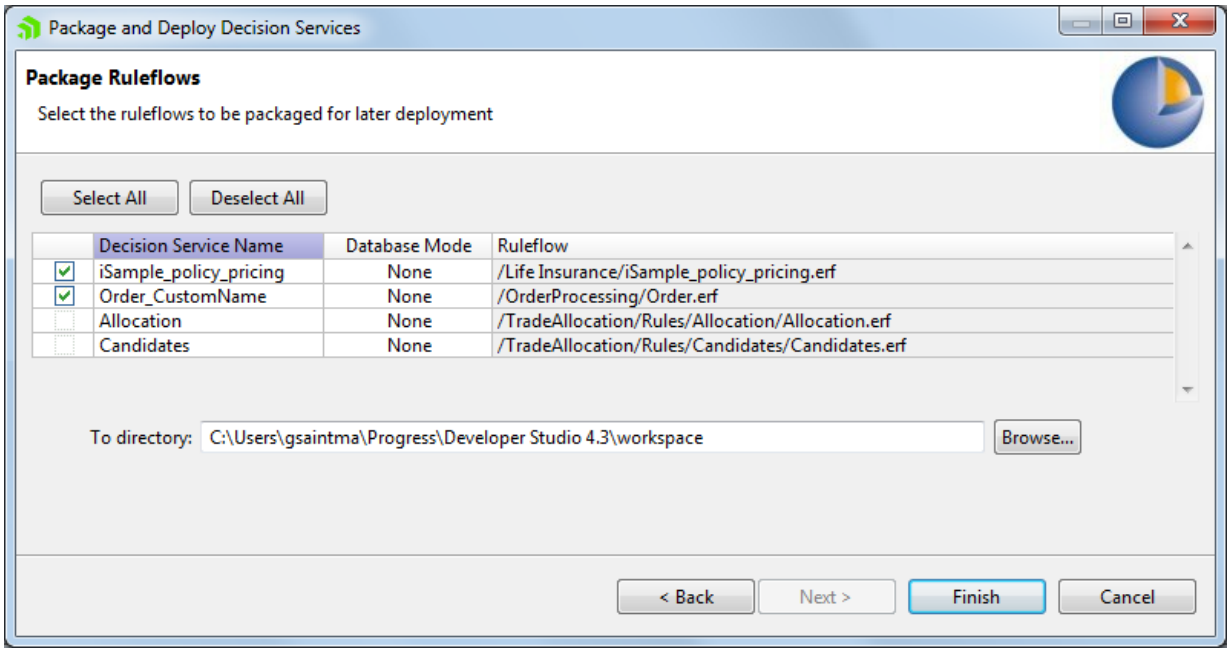
   The packaging and deployment progress is shown. It can be stopped (although what has been completed is not backed out) by clicking the **Stop** button adjacent to the progress bar.

When all the packaging and deployment processes are successful, the wizard alerts you with a **Deployment Success** message. If there are problems, the wizard lists the errors.

Compiling and saving to Studio disk for later deployment

When you choose to package and saver for later deployment, the wizard lists the Ruleflows selected to compile and save to local storage.

The **Package Ruleflows** panel lists the Ruleflows in the context you specified:

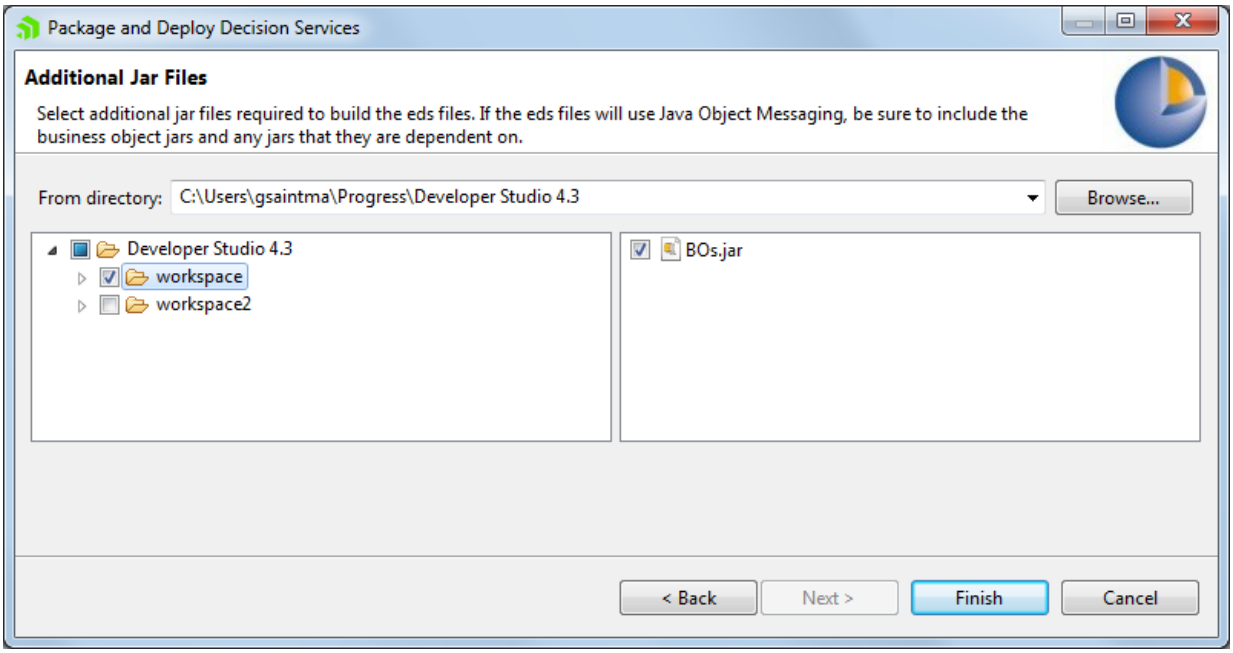


The columns on this panel show the following about each Ruleflow:

- The wizard copies the **Decision Service Name** from the Ruleflow file name.
- **Database mode** for a Ruleflow that will have a database connection.
- **Ruleflow** location within the current workspace.

1. You can change any **Decision Service Name** to save the Decision Service with a preferred name.
2. Click the selection box for each Ruleflow to be packaged and stored at a network-accesible disk location as a Decision Service.
3. In the **To directory** entry area, either enter or browse to a folder location where the packaged Decision Services will be saved.
4. Click **Next**.

   The **Additional Jar Files** panel opens:



5. If additional JAR files are needed to build the Decision Service files (such as business object JARs and their dependent JARs), navigate in the **From directory** entry area to enter or browse to the common folder location of these required JARs, and then select the subdirectories and the JAR files in those locations.
6. Click **Finish**.

   The packaging and save progress is shown. It can be stopped (although what has been completed is not backed out) by clicking the **Stop** button adjacent to the progress bar.

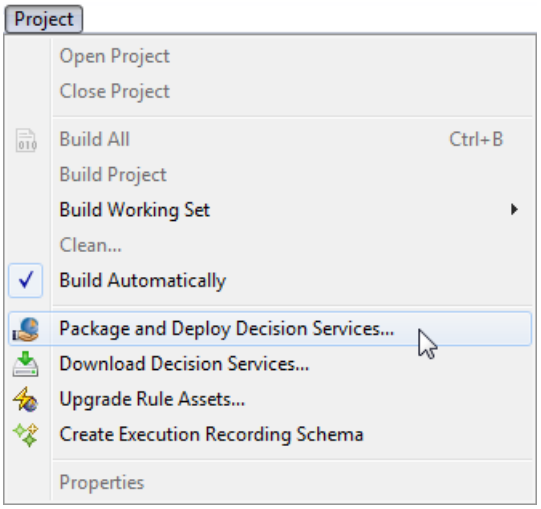
When the processes are successful, the wizard alerts you with a **Deployment Success** message. If there are problems, the wizard lists the errors.

Deploying Decision Services into Web Console Applications from Studio

You can deploy from Studio to servers managed by a Web Console. While the Studio's Publish wizard enables compiling a Ruleflow into a Decision Service to be staged locally or deployed to a running Server, this feature enables deploying one or more Ruleflows into an Application on a Web Console server. Servers and server groups that are hosting the application immediately deploy (or redeploy) the Decision Services to all running servers.

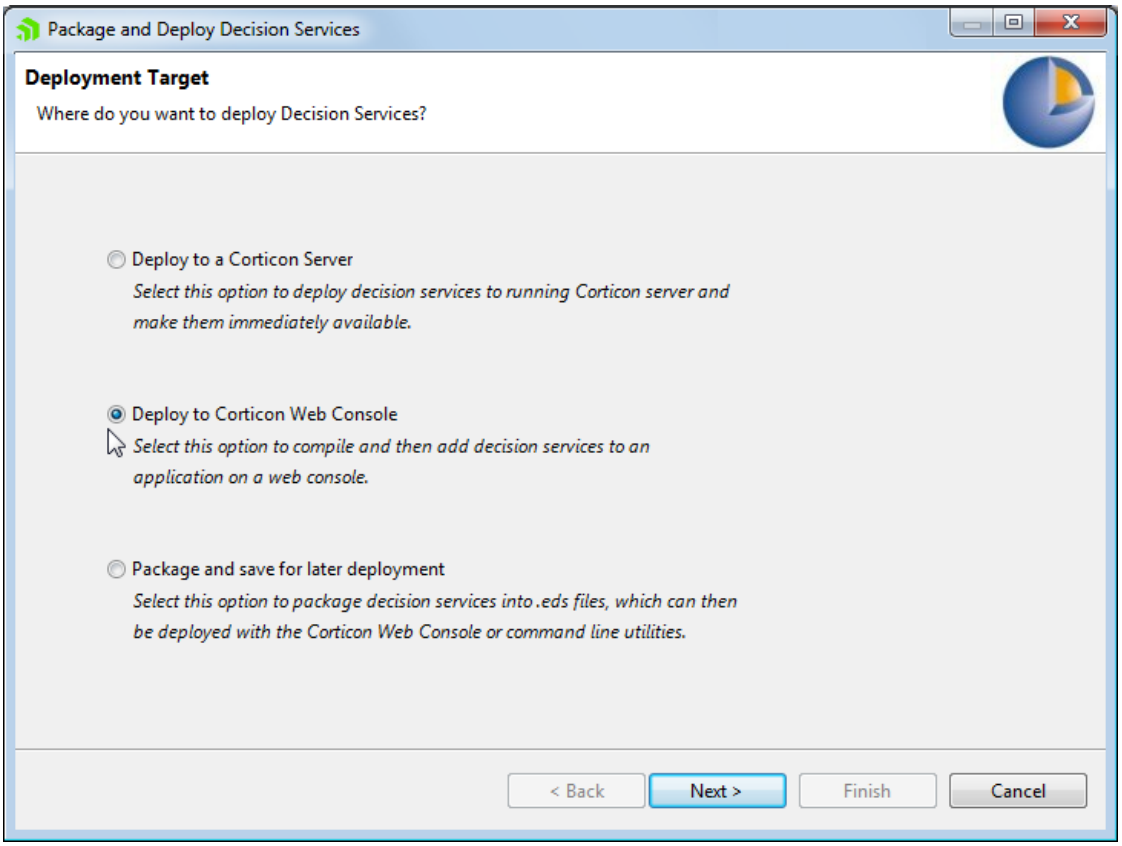
To deploy Ruleflows in Corticon Studio as Decision Services on servers managed by the Web Console:

1. Confirm that the Web Console server you want to use is running. Also confirm that the servers that will run the deployed Decision Services are running.
2. In Corticon Studio, choose **Project > Package and Deploy Decision Services**:

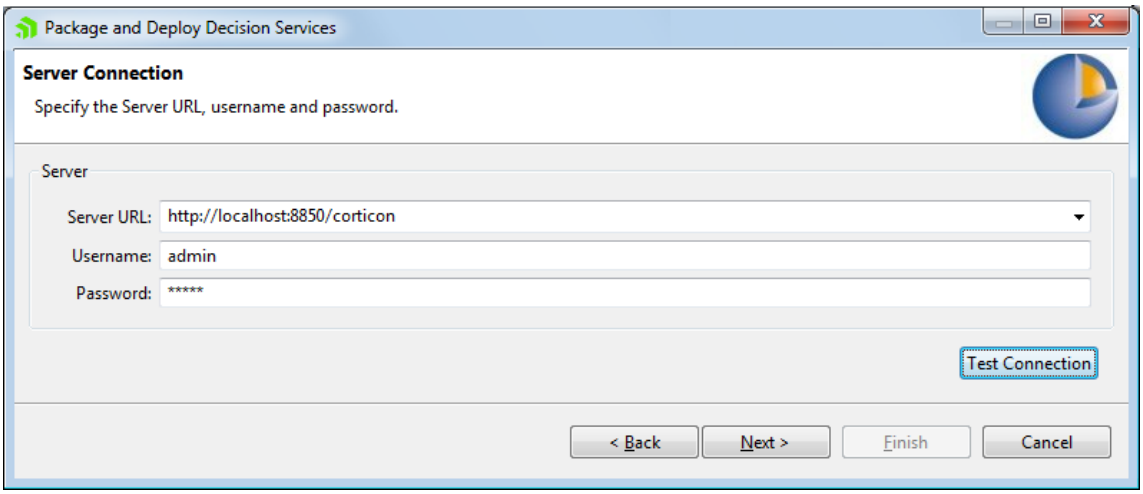


When a project is selected or there is an active file in its editor, the Ruleflows of only that project will be listed. When no projects are selected and no files are in their editor, the Ruleflows of all projects in the workspace will be listed.

3. In the **Package and Deploy Decision Services** dialog, choose the deployment target **Deploy to Corticon Web Console**.

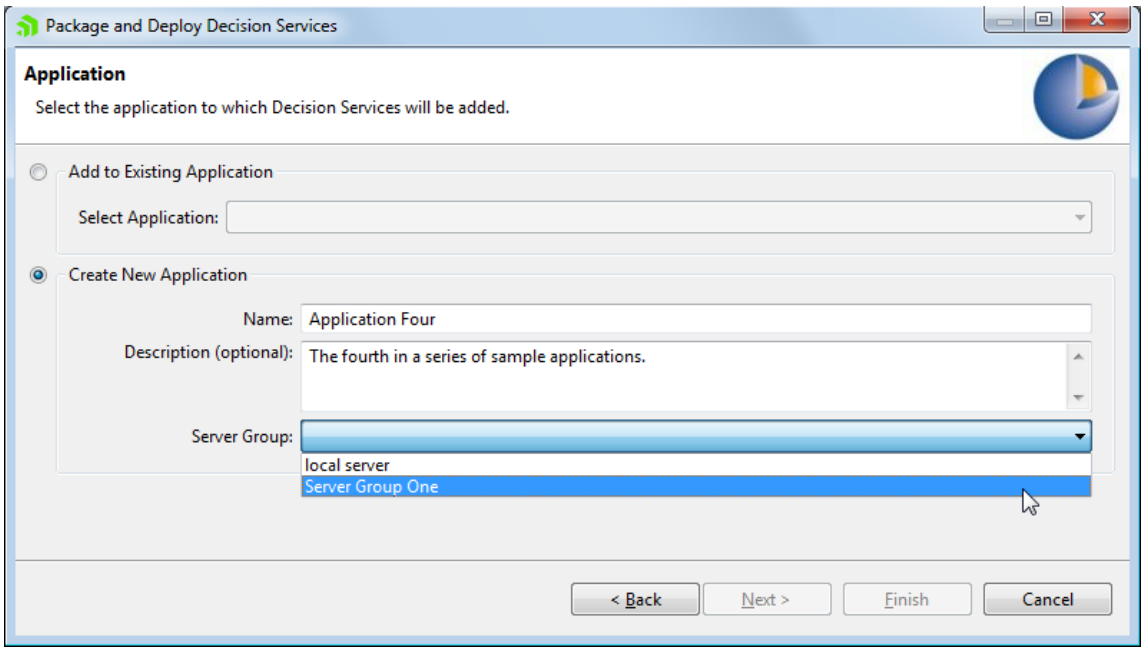


4. Click **Next**.
5. Enter the server connection URL with its port and `/corticon`, then the username and password for that Web Console. The administrative username is `admin` with the initial password `admin`.



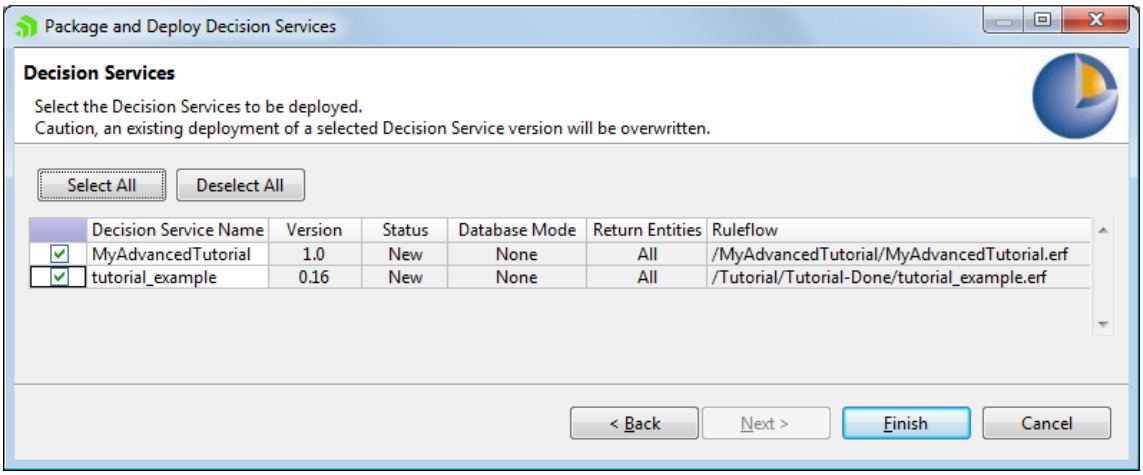
The connection information is persisted locally, so that it can be offered for subsequent publishing to known Web Console locations.

6. Select whether to use an existing Application or to create a new one:
   - To add to an existing Application, choose **Add to Existing Application**, select an Application on the pull-down list, and then click **Next**.
   - To create a new Application, choose **Create New Application**, and then enter a new Application name and its description.



In the **Server Group**'s dropdown menu, choose the server or server group that will host the Application, and then click **Next**.

7. The **Decision Services** panel opens:



Select the Ruleflows to deploy as Decision Services. You can edit the **Decision Service Name** to make it a distinct deployment even though the same Ruleflow Version might already be deployed under another name.

Note: When deploying EDC-enabled Decision Services you must set Database Mode to **Read Only** or **Read/Update** for the Decision Services to access the database once deployed. Your Corticon Server must be licensed for EDC.

When your selections are complete, click **Finish**.

The wizard then compiles the Ruleflows locally, creates a new Application (if required) on the Web Console, and then adds (or updates) the Decision Services in the Application. Then, the Application is updated automatically to deploy/update the Decision Services in Servers and all active server members in Server Groups hosting the Application.

Using Web Console to deploy Decision Services

You can use features in the Corticon Web Console to deploy and manage Decision Services from a browser. For more information, see the topic *"Decision Services and Applications"* in the Corticon Server: Web Console Guide.

Using Deployment Descriptors to deploy Decision Services

A Deployment Descriptor file is an XML text file that identifies one or more Ruleflows or Decision Services to be deployed, and the properties to be set on the Decision Service.

You can create CDD files with either an XML editor or the Server Deployment Console.

When you specify Ruleflows in a CDD file, the Corticon Server performs the compilation of the Ruleflow to create the Decision Service to be deployed. When you precompile Decision Services (`.eds`) files and specify them in the CDD, the overhead of compilation does not impact system performance -- this is highly recommended for production deployments.

To get a taste for authoring a CDD file, create one in the Server Deployment Console, and then open the `.cdd` file in a text editor to see how it is formatted. You will readily see how the parameter names correspond to fields in the Deployment Console. You can then add other properties that are not produced through the Deployment Console.

Structure of a Deployment Descriptor (.cdd) file

The following code segment shows the general pattern of two Decision Services in a CDD file:

```
<cdd soap_server_binding_url="http://localhost:8850/axis/services/Corticon">
  <decisionservice>
     <name>[Name1]</name>
     <path>[Path1.erf]</path>
     <options>
        <option name="" value="">
              .
              .
     </options>
  </decisionservice>

  <decisionservice>
      <name>[Name2]</name>
      <path>[Path2.eds file]</path>
      <options>
         <option name="" value="">
               .
               .
      </options>
  </decisionservice>

        ...
</cdd>
```

Notice that the first `decisionservice` is a Ruleflow (.erf) file while the second is an unrelated `decisionservice` that was precompiled from a Ruleflow into a Decision Service (.eds) file. You could add several more `decisionservice` sections to the CDD file.

Setting properties in a CDD file

When deploying with Corticon Deployment Descriptor (CDD) files, you might want to set deployment properties, such as controlling rule messages, in the CDD file so that the CDD fully describes the deployment configuration.

The properties for CDD file are set in name-value pairs, as shown:

```
<option name "name1" value="value1">
<option name "name2" value="value2">
```

The valid options in a CDD file and their values are as follows (each applicable default value is underlined):

Option name	Value
PROPERTY_AUTO_RELOAD	false | true
PROPERTY_MAX_POOL_SIZE	1 | [positive integer]
PROPERTY_MESSAGE_STRUCTURE_TYPE	<null> | HIER | FLAT
PROPERTY_DATABASE_ACCESS_MODE	<null> | R | RW

Option name	Value
PROPERTY_DATABASE_ACCESS_RETURN_ENTITIES_MODE	ALL | IN
PROPERTY_DATABASE_ACCESS_PROPERTIES_PATH	[explicit or relative path]
PROPERTY_DATABASE_ACCESS_CACHING_ENABLED	false (default) | true
The following properties can be set as overrides in the server's brms.properties file, all of which are set there to false.	
PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_INFO	false | true
PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_WARNING	false | true
PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_VIOLATION	false | true
PROPERTY_EXECUTION_RESTRICT_ RESPONSE_TO_RULEMESSAGES_ONLY	false | true

Note: The path names to the Ruleflow (`.erf`) and Decision Service (`.eds`) files can be expressed *relative* to the location of the Deployment Descriptor file (indicated by the `../` syntax). That's a good practice, as the explicit path on deployment Servers might be different. These paths can be edited if changes are required. If the saved location of the Deployment Descriptor file has its path in common with the location of the Ruleflow (`.erf`) or Decision Service (`.eds`) file, then the path is typically expressed in relative terms. If the two locations have no path in common (for example, they are saved to separate machines), then the path must be expressed in absolute terms. UNC paths can also be used to direct Corticon Server to look in remote directories.

Setting deployment properties in a CDD file through APIs

To deploy a Decision Service using APIs, at a minimum you have to supply a Decision Service Name and a Rule Asset Path. The Properties object can contain additional options related to that Decision Service. Any properties you do not specify will assume default values.

You define the properties to set using the `ICcServer` API's `deployDecisionService(...)` method in the form:

```
void deployDecisionService(String astrDecisionServiceName, String astrRuleAssetPath,
Properties apropDeploymentOptions)
```

which then uses the `ICcServer` API's `addDecisionService(...)` method in the form:

```
void addDecisionService(String astrDecisionServiceName, String astrRuleAssetPath,
Properties apropDeploymentOptions)
```

Valid properties for a Decision Service

The **Corticon Server API** Javadoc for `ICcServer` has constants defined for each property. It is a good practice to use those constants instead of literal values, as will be demonstrated.

The following list of properties (all are `public static final String`) can be set for this method:

- `PROPERTY_AUTO_RELOAD = "PROPERTY_AUTO_RELOAD";`

- PROPERTY_MAX_POOL_SIZE = "PROPERTY_MAX_POOL_SIZE";
- PROPERTY_MESSAGE_STRUCTURE_TYPE = "PROPERTY_MESSAGE_STRUCTURE_TYPE";
- PROPERTY_DATABASE_ACCESS_MODE = "PROPERTY_DATABASE_ACCESS_MODE";
- PROPERTY_DATABASE_ACCESS_RETURN_ENTITIES_MODE = "PROPERTY_DATABASE_ACCESS_RETURN_ENTITIES_MODE";
- PROPERTY_DATABASE_ACCESS_PROPERTIES_PATH = "PROPERTY_DATABASE_ACCESS_PROPERTIES_PATH";
- PROPERTY_DATABASE_ACCESS_CACHING_ENABLED = "PROPERTY_DATABASE_ACCESS_CACHING_ENABLED";
- PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_INFO = "PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_INFO";
- PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_WARNING = "PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_WARNING";
- PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_VIOLATION = "PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_VIOLATION";
- PROPERTY_EXECUTION_RESTRICT_RESPONSE_TO_RULEMESSAGES_ONLY = "PROPERTY_EXECUTION_RESTRICT_RESPONSE_TO_RULEMESSAGES_ONLY";

Example

The following example use these methods to deploy a Decision Service using the generic API on ICcServer:

```
public void deployDecisionService() throws Exception
   {
      String lstrDecisionService = "TestDeploy";
      String lstrEdsFilePath = "c:/Temp/MyDS.eds";

      Properties lpropOptions = new Properties();

      lpropOptions.put(ICcServer.PROPERTY_
           AUTO_RELOAD,
           Boolean.TRUE);
      lpropOptions.put(ICcServer.PROPERTY_
           MAX_POOL_SIZE,
           2);
      lpropOptions.put(ICcServer.PROPERTY_
           MESSAGE_STRUCTURE_TYPE,
           ICcServer.XML_STYLE_AUTODETECT);
      lpropOptions.put(ICcServer.PROPERTY_
           DATABASE_ACCESS_MODE,
           ICcServer.DATABASE_ACCESS_READ_WRITE);
      lpropOptions.put(ICcServer.PROPERTY_
           DATABASE_ACCESS_RETURN_ENTITIES_MODE,
           ICcServer.DATABASE_ACCESS_RETURN_ALL_ENTITY_INSTANCES);
      lpropOptions.put(ICcServer.PROPERTY_
           DATABASE_ACCESS_PROPERTIES_PATH,
           "c:/Temp/dbconnect.properties");
      lpropOptions.put(ICcServer.PROPERTY_
           DATABASE_ACCESS_CACHING_ENABLED,
           Boolean.TRUE);
      lpropOptions.put(ICcServer.PROPERTY_
           EXECUTION_RESTRICT_RULEMESSAGES_INFO,
           Boolean.FALSE);
      lpropOptions.put(ICcServer.PROPERTY_
           EXECUTION_RESTRICT_RULEMESSAGES_WARNING,
           Boolean.FALSE);
      lpropOptions.put(ICcServer.PROPERTY_
           EXECUTION_RESTRICT_RULEMESSAGES_VIOLATION,
```

```
            Boolean.FALSE);
        lpropOptions.put(ICcServer.PROPERTY_
            EXECUTION_RESTRICT_RESPONSE_TO_RULEMESSAGES_ONLY,
            Boolean.FALSE);

        ICcServer lICcServer = CcServerFactory.getCcServer();

        //  Add Decision Service to ICcServer
        lICcServer.addDecisionService(lstrDecisionService,
                                      lstrEdsFilePath,
                                      lpropOptions);
    }
```

Compiling a CDD using APIs

The API methods `loadFromCdd()` and `loadFromCddDir()` compile one or several Deployment Descriptors. The simplest, `loadFromCdd()`, requires you to provide the complete path to the **specific** Deployment Descriptor file you want to load. The other, `loadFromCddDir()`, requires the path to the directory where Corticon Server will look and load **all** Deployment Descriptor files it finds there.

These methods are summarized in the Java API Summary in Corticon API reference on page 251 and described fully in the Corticon Server Javadoc.

Example of a complete CDD file

The first Decision Service in this CDD shows all options and the second accepts all defaults.

```
<cdd soap_server_binding_url="http://localhost:8850/axis/services/Corticon">
   <decisionservice>
      <name>AllocateTrade</name>
      <path>../AllocateTrade.eds</path>
      <options>
         <option name="PROPERTY_AUTO_RELOAD"
                  value="true" />
         <option name="PROPERTY_MAX_POOL_SIZE"
                  value="1" />
         <option name="PROPERTY_MESSAGE_STRUCTURE_TYPE"
                  value="HIER" />
         <option name="PROPERTY_DATABASE_ACCESS_MODE"
                  value="R"  >
         <option name="PROPERTY_DATABASE_ACCESS_RETURN_ENTITIES_MODE"
                  value="ALL" />
         <option name="PROPERTY_DATABASE_ACCESS_PROPERTIES_PATH"
                  value="../MyDBaccess.txt" />
         <option name="PROPERTY_DATABASE_ACCESS_CACHING_ENABLED"
                  value="true" />
         <option name="PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_INFO"
                  value="true" />
         <option name="PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_WARNING"
                  value="true" />
         <option name="PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_VIOLATION"
                  value="true" />
         <option name="PROPERTY_EXECUTION_RESTRICT
                                    _RESPONSE_TO_RULEMESSAGES_ONLY"
                  value="true" />
      </options>
   </decisionservice>
   <decisionservice>
      <name>Candidates</name>
      <path>../Candidates.erf</path>
      <options/>
   </decisionservice>
</cdd>
```

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

- There are two `<decisionservice>` sections, one for a pre-compiled Decision Service, and one for a Ruleflow.
- The first `<decisionservice>` specifies that it uses EDC database access by chosing the value `R`, the Read-Only setting, and the database related entities returned option, the option to enable database caching, and the location of the Database Access Properties file that defines the database connection.

Important: If you are using the bundled Progress Application Server to test and deploy your Ruleflows, copy the Deployment Descriptor file to the Corticon Server installation's `[CORTICON_WORK_DIR]\cdd` directory. When Corticon Server starts, it reads all `.cdd` files in that default location.

Updating and extending older CDD files

If you have CDD files created in releases before 5.5.1, they will continue to perform and deploy as expected. You can use the Deployment Console to open such CDD files and the save them -- the CDD file is reformatted with all the options you had set. You can then edit the CDD in a text editor to add in properties that were not previously available through the Deployment Console.

Using the Server Deployment Console

The Corticon Deployment Console provides a graphical interface to help you package Decision Services for deployment. This packaging can also be achieved with the `compile` corticonManagement command line utility. The Deployment Console does not actually deploy Decision Services, it just makes them ready to deploy.

The Deployment Console has two functions:

- **Create Deployment Descriptor (.cdd) files**, the XML documents that instruct Corticon Server which Ruleflows and Decision Services to load, and how to configure appropriate parameters and settings for each.
- **Create XML service contract documents**, the files used for Ruleflow integration. These are discussed in topics at Integrating Corticon Decision Services on page 63.

To start the Corticon Deployment Console:

- **Java Server:** On the Windows Start menu, choose **Programs** > **Progress** > **Corticon n.n** > **Corticon Deployment Console**.
- **.NET Server:** On the Windows Start menu, choose **Programs** > **Progress** > **Corticon n.n** > **Corticon .NET Deployment Console**.

Note: For Linux installations, the Deployment Console is included in the server archive zip file. See its readme for information on how to run it.

Functions of the Deployment Console tool

The Deployment Console is divided into two sections. 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

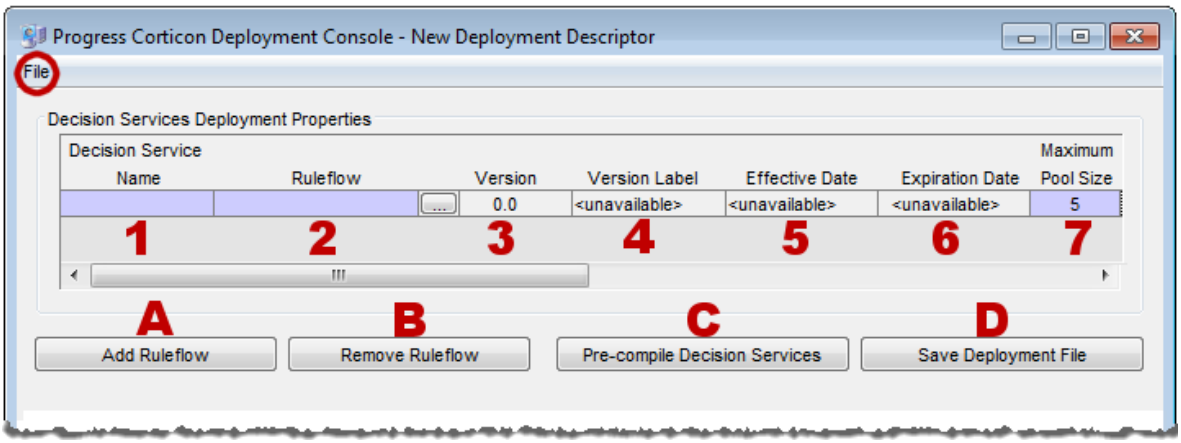
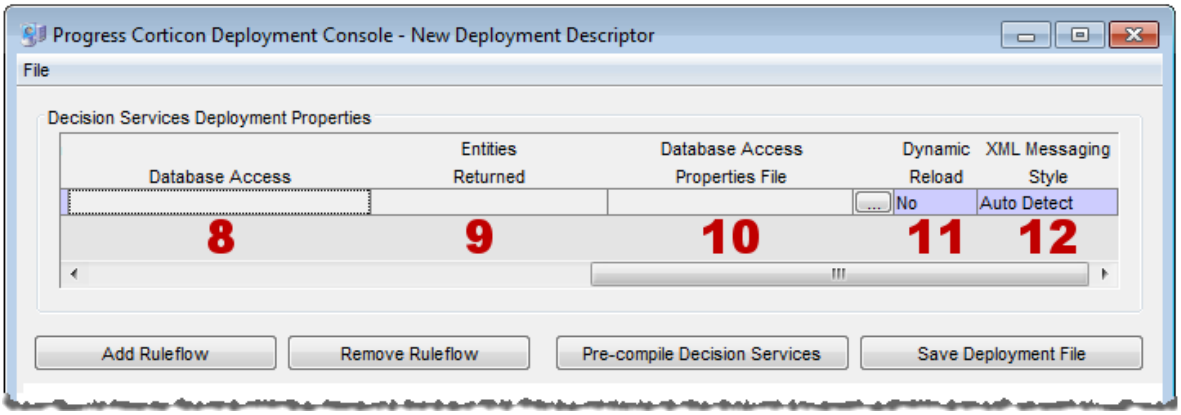


Figure 28: Right Portion of Deployment Console, with Deployment Descriptor File Settings Numbered



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 Ruleflows 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. **Maximum Pool Size** - Specifies how many execution threads for this Decision Service will be added to the Execution Queue. This parameter is an issue only when Allocation is turned on. If you are evaluating Corticon, your license requires that you set the parameter to `1`. See *'Multi-threading, concurrency reactors, and server pools' in "Inside Corticon Server" section of the Integration and Deployment Guide* for more information.

Note: Minimum Pool Size, previously associated with this property, is deprecated as of version 5.5.

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

8. **Database Access** - *Active if your Corticon license enables EDC* - 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.
9. **Entities Returned** - *Active if your Corticon license enables EDC* - 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**).
10. **Database Access Properties File** - *Active if your Corticon license enables EDC* - 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.

11. **Dynamic Reload** - When **Yes**, the `ServerMaintenanceThread` will detect if the Ruleflow or `.eds` file has been updated; if so, the Decision Service will be updated into memory and -- for any subsequent calls to that Decision Service -- that execution Thread will execute against the newly updated Rules. When **No**, the `CcServerMaintenanceThread` will ignore any changes to the Ruleflow or `.eds` file. The changes will not be read into memory, and all execution Threads will execute against the existing Rules that are in memory for that Decision Service.
12. **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 Ruleflows 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 Ruleflows *until* they are deployed to Corticon Server. Here, you choose to pre-compile Ruleflows 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.)

Setting the autoloaddir property

The `autoloaddir` property is set by default in the `CcServer.properties -> com.corticon.ccserver.autoloaddir` property. By default, the value is set to `[CORTICON_WORK_DIR]/cdd` in the form:

```
com.corticon.ccserver.autoloaddir=%CORTICON_WORK_DIR%/cdd
```

where `%CORTICON_WORK_DIR%` is the absolute path of the work directory, typically `C:/Users/{username}/Progress/Corticon x.x`

If you decide you would rather specify your own directory, then add the `com.corticon.ccserver.autoloaddir` property with your preferred value to the `brms.properties` file. This will be used to override the default.

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

Using command line utilities to compile Decision Services

Users wanting to automate the building and testing of Decision Services can use the `corticonManagement` utility to compile Ruleflows into Decision Services ready for deployment, create XSD and WSDL files for clients who will call the Decision Services, and to run Ruletests to validate that the Decision Services perform as expected.

The commands can be used to script these processes and integrate them with other automated processes such as the "build" procedure for a larger project. To make this integration easier, a set of ANT macros are provided that make it easy to perform the building and testing of Decision Services within a custom ANT build script.

Note: When the target for deployment is the Corticon Server for .NET, you can use the `corticonManagement` utilities and ANT scripts to build `.eds` files and run tests. Then, running the IKVM utilities against the `.eds` file will generate the .NET bytecode.

Syntax of the compile and test commands

The `corticonManagement` utility is located at `[CORTICON_HOME]\Server\bin`. It can be run by choosing **Start > All Programs > Progress > Corticon 5.6 > Corticon Command Prompt** and then typing `corticonManagement` to display its usage:

Table 3: usage: corticonManagement

Argument	Description
`-c,--compile`	Compile a Ruleflow into a Decision Service
`-e,--extractDiagnostics`	Extract diagnostic data from a log file
`-h,--help`	Print this message
`-m,--multicompile`	Compile Ruleflows in the specified input file
`-s,--schema`	Generate the WSDL/XSD schema for a vocabulary or Ruleflow
`-t,--test`	Execute tests for a Ruleflow or Rulesheet

To use `corticonManagement` in a script you need to add the Corticon `bin` folder to your `PATH` environment variable: `set PATH=%PATH%;[CORTICON_HOME]\Server\bin`

Note: This section discusses all these options except `extractDiagnostics`. See Diagnosing runtime performance of server and Decision Services on page 184 for complete information on **diagnostic** data.

Compiling a Decision Service from a Ruleflow

The `compile` option compiles a Ruleflow into a Decision Service .eds file that can then be deployed to a Corticon Server through the Web Console, a `.cdd` file, or other supported tools.

Table 4: usage: corticonManagement --compile

Argument	Description
`-i,--input` *`file`*	Required. The source Ruleflow `.erf` file to be compiled.

Argument	Description
`-o,--output` *`folder`*	Required. Explicit path to the output folder. If the folder does not exist, it is created.
`-s,--service` *`name`*	Required. The Decision Service file name. (Do not add the .eds extension, it will be done for you.)
`-v, --version`	The major and minor version for the Decision Service as specified on the Ruleflow is appended to the .eds file name in the output folder as *`service_vversionMajor_versionMinor`*`.eds`.
`-e,--edc [`*`R`*`|`*`RW`*`]`	Required when the Vocabulary has been mapped to a database. Sets the database access mode (read only or read write).
`-dj,--dependentjars` *`dependentjar,..`*	Required when using extensions. Explicit path to JAR files required for this Decision Service.
`-ij,--includedjars` *`includedjar,..`*	Required when using extensions. Explicit path to JAR files (that are specified as `dependentjars`) to include in the generated EDS file.

A complete command might look like this:

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

Testing a Decision Service with a Ruletest

The `test` option executes one or more test sheets in a Corticon ruletest (.ert) file, and produces an output file with the test results

Table 5: usage: corticonManagement --test

Argument	Description
`-i,--input` *`file`*	Required. The source Ruletest `.ert` file to run.
`-o,--output` *`file`*	Required. Explicit path to the preferred output folder and file name (an .xml format file in the JUnit test output style.) The output file is not overwritten if it exists, instead the new test output is appended after test execution, thus enabling multiple executions of different test sets to log their output into a single report file.

Argument	Description
`-a,--all`	Required unless `--sheet` is stated. Runs tests for all the testsheets in the specified Ruletest in the order that they are defined in the file. Overrides any specific testsheets listed in the sheet option.
`-s,--sheet` *`sheet_names`*	Required unless `--all` is stated. Runs tests for only the one or more (in a comma-separated list) specified testsheets in the Ruletest in the order that they are listed.
`-ll,--loglevel` *`level`*	Sets the log level to the specified level of detail. Defaults to current server log level, typically `INFO`.
`-lp,--logpath` *`path`*	Explicit path to the folder where `CcServer.log` will be saved. Defaults to the server's current log location, typically `[CORTICON_WORK_DIR]/logs`.

Example usage:

```
corticonManagement --test
                   --input C:\MyTest.ert
                   -a
                   -o C:\MyTest_out.xml

corticonManagement -t
                   -i C:\MyTest.ert
                   -o C:\MyTest_out.xml
                   -s sheet1,sheet2,sheet3
```

Generating XSD and WSDL schema files

The `schema` option generates XSD and WSDL schema files from either a Ruleflow (.erf) or Vocabulary (.ecore) file. This is the same functionality that DeploymentConsole provides.

Table 6: usage: corticonManagement --schema

Argument	Description
`-i,--input` *`file`*	Required. The source Vocabulary (.ecore) file for a vocabulary-level schema, or Ruleflow (.erf) file for a Decision Service-level schema.
`-o,--output` *`folder`*	Required. Explicit path to the output folder.
`-m,--messagestyle [`*`FLAT`*`|`*`HIER`*`]`	Optional. Specifies whether the messaging style should be flat, hierarchical. When omitted, defaults to auto-detect where the schema generator determines the best option.
`-s,--service` *`name`*	Required for a Ruleflow. The name of the Decision Service for the schema.

Argument	Description
`-t,--type [WSDL|XSD]`	Required. Type of schema to generate.
`-u,--url address`	Required. Specifies the Server URL to set in the schema document. This URL should match the URL of the Decision Service when deployed so that clients using the schema have the correct URL. server URL to substitute in WSDL schema.

Example usage:

```
corticonManagement --schema
                    -i C:\myRuleflow.erf
                    -t WSDL
                    -m HIER
                    -u http://myserver:5555/myservice
                    -o C:\Output
                    -s C:\MyDS


corticonManagement  -s
                    -i C:\MyVocab.ecore
                    -t XSD
                    -u http://myserver:5555/myservice
                    -o C:\Output
```

Compiling multiple Decision Services

Using the Multiple Compilation feature, you can compile multiple Decision Services using directives specified in an XML file.

Table 7: usage: corticonManagement --multicompile

Argument	Description
`-i,--input file`	XML file containing directives for Ruleflow (.erf files) to compile

Example usage:

```
corticonManagement --multicompile
                    -i C:\precompile.xml
```

Template

The following template, provided as `[CORTICON_HOME]\Server\bin\multipleCompilation.xml`, presents the settings for the logs and the pattern for each of several Ruleflows to compile:

```
<MultipleCompilation>
   <CompilationLogDirectory>
    **Fully qualified path to directory
      where log will be placed**
      </CompilationLogDirectory>
   <CompilationObjects>
   <CompilationObject>
   <DecisionServiceName>
```

```
        **Name of the Decision Service**
        </DecisionServiceName>
    <RuleflowPath>
        **Explicit path to the Ruleflow to compile**
        </RuleflowPath>
    <OutputDirectory>
        **Explicit path to output directory for the  .eds file**
        </OutputDirectory>
    <OverrideIfExists>
        **true/false: Determines whether to
          overwrite a matching file in the output directory**
        </OverrideIfExists>
    <DatabaseAccessMode>
     **empty value/R/RW: Determines if and
       how the Rules will be compiled for EDC compatibility**
     </DatabaseAccessMode>
   </CompilationObject>

   <CompilationObject>
    ...
   </CompilationObject>
  </CompilationObjects>
 </MultipleCompilation>
```

The following example of `multipleCompilation.xml` specifies two Ruleflows to compile, each as its own Decision Service.

```
<MultipleCompilation>
 <CompilationLogDirectory>C:\Corticon\Compilation_logs</CompilationLogDirectory>
 <CompilationObjects>
  <CompilationObject>
   <DecisionServiceName>Cargo</DecisionServiceName>
   <RuleflowPath>C:\Corticon\staging\Ruleflows\cargo.erf</RuleflowPath>
   <OutputDirectory>C:\Corticon\staging\DecisionServices</OutputDirectory>
   <OverrideIfExists>true</OverrideIfExists>
   <DatabaseAccessMode>RW</DatabaseAccessMode>
  </CompilationObject>
  <CompilationObject>
  <CompilationObject>
   <DecisionServiceName>GroceryStore</DecisionServiceName>
   <RuleflowPath>C:\Corticon\staging\Ruleflows\grocery.erf</RuleflowPath>
   <OutputDirectory>C:\Corticon\staging\DecisionServices</OutputDirectory>
   <OverrideIfExists>true</OverrideIfExists>
   <DatabaseAccessMode></DatabaseAccessMode>
  </CompilationObject>
  <CompilationObject>
  </CompilationObject>
 </CompilationObjects>
</MultipleCompilation>
```

Once the compilation objects are defined, launching `multipleCompilation.bat` compiles each of the Ruleflows into its target Decision Service.

Deploying a Decision Service

You can make scripted calls to the Web Console on Windows and Linux to deploy Decision Services. This combined with ability to build and test Decision Services from a script allow you to automate the deployment of your Decision Services. The scripting is a command line utility that makes REST calls to the Web Console to perform actions, and then returns codes that identify success or failure.

The utility, `corticonWebConsole.bat`, is included on all server installations, and is located at `[CORTICON_HOME]/Server/bin`. The general format of this utility's commands is:

```
corticonWebConsole {command} {command options}
```

Commands supported in this utility are as follows, many parameters showing both their short form and long form:

corticonWebConsole -help

```
usage: corticonWebConsole
 -application,--application   Perform actions on an application
 -ds,--decisionservice        Perform actions on a decision service
 -h,--help                    print this message
 -login,--login               initiate login to web console
 -logout,--logout             initiate logout from web console
```

Lists the syntax of the commands.

corticonWebConsole -login

You must first login to the Web Console before the other commands can be applied.

```
usage: login
 -h,--help                  print this message
 -n,--name <username>       Login username for the web console
 -p,--password <password>   Login password for the web console, if omitted
                            password will be requested through standard
                            input.
 -u,--url <url>             Url leading to the web console e.g
                            http://localhost:8850/corticon
 -v,--verbose               Include debugging output
```

Authenticates the user on the specified Web Console server. No other commands have any effect until this command executes successfully. Choosing to omit the password will prompt for its entry through standard input.

Note: The login command stores an encrypted login token in your work directory so that, when you later perform commands, you can do so without logging in. When using a batch process to perform deployment, you will need to have this login token available in the work directory of the Corticon install used by the batch process.

---

corticonWebConsole -logout

```
usage: logout
 -h,--help      print this message
 -v,--verbose   Include debugging output
```

The logout command closes the connection to the Web Console.

corticonWebConsole -ds

```
usage: decisionservice
 -add,--add          Add a decision service to the web console
 -delete,--delete    Remove a decision service from the web console
 -h,--help           print this message
```

corticonWebConsole -ds -add

```
usage: add
 -application,--application <application name>    Name of application that
                                                  the decision service
                                                  will be added to
 -dbaccessmode,--dbaccessmode <mode>              Database access mode to
                                                  use [optional]
 -dbproperties,--dbproperties <properties file>   Path to database
                                                  properties file
                                                  [optional]
 -dbreturnmode,--dbreturnmode <mode>              Database access return
                                                  entities mode to use
                                                  [optional]
 -deploy,--deploy                                 When this flag is set
                                                  the decision service
                                                  will be deployed after
                                                  being added to the
                                                  application [optional]
 -enablecache,--enablecache                       When this flag is
                                                  present, database
                                                  caching will be enabled
                                                  [optional]
 -f,--file <eds file>                             Path to decision service
                                                  eds file to be added
 -h,--help                                        print this message
 -maxpool,--maxpool <max pool size>               Maximum pool size
                                                  [optional]
 -n,--name <name>                                 Name given to the
                                                  decision service to be
                                                  added
 -overwrite,--overwrite                           When this flag is
                                                  present, the decision
                                                  service will overwrite
                                                  an existing decision
                                                  service in the
                                                  application with a name
                                                  matching that of the
                                                  value of the name
                                                  argument. When this flag
                                                  is not present, and the
                                                  decision service exists
                                                  the deploy will fail.
                                                  [optional]
 -v,--verbose                                     Include debugging output
                                                  [optional]
 -xmlstyle,--xmlstyle <xmlstyle>                  XML message style, ether
                                                  null, FLAT, or HIER
```

Adds the specified Decision Service to the specified application.

The database options (`--dbproperties`, `--dbaccessmode`, and `--dbreturnmode`) are used when the Decision Service is configured for EDC database connectivity.

Adding `--deploy` will deploy the specified Decision Service to each Server or Server Group that includes the specified application.

Adding `--overwrite` to the command will replace a corresponding Decision Service that exists.

corticonWebConsole -ds -delete

```
usage: delete
 -application,--application <application name>   Name of application that
                                                 the decision service will
                                                 be removed from
 -h,--help                                       print this message
 -n,--name <name>                                Name given to the
                                                 decision service to be
                                                 added
 -undeploy,--undeploy                            When this flag is set,
                                                 the decision service will
                                                 be undeployed after being
                                                 removed from the
                                                 application [optional]
 -v,--verbose                                    Include debugging output
                                                 [optional]
```

Removes a specified Decision Service from the Web Console server completely.

Adding `--undeploy` will undeploy the Decision Service from each Server or Server Group that includes the specified application.

corticonWebConsole -application

```
usage: application
 -deploy,--deploy       Deploy the specified application to its associated
                        server/server group
 -h,--help              print this message
 -undeploy,--undeploy   Undeploy the specified application to its
                        associated server/server group
```

corticonWebConsole -application -deploy

```
usage: deploy
 -h,--help          print this message
 -n,--name <name>   Name given to the application to be deployed
 -v,--verbose       Include debugging output [optional]
```

Deploys the specified application to its associated servers/server groups.

corticonWebConsole -application -undeploy

```
usage: undeploy
 -h,--help          print this message
 -n,--name <name>   Name given to the application to be undeployed
 -v,--verbose       Include debugging output [optional]
```

Undeploys the specified application from its associated servers/server groups.

Creating a build process in Ant

Corticon provides Ant macros for the `corticonManagement` command line utilities in the file `[CORTICON_HOME]\Server\lib\corticonAntMacros.xml` . You can download and install Apache Ant, and then add its `/lib` to your global path, and set its `/bin` to `ANT_HOME`.

Note: The Ant process needs to set the environment for `CORTICON_HOME` and `CORTICON_WORK_DIR` so that the macros can locate the necessary libraries and have the scratch location for temporary files. To do this, either start Corticon Command Prompt or just running `corticon_env.bat` before running Ant.

Compile

Arguments for the compile macro:

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

Example of a call to the compile macro:

```
<corticon-compile
    input="${project.home}/Order.erf"
    output="${project.home}"
    service="OrderProcessing" />
```

Multicompile

Arguments for the multiCompile macro:

```
<attribute name="input"  default=""/>
<attribute name="failonerror" default="false"/>
```

Schema

Arguments for the schema macro:

```
<attribute name="input" default=""/>
<attribute name="output" default="" />
<attribute name="service" default="" />
<attribute name="type" default="" />
<attribute name="messagestyle" default="" />
<attribute name="url" default="" />
<attribute name="failonerror" default="false" />
```

Example of a call to the schema macro:

```
<corticon-schema
    input="${project.home}/Order.erf"
    output="${project.home}"
    service="OrderProcessing"
    type="WSDL"
    url="http://localhost:8850/axis"
    messagestyle="HIER"
/>
```

Test

Arguments for the test macro:

```
<attribute name="input" default=""/>
<attribute name="output" default="" />
<attribute name="all" default="false" />
<attribute name="sheet" default="" />
<attribute name="loglevel" default="" />
<attribute name="logpath" default="" />
<attribute name="failonerror" default="false" />
```

Example of a call to the test macro:

```
<corticon-test
    input="${project.home}/Order.ert"
```

```
output="${project.home}/TestResults.xml"
all="true" />
```

Additional properties

```
<property name="corticon.compile.maxmem" value="512m" />
<property name="corticon.compile.permgen" value="64m" />
```

Loading the macros into another build file

You can load the macro file into another build file by using the following `import` syntax:

```
<import file="${env.CORTICON_HOME}/Server/lib/corticonAntMacros.xml " />
```

Using Server API to compile and deploy Decision Services

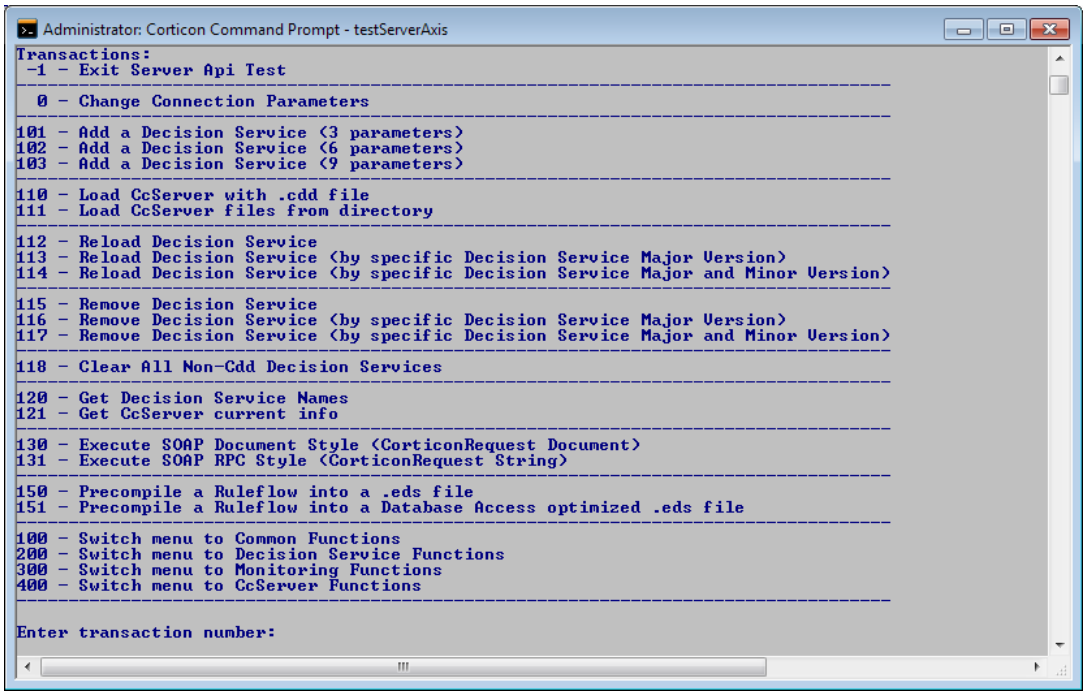
Corticon provides a Java API that can be used in custom code to compile, deploy, and manage Decision Services. The API can be used in code running an in-process Corticon Server, or can be used to manage a remote Corticon Server through the server's SOAP interface.

Using the Test Server Scripts

The test server scripts provide fully-functional commands that are structures in the Server source file `[CORTICON_HOME]\src\CcServerApiTest.java`. For example, to add a Decision Service that uses the Enterprise Data Connector, start the server, and then start the Axis test batch file `[CORTICON_HOME]\Server\bin\testServerAxis.bat`.

Enter `100` to list the 100 series of commands.

Figure 29: testServerAxis.bat 100 commands



Enter `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 known as *auto-reload*)
4. Maximum Pool Size (note that Minimum Pool Size was deprecated in version 5.5. See Multi-threading, concurrency reactors, and server pools on page 147 for more information.
5. Message Structure Type
6. Database Access Mode (<null>, R, RW) If you skip it, the Server defaults to <null> -- EDC for this Decision Service is turned off
7. Return Entities Mode (ALL, IN) If you skip it, the Server defaults to ALL)
8. Database Access Properties Path
9. Database Cache (true, false)

Looking at the snippet source code for `addDecisionService9()`

```
public void addDecisionService9() throws CcException
   {
      InputParameters lInputParameters = new InputParameters();
      lInputParameters.getBaseInformation(1);
      lInputParameters.getRuleAssetPath();
      lInputParameters.getAutoReload();
      lInputParameters.getMinPoolSize();
      lInputParameters.getMaxPoolSize();
      lInputParameters.getMessageStructureType();
      lInputParameters.getDatabaseAccessMode();
      lInputParameters.getDatabaseAccessReturnEntityMode();
      lInputParameters.getDatabaseAccessPropertiesPath();
      lInputParameters.getDatabaseAccessCacheEnabled();

      String lstrResult = null;
      if (ibRunInprocess)
      {
         Object[] lars = lInputParameters.getArgumentList();

         Properties lpropDeploymentOptions = new Properties();

         lpropDeploymentOptions.put
           (ICcServer.PROPERTY_AUTO_RELOAD, (Boolean)lars[2]);
         lpropDeploymentOptions.put
           (ICcServer.PROPERTY_MIN_POOL_SIZE, (Integer)lars[3]);
         lpropDeploymentOptions.put
           (ICcServer.PROPERTY_MAX_POOL_SIZE, (Integer)lars[4]);

         if ((String)lars[5] != null)
            lpropDeploymentOptions.put
              (ICcServer.PROPERTY_MESSAGE_STRUCTURE_TYPE, (String)lars[5]);

         if (lars[6] != null)
         {
            lpropDeploymentOptions.put
              (ICcServer.PROPERTY_DATABASE_ACCESS_MODE,
                 (String)lars[6]);
            lpropDeploymentOptions.put
              (ICcServer.PROPERTY_DATABASE_ACCESS_RETURN_ENTITIES_MODE,
                 (String)lars[7]);
            lpropDeploymentOptions.put
```

```
                (ICcServer.PROPERTY_DATABASE_ACCESS_PROPERTIES_PATH,
                   (String)lars[8]);
             lpropDeploymentOptions.put
                (ICcServer.PROPERTY_DATABASE_ACCESS_CACHING_ENABLED,
                   (Boolean)lars[9]);
          }

          getInprocessCcServer().addDecisionService
            ((String)lars[0], (String)lars[1], lpropDeploymentOptions);
            lstrResult = "COMPLETE";
       }
       else
       {
          String lstrMethodName = "addDecisionService9";
          Object[] lars = lInputParameters.getArgumentList();
          lstrResult = CcMessageHandler.executeRPC
             (istrApacheAxisCorticonAdminUrl, lstrMethodName, lars);
       }

       System.out.println(lstrResult);
    }
```

Using the APIs to compile a Decision Service

The following code snippet is a framework of Java code that uses the method `precompileDecisionService()` in the `ICcServer` interface.

```
package com.progress.corticon.util;

import com.corticon.eclipse.studio.deployment.swing.CcDeployFactory;
import com.corticon.eclipse.studio.deployment.swing.ICcDeploy;

public class Example {

       public static void main(String[] args) {
              // TODO Auto-generated method stub
              String strERFPath = "<path>/<file.erf>";
              String strServiceName = "Example";
              String strOutputPath = "C:/Temp";
              boolean bOverwriteFile = true;

              ICcDeploy d = CcDeployFactory.newDeployment();
              try {
                     d.precompileDecisionService
                        (strERFPath,
                         strServiceName,
                         strOutputPath,
                         bOverwriteFile);
              } catch (Exception e) {
                     // TODO Auto-generated catch block
                     e.printStackTrace();
              }
       }
}
```

For more examples of code that uses Corticon API methods to compile rules, see Corticon Knowledgebase article 57671.

5

Integrating Corticon Decision Services

This section explains how to correctly call or “consume” a Decision Service. Corticon Ruleflows, 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 section, we will not actually make a call to a deployed Decision Service – that is discussed in Invoking Corticon Server on page 85. Instead, this section focuses 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
- Service contracts: Describing the call
- Types of XML service contracts
- Passing null values in messages

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 examples on page 231.

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.

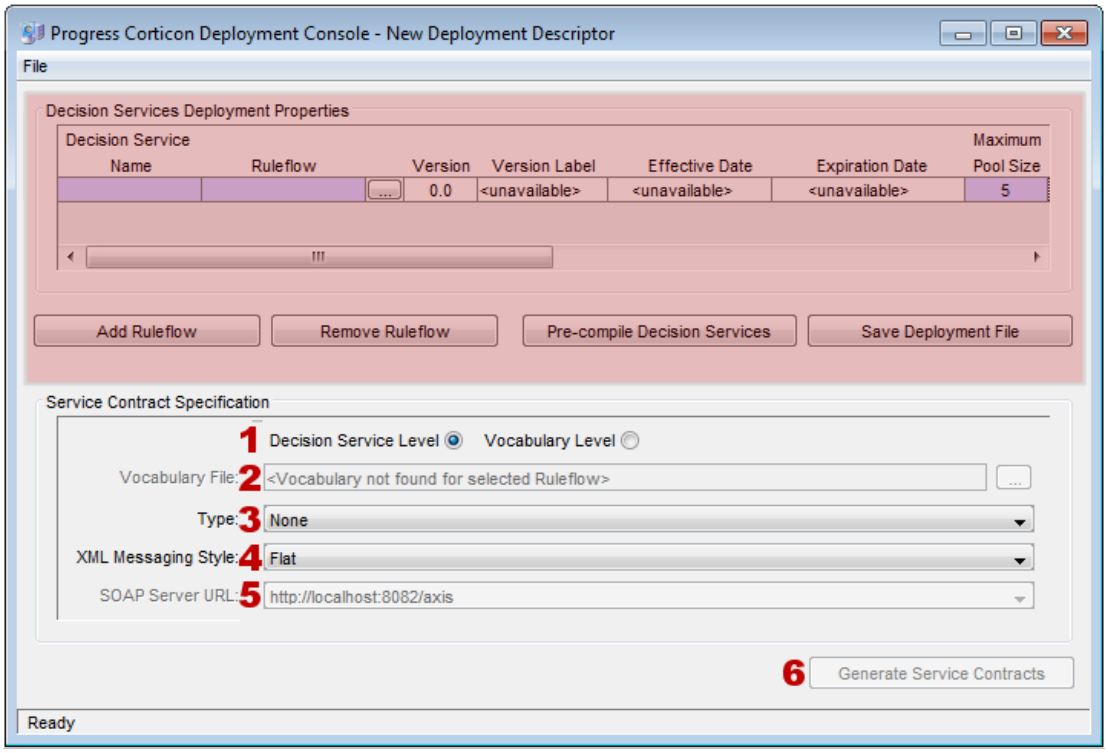
Creating XML service contracts with Corticon Deployment Console

In the topic Using the Server Deployment Console on page 47, the *Deployment Console* is discussed as a way to create Deployment Descriptor files to easily deploy Decision Services and manage their deployment settings. The screenshot in Functions of the Deployment Console tool on page 48 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 section. Now is the time to discuss the lower portion of the *Deployment Console*.

Note: Consider also the command line utility Generating XSD and WSDL schema files on page 53 to achieve the result.

To start the Deployment Console, choose **Start > All Programs > Progress > Corticon 5.x > Corticon Deployment Console**.

Figure 30: 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.

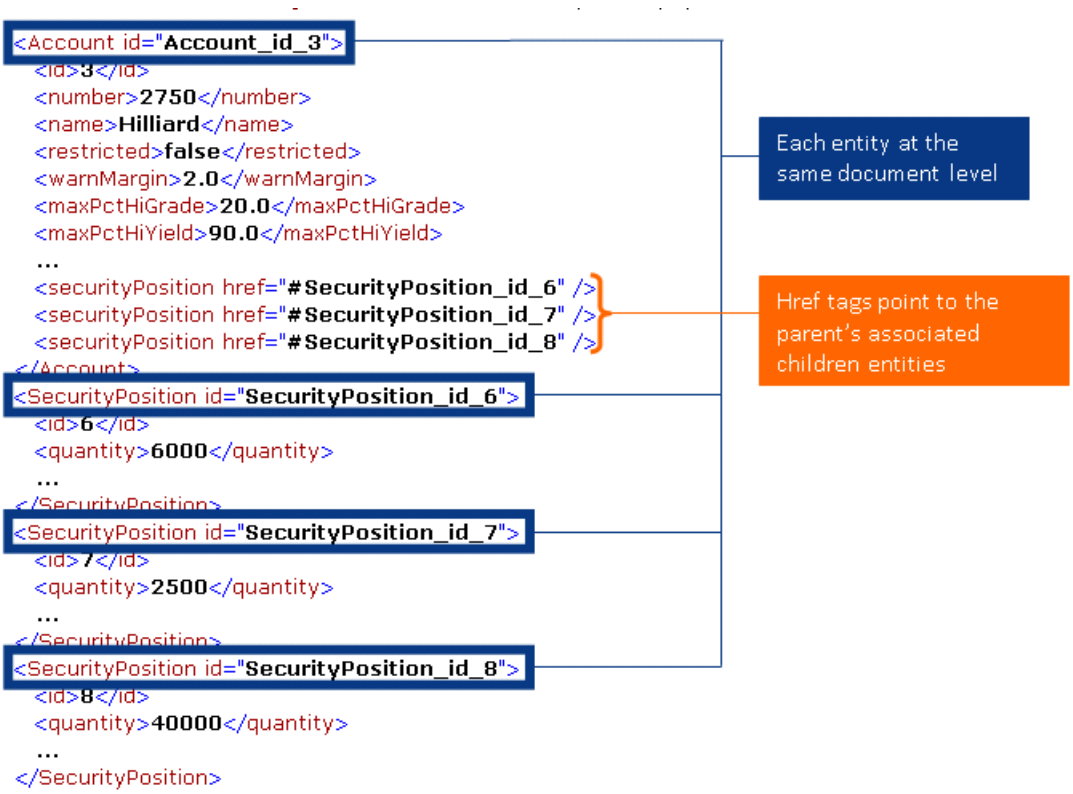
2. **Vocabulary File** - When generating a Vocabulary-level service contract, enter the Vocabulary file name (`.ecore`) here. When generating a Decision Service-level contract, this field is read-only and shows the Vocabulary associated with the currently highlighted Ruleflow row above. See Corticon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

3. **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.

4. **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 31: Example of a Flat (FLAT) Message



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

Figure 32: Example of a Hierarchical (HIER) Message

```
<Account id="Account_id_3">
  <id>3</id>
  <number>2750</number>
  <name>Hilliard</name>
  <restricted>false</restricted>
  <warnMargin>2.0</warnMargin>
  <maxPctHiGrade>20.0</maxPctHiGrade>
  <maxPctHiYield>90.0</maxPctHiYield>
  ...
  <securityPosition id="SecurityPosition_id_6">
    <id>6</id>
    <quantity>6000</quantity>
    ...
  </securityPosition>
  <securityPosition id="SecurityPosition_id_7">
    <id>7</id>
    <quantity>2500</quantity>
    ...
  </securityPosition>
  <securityPosition id="SecurityPosition_id_8">
    <id>8</id>
    <quantity>40000</quantity>
    ...
  </securityPosition>
</Account>
```

Child entity instances are embedded (or indented) under parents via their association role names

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.

5. **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:8850/axis/services/Corticon` that makes a Decision Service available to the Corticon Server's Progress Application Server. This Deployment property's default value can be overridden in your `brms.properties` file.

6. **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:

- 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.

For more information on WSDL, see `http://www.w3schools.com/webservices/ws_wsdl_intro.asp`

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 examples on page 231, including the following variations.

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 messages

REST/JSON

Passing a null value to any Corticon Server using JSON payloads is accomplished by either:

- Omitting the JSON attribute inside the JSON object
- Including the attribute name in the JSON Object with a value of `JSONObject.NULL`

JSON payloads with null values created by Rules

When Rules set a null value that propagates into the payload of a request, JSON treats the null as follows:

Assume that the incoming payload is...

```
Person
  Age  : 45
  Name : Jack
```

... and that rule processing sets `Age` to a `JSONObject.NULL` object.

If `JSONObject.toString()` is called, the output would look like this:

```
Person
  Age  : null
  Name : Jack
```

SOAP/XML

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

Vocabulary Type	Passing a null in an XML message
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.

XML Payloads with null values created by Rules

When Rules set a null value that propagates into the payload of a request, XML treats the null as follows

Assume that the incoming payload is...

```
Person
  Age  : 45
  Name : Jack
```

... and that rule processing sets `Age = null`.

The output would remove `Age` from the payload like this:

```
Person
  Name : Jack
```

6

Secure servers with authentication and encryption

When planning how you will deploy and manage Corticon Decision Services, you need to consider how you will secure the deployment. When deploying Corticon you can use the basic authentication and encrypted communication of your host application server to secure your deployment.

- **Authentication** requires anyone accessing a server to authenticate by supplying their username and password credentials. Configuring authentication gives you control over which users can access a server and the actions they can perform. For example, a user might be able call a Decision Service but not to deploy one..
- **Encryption** enables confidential communication between a server and a client. Enabling HTTPS requires that you add your signed CA certificate to each server, and a CA client certificate on each of the clients that will access it.

For details, see the following topics:

- Implementing deployment security

Implementing deployment security

When deploying a Decision Service outside a firewall, in a cloud – anywhere unapproved users might try to access it -- you must consider how to secure the Decision Service and the Corticon Server's administrative API.Corticon Server lets you deploy secure Decision Services by:

- Securing calls to Decision Services

- Securing calls to management APIs
- Encrypting payloads
- Using user credentials for database access when using Enterprise Data Connections (EDC)

Setting up authentication for secure server access

To define the authentication mechanism and constraints, update the security configuration defined in the `web.xml` file inside the web archive on the Corticon Server. In a default installation that location is `[CORTICON_SERVER_WORK_DIR]\pas\server\webapps\axis\WEB-INF\web.xml`.

Within the `web.xml` is a commented-out block that defines common security constraints. Uncommenting this block enables basic authentication when you restart the server.

```
<security-constraint>
   <web-resource-collection>
     <web-resource-name>All Corticon SOAP Servlet Access</web-resource-name>
     <url-pattern>/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
     <role-name>ROLE_CorticonAdmin</role-name>
   </auth-constraint>
</security-constraint>

<login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>Corticon Server Realm</realm-name>
</login-config>

<security-role>
  <role-name>*</role-name>
</security-role>
```

Note: If you already uncommented this section to enable HTTPS, review the `web-resource-collection` defined, and then add the `auth-constraint` block, and uncomment the `login-config` and `security-role` sections.

With the above configuration, every time a user tries to access the server through a URL, a valid username/password must be supplied and verified. You need to decide whether to restrict defined user *roles* to specified URLs – the *endpoints* that perform specific actions. That is described in the next topic.

The default user definitions used by a PAS server are defined in the `tomcat-users.xml` file (in a default installation its location is `[CORTICON_SERVER_WORK_DIR]\pas\server\conf\tomcat-users.xml`) as follows:

```
<role rolename="ROLE_CorticonAdmin" />
<role rolename="ROLE_CorticonExecute" />

<user username="admin" password="admin" roles="ROLE_CorticonAdmin, ROLE_CorticonExecute"
 />
<user username="ccuser" password="ccuser" roles="ROLE_CorticonExecute" />
```

You can modify these passwords as well as add additional users to this file.

Securing Server endpoints

You can choose to restrict defined user *roles* to specified URLs – the *endpoints* that perform specific actions. You can put a constraint on a particular URL or many URLs associated with the Web Application. Endpoint security is defined in **`<security-constraint>`** tags.

```
<security-constraint>
    <web-resource-collection>
       <url-pattern>
```

You can define the URL(s) where the constraint will apply. For the constraint to be applied to all URLs associated with your Web Application, set the value to `/*`. However, a finer level of granularity can be applied. Here are the five core URL pattern matches associated with Corticon SOAP and REST API URLs:

1. Corticon Admin SOAP Servlet:

   ```
   /services/CorticonAdmin
   ```

   You can perform Admin operations over SOAP such as deployment or configuration of Decision Services. See Admin SOAP example.

2. Corticon Execute SOAP Servlet:

   ```
   /services/CorticonExecute
   ```

   You can execute Decision Services via SOAP requests. See Execute SOAP example.

3. Corticon Decision Service Admin REST:

   ```
   /corticon/decisionService/*
   ```

   Similar to the Admin SOAP endpoint, this endpoint is over REST. Because REST has a distinct URL for each operation, this endpoint use an * for URL matching of all Decision Service Admin operations. See Admin DS example.

4. Corticon Server Admin REST:

   ```
   /corticon/server/*
   ```

   You can perform Admin operations against the Corticon Server over REST such as getting Corticon Server metrics and properties. See Admin REST example.

5. Corticon Execute REST:

   ```
   /corticon/execute
   ```

   You can execute Decision Services via REST requests. See Execute REST example.

```
<security-constraint>
    <web-resource-collection>
    <auth-constraint>
```

When the user passes in their credentials (username/password), the Authentication process verifies that the credentials match a role that has been defined on the server. This section in the `web.xml` lets you limit access to the URL based on particular roles. If you don't want to limit based on particular roles, then this value should be set to *.

However, you may not want everyone to have access to the Admin URLs. In this particular case, you would want to add a specific `<security-constraint>` for the Admin URLs with an assigned `<role-name>` to limit which types of users can access the URLs.

```
<security-constraint>
    <web-resource-collection>
```

```
        <web-resource-name>Corticon Admin SOAP Servlet Access</web-resource-name>
        <url-pattern>/services/CorticonAdmin</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>ROLE_CorticonAdmin</role-name>
    </auth-constraint>
</security-constraint>
```

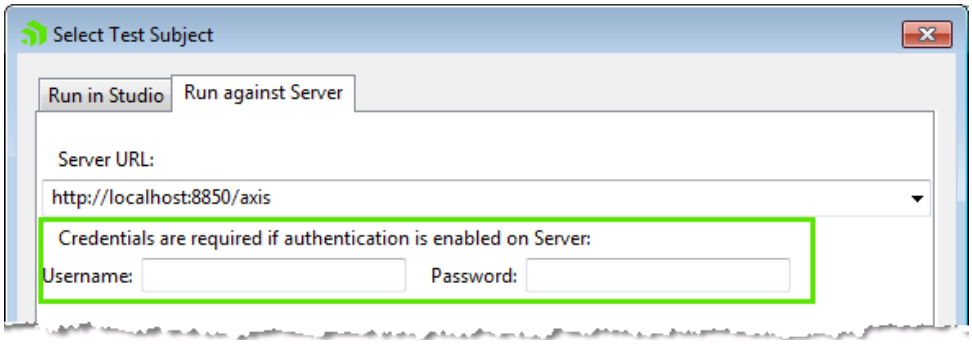
Each role-name defined in any security-constraint in the `web.xml` must be defined in a `<security-role>` tag.

```
Example: (All role names)

<security-role>
    <role-name>*</role-name>
</security-role>

Example: (Two specific role names)

<security-role>
    <role-name>ROLE_CorticonAdmin</role-name>
</security-role>
<security-role>
    <role-name>ROLE_CorticonExecute</role-name>
</security-role>
```

Note: If you want all role names to be valid for the Web Application, then just put `*` in the XML Element. Even if you have set the constraint's role-name to `*`, you still need to define a `<security-role>` role-name with `*`.

Using authentication in Studio

When a Corticon Studio Ruletest wants to use a secured deployed server to test its rules, credentials are required to connect to the server. To support this functionality the **Select Test Subject** dialog's **Run against Server** tab lets you add credentials, as shown:



The credentials are saved within each Testsheet for automatic re-use later.

Note: The credentials must permit both Admin and Execution operations.

Using authentication in client requests

When a request is submitted by a client to a Decision Service on a secured Corticon Server, the client must have credentials to enable connection to the server. The credentials must permit Execution operations.

You embed the username/password in the HTTP `GET` or an HTTP `POST` in the `Authorization` parameter in the header of the request, as follows:

1. Get the `HttpPost` object that is connecting to the URL on the Server:

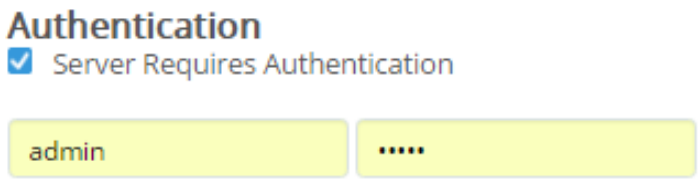
```
HttpPost postRequest = new HttpPost("http://localhost:8850/corticon/execute");
```

2. Create an `encodedString` and add that to the specific Header:

```
//  Encode the username/password in Base64.  This is needed for REST calls
String username = "admin";
String password = "admin";
encodedString = Base64.encode((username + ":" + password).getBytes());

String base64EncodedCredentials = "Basic " + encodedString;

//  Add the String to the appropriate Header
postRequest.setHeader("Authorization", base64EncodedCredentials);
```

Using authentication within Web Console

When you manage servers in the Web Console, you need to maintain credentials that enable access to each managed server, as shown:



The credentials are saved within each Server definition in the Web Console repository.

Using LDAP for Web Console authentication

You can set up Corticon Web Console to authenticate users using an LDAP server. To configure LDAP authentication, edit the file `CorticonServerConsoleConfig.groovy` located in `[CORTICON_WORK_DIR]\etc`.

Uncomment all property lines in this file and enter values for the first four properties. Here is an example:

```
/*Configure the manager distinguished name based on your ldap configuration*/
grails.plugin.springsecurity.ldap.context.managerDn = 'uid=admin,ou=system'

/*Ldap password*/
grails.plugin.springsecurity.ldap.context.managerPassword = 'secret'

/*Ldap server url */
grails.plugin.springsecurity.ldap.context.server = 'ldap://localhost:10389'

/*Ldap search base*/
grails.plugin.springsecurity.ldap.search.base = 'o=mojo,ou=groups'
```

To map Web Console Admin and User roles to LDAP user groups, specify the user group names in the `ldap{}`section at the bottom as shown in the image below. Use commas to define multiple user groups for each role.

```
corticon
{
    ldap {
        admin = "Admin,DBAdmin"
        users = "User,Analyst,Developer"
        authenticateUsersWithGroups = "false" //This flag is used to
authenticate the Admin and User groups listed above. If the flag is
false, and the group is not listed above, the user will be defaulted
to user role.

   }
}
```

After setting these properties, save the file and restart Corticon Server. After Corticon Server starts, LDAP users can log in to Web Console using their LDAP user credentials. Once an LDAP user logs in, they are added to the **USERS** page in Web Console.

Note: Setting up LDAP authentication adds LDAP users to the Web Console user base. You can still add other users in the **USERS** page and have them access Web Console using their Web Console user credentials.

Using authentication in server test scripts

You can enter a username and password in test applications that can be used for Authentication against a secured server. In a test application, you can change the connection properties by selecting option 0 (zero). That option has parameters that add a username and password to use with all calls to the remote server in the session. For example:

```
-----------------------------------------------------------------
Enter transaction number:
0

=== Results ===

Input new URL to J2EE Web Server: type cancel <enter> to stop operation
Current Server Location: http://localhost:8850
example locations:
    Default Weblogic:      http://localhost:7001
    Default Websphere:     http://localhost:9080
    Default Apache Tomcat: http://localhost:8850
-------------------------------------------------------------------------------
http://locahost:8850

Input new Web Application name: type cancel <enter> to stop operation
(example: axis)
axis

(optional) Username if using a secured server: type cancel <enter> to stop operation
admin

(optional) Password if using a secured server: type cancel <enter> to stop operation
admin

Transaction completed.

Enter transaction number:
```

To clear the username and password, press Enter at the username or password prompts.

Note: Depending on how you secured the Corticon Server, you may need to use option 0 (zero) often. For example, if the URLs `services/CorticonAdmin` and `services/CorticonExecute` have different constraints with different Role Names in the `web.xml`, then you will need to switch back and forth between Admin approved username/password and Execute username/password depending on what you want to do. To make things easier, it is a good idea to have just one constraint for both URLs.

Using API calls to a secure web application

The following are examples of each of the five core URL patterns outlined in Securing Server endpoints on page 75.

Corticon Admin SOAP example

In this example, the SOAP call is made to the `/axis/services/CorticonAdmin` URL requesting a list of Decision Services that are deployed on a remote server.

```
IMPORT STATEMENTS:
import java.net.URL;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;

SAMPLE CODE:
Service service = new Service();
Call call = (Call) service.createCall();

String username = "admin";
String password = "admin";

//  Add in username/password
call.setProperty(Call.USERNAME_PROPERTY, username);
call.setProperty(Call.PASSWORD_PROPERTY, password);

URL serverURL = new URL("http://localhost:8850/axis/services/CorticonAdmin");
String methodName = "getDecisionServiceNames";
Object[] variables = new Object[]{};

call.setTargetEndpointAddress(serverURL);
call.setOperationName(methodName);

//  Make the actual call to the Web Service
String targetNamespaceXSD = "http://soap.corticon.com";
String returnValue = (String) call.invoke(targetNamespaceXSD, methodName, variables);
```

CorticonExecute SOAP example

In this example, the SOAP call is made to the `/axis/services/CorticonExecute` URL while passing a `CorticonRequest` String through an RPC call to the remote server.

```
IMPORT STATEMENTS:
import java.net.URL;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;

SAMPLE CODE:
Service service = new Service();
Call call = (Call) service.createCall();

String username = "ccuser";
String password = "ccuser";

//  Add in username/password
call.setProperty(Call.USERNAME_PROPERTY, username);
call.setProperty(Call.PASSWORD_PROPERTY, password);

URL serverURL = new URL("http://localhost:8850/axis/services/CorticonExecute");
String methodName = "executeRPC";

String corticonRequest = "";
corticonRequest += "<CorticonRequest decisionServiceName=\"ProcessOrder\">";
corticonRequest += "   <WorkDocuments>";
corticonRequest += "      <Order id=\"Order_id_1\">";
corticonRequest += "         <dueDate>1/1/2008 12:00:00 AM</dueDate>";
corticonRequest += "         <myItems id=\"Item_id_1\">";
corticonRequest += "            <price>10.000000</price>";
corticonRequest += "            <product>Ball</product>";
corticonRequest += "            <quantity>20</quantity>";
corticonRequest += "         </myItems>";
corticonRequest += "      </Order>";
corticonRequest += "   </WorkDocuments>";
corticonRequest += "</CorticonRequest>";
```

```
Object[] variables = new Object[]{corticonRequest};

call.setTargetEndpointAddress(serverURL);
call.setOperationName(methodName);

//  Make the actual call to the Web Service
String targetNamespaceXSD = "http://soap.corticon.com";
String returnValue = (String) call.invoke(targetNamespaceXSD, methodName, variables);
```

Corticon Decision Service Admin REST example

In this example, the REST API call is made to the `/axis/corticon/decisionService/list` URL requesting a list of Decision Services that are deployed on the remote server.

```
IMPORT STATEMENTS:
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;

import com.sun.org.apache.xml.internal.security.utils.Base64;

SAMPLE CODE:
String serverURL = "http://localhost:8850/axis/corticon/decisionService/list";

HttpGet getRequest = new HttpGet(serverURL);

String username = "admin";
String password = "admin";

//  Encode the username/password in Base64.  This is needed for REST calls
String encodedString = Base64.encode((username + ":" + password).getBytes());

String base64EncodedCredentials = "Basic " + encodedString;

getRequest.setHeader("Authorization", base64EncodedCredentials);

//  Send the request; It will immediately return the response in HttpResponse object //
  if any
CloseableHttpClient httpClient = HttpClients.createDefault();
HttpResponse response = httpClient.execute(getRequest);

//  First verify for valid status code
int statusCode = response.getStatusLine().getStatusCode();
if (statusCode != 200)
{
   String error = "Failed with HTTP error code " + statusCode + ": " +
  response.getFirstHeader("error");

   throw new Exception(error);
}

HttpEntity httpEntity = response.getEntity();
String responseObjectString = EntityUtils.toString(httpEntity, "UTF-8");
```

CorticonAdmin REST example

In this example, the REST API call is made to the `corticon/server/info` URL requesting a `JSONObject` containing information about what Decision Services are currently deployed on the remote server.

```
IMPORT STATEMENTS:
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpGet;
```

```
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;

import com.sun.org.apache.xml.internal.security.utils.Base64;

SAMPLE CODE:
String serverURL = "http://localhost:8850/axis/corticon/server/info";

HttpGet getRequest = new HttpGet(serverURL);

String username = "admin";
String password = "admin";

//  Encode the username/password in Base64.  This is needed for REST calls
String encodedString = Base64.encode((username + ":" + password).getBytes());

String base64EncodedCredentials = "Basic " + encodedString;

getRequest.setHeader("Authorization", base64EncodedCredentials);

//  Send the request; It will immediately return the response in HttpResponse object //
  if any
CloseableHttpClient httpClient = HttpClients.createDefault();
HttpResponse response = httpClient.execute(getRequest);

//  First verify for valid status code
int statusCode = response.getStatusLine().getStatusCode();
if (statusCode != 200)
{
String error = "Failed with HTTP error code " + statusCode + ": " +
response.getFirstHeader("error");

throw new Exception(error);
}

HttpEntity httpEntity = response.getEntity();
String responseObjectString = EntityUtils.toString(httpEntity, "UTF-8");
```

CorticonExecute REST example

In this example, the REST call will be made to the `/corticon/execute` URL while passing a JSONObject as the payload to the remote server.

```
IMPORT STATEMENTS:
import java.io.StringWriter;

import javax.ws.rs.core.MediaType;

import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;
import org.json.JSONArray;
import org.json.JSONObject;

import com.sun.org.apache.xml.internal.security.utils.Base64;

SAMPLE CODE:
String serverURL = "http://localhost:8850/axis/corticon/execute";

HttpPost postRequest = new HttpPost(serverURL);

//Set the API media type in http content-type and the Decision Service Name to the header
postRequest.addHeader("content-type", MediaType.APPLICATION_JSON);
```

```
postRequest.addHeader(ICcServer.REST_HEADER_DECISION_SERVICE_NAME, "ProcessOrder");

String username = "ccuser";
String password = "ccuser";

//  Encode the username/password in Base64.  This is needed for REST calls
String encodedString = Base64.encode((username + ":" + password).getBytes());

String base64EncodedCredentials = "Basic " + encodedString;

postRequest.setHeader("Authorization", base64EncodedCredentials);

//  Create JSON Payload
StringWriter writer = new StringWriter();

   /*
   String jsonPayload = {"Objects": [{
      "myItems": {
          "product": "Ball",
          "price": "10.000000",
          "quantity": "20",
          "__metadata": {
              "#id": "Item_id_1",
              "#type": "Item"
          }
      },
      "__metadata": {
          "#id": "Order_id_1",
          "#type": "Order"
      },
      "dueDate": "1/1/2008 12:00:00 AM",
  }]}
  */

JSONObject jsonPayload = new JSONObject();

//  Create Order object
JSONObject order = new JSONObject();
order.put("dueDate", "1/1/2008 12:00:00 AM");

JSONObject orderMetadata = new JSONObject();
orderMetadata.put("#id", "Order_id_1");
orderMetadata.put("#type", "Order");

order.put("__metadata", orderMetadata);

//  Create Item object
JSONObject item = new JSONObject();
item.put("product", "Ball");
item.put("price", 10.0);
item.put("quantity", 20);

JSONObject itemMetadata = new JSONObject();
itemMetadata.put("#id", "Item_id_1");
itemMetadata.put("#type", "Item");

item.put("__metadata", itemMetadata);

//  Add Item object to rolename myItems
order.put("myItems", item);

//  Add Order to main payload
JSONArray rootObjects = new JSONArray();
rootObjects.put(order);
jsonPayload.put("Objects", rootObjects);

//  Attach JSONObject to the postRequest
jsonPayload.write(writer);
StringEntity userEntity = new StringEntity(writer.getBuffer().toString(), "UTF-8");
```

```
postRequest.setEntity(userEntity);

//  Send the request; It will immediately return the response in HttpResponse object //
  if any
CloseableHttpClient httpClient = HttpClients.createDefault();
HttpResponse response = httpClient.execute(postRequest);

//  First verify for valid status code
int statusCode = response.getStatusLine().getStatusCode();
if (statusCode != 200)
{
   String error = "Failed with HTTP error code " + statusCode + ": " +
   response.getFirstHeader("error");

   throw new Exception(error);
}

HttpEntity httpEntity = response.getEntity();
String responseObjectString = EntityUtils.toString(httpEntity, "UTF-8");
```

Setting up encryption between servers and clients

Enabling HTTPS

Corticon Server supports encrypted communications between the Web server and a Web service client. If you attempt to use the default HTTPS port, `8851` (for example, connecting from the Web Console, you get a security message indicating that your connection is not private. If you want to use HTTPS, you must enable the HTTPSconnections.

Note: The following procedure pertains to the security of communication between the client application and the Server. To enable HTTPS communication between the Server and the client, you must obtain and install public key certificates for the Server host machine and complete separate configuration procedures for each deployed Client service and for the Server.

To enable HTTPS on Corticon Server for Java:

1. Obtain a private key and a Web server digital certificate.
2. Install the Web server digital certificate in the Web server.
3. Start the Corticon Server. When startup is complete, stop it. The initial startup creates the `web.xml` file.
4. Edit the file `web.xml` located at `[CORTICON_WORK_DIR]\pas\server\webapps\axis\WEB-INF\` to uncomment the following section:

   ```
   <security-constraint>
     <web-resource-collection>
       <web-resource-name>Corticon Server</web-resource-name>
       <url-pattern>/*</url-pattern>
     </web-resource-collection>
   ...
   </security-constraint>
   ```

   Add in the following block to replace … above:

   ```
   <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
   ```

Note: If you already uncommented this section to enable Basic Authentication, review the `web-resource-collection` defined, and then add the `user-data-constraint` `transport-guarantee` block.

5. Save the file.

When you restart the Corticon Server, HTTPS is enabled on its default port, `8851`.

Enabling the Corticon Studio to publish to a secure Corticon Server

Corticon Studio supports encrypted communications to a Corticon Server. To enable HTTPS communication between the Server and the Client, you must obtain and install public key certificates for the Corticon Studio. The public certificate then needs to be imported to the Java keystore for the Corticon Studio.

7

Invoking Corticon Server

The previous section discussed the *contents* of a call to Corticon Server, and what those contents need to look like. This section 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
- SOAP call
- Java call
- REST call
- Request/Response mode
- Administrative APIs

Methods of calling Corticon Server

There are three ways to call Corticon Server:

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

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 Progress Application 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 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 Tutorial: Basic Rule Modeling*, 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\bin`) 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 the testServer API console. 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.

REST call

On the Corticon Server for .NET, you can use REST to execute a JSON-formatted Corticon Request by specifying the target Decision Service. The REST method supported is `HTTP POST` in the form:

```
HTTP POST:base/axis/corticon/execute
```

where *base* is the Corticon Server for .NET's IP or DNS-resolvable name and port. The Decision Service payload is enclosed in the request message's body as a JSON string. The following is the header of an HTTP request defined for executing Decision Service `ProcessOrder` on a local base:

```
http://localhost:80/axis/corticon/execute
Content-Type: application/json
dsName: ProcessOrder
Host: localhost:80
Content-Length: 1035
```

The `POST` request does not require the header to carry information about the Decision Service. Those parameters can be set as fields inside the JSON object to specify the Decision Service, shown in the following example for `ProcessOrder v1.16`:

```
http://localhost:80/axis/corticon/decisionService/setProperties


{
  "name" : "ProcessOrder",
  "majorVersion" : "1",
  "minorVersion" : "16",
  "msgStyle":"FLAT",
  "autoReload", "true"
}
```

Evaluation of a request will look first for the message body, and if not found, it will look for it as an HTTP header.

Success and error responses

The response returned by the server has two parts:

- A HTTP status code
- An HTTP header containing message execution info

Success response - In addition to the processed request, when execution is successful, the response message's body contains the updated JSON string. A successful execution's status and header are, for example:

```
200 The Decision Service executed
```

```
HTTP/1.1 200 OK
Content-Length: 1397
Content-Type: application/json
Server: Microsoft-IIS/7.5
X-AspNet-Version: 4.0.30319
Set-Cookie: ASP.NET_SessionId=wfjtuzcvwvz55c1qyw4q4kym; path=/; HttpOnly
X-Powered-By: ASP.NET
```

Common error behavior

For all REST executions, there is a common error behavior. The error response has an HTTP status code other than 200, and a user-friendly error message in the header. For example:

```
400 Bad Request

HTTP/1.1 400 Null value for Header attribute 'dsName'.  Value cannot be null.
Server: Microsoft-IIS/7.5
Connection: close
X-AspNet-Version: 4.0.30319
error: Null value for Header attribute 'dsName'.  Value cannot be null.
Set-Cookie: ASP.NET_SessionId=dyt1gckr5ph13t1h5q5llbqx; path=/; HttpOnly
Content-Length: 0
```

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.
- Starts Corticon Server's Dynamic Update monitoring service (`startDynamicUpdateMonitoringService`). This is the same update service that can be set statically using Server properties in your `brms.properties` file.
- Stops Corticon Server's Dynamic Update monitoring service (`stopDynamicUpdateMonitoringService`). This is the same update service that can be set statically using Server properties in your `brms.properties` file.

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*, typically on port `8850` for HTTP, and on port `8851` for HTTPS, which among other things, exposes a set of WSDLs. See Inside Corticon Server on page 145 for more information.

8

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.

Note: Identity and strategy concepts are general relational database concepts. Refer to your RDBMS brand's documentation for more information, especially the identity strategies that are specific to certain brands.

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
- 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

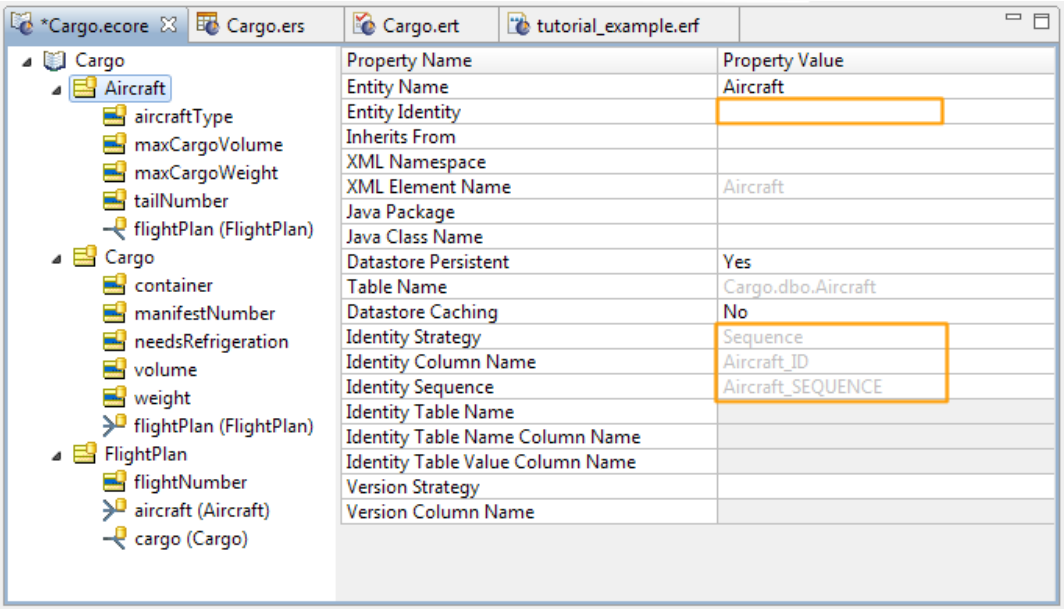
A Vocabulary entity uses datastore identity when it does not have an attribute that matches the database table's primary key field(s) . The table's primary key is effectively a *surrogate key* which really has no business meaning. If the designated primary key fields in the imported database metadata 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.

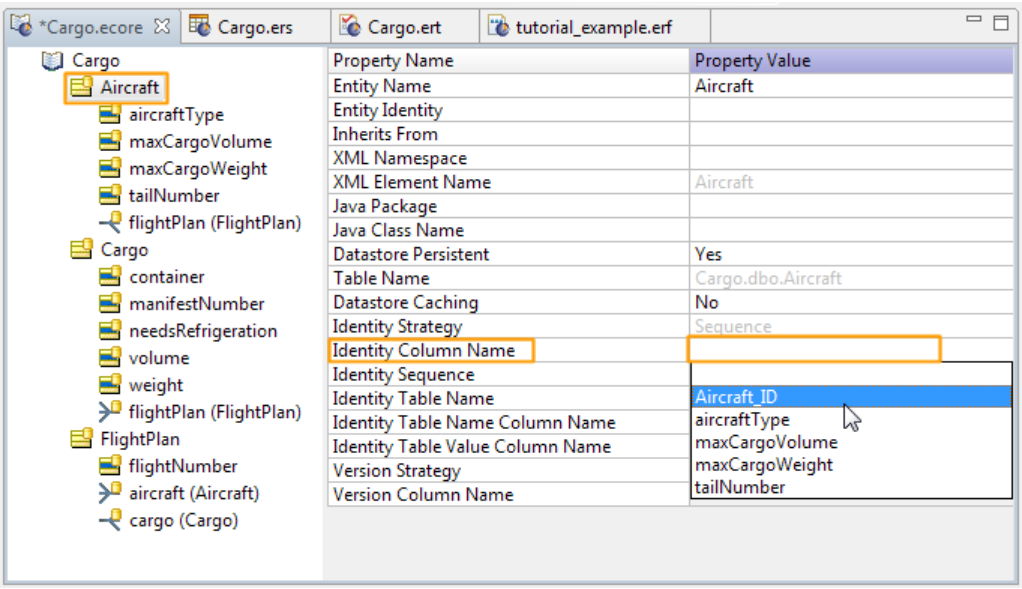
When the auto-mapper updated the schema, the Entity Identity was set to a `NULL`, and set the primary key field(s) in **Identity Column Name** as `Aircraft_ID`, as shown:

Figure 33: 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 34: 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.

For examples of proper syntax for datastore identities in query payloads, see the topic Metadata for Datastore Identity in XML and JSON Payloads on page 123

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.

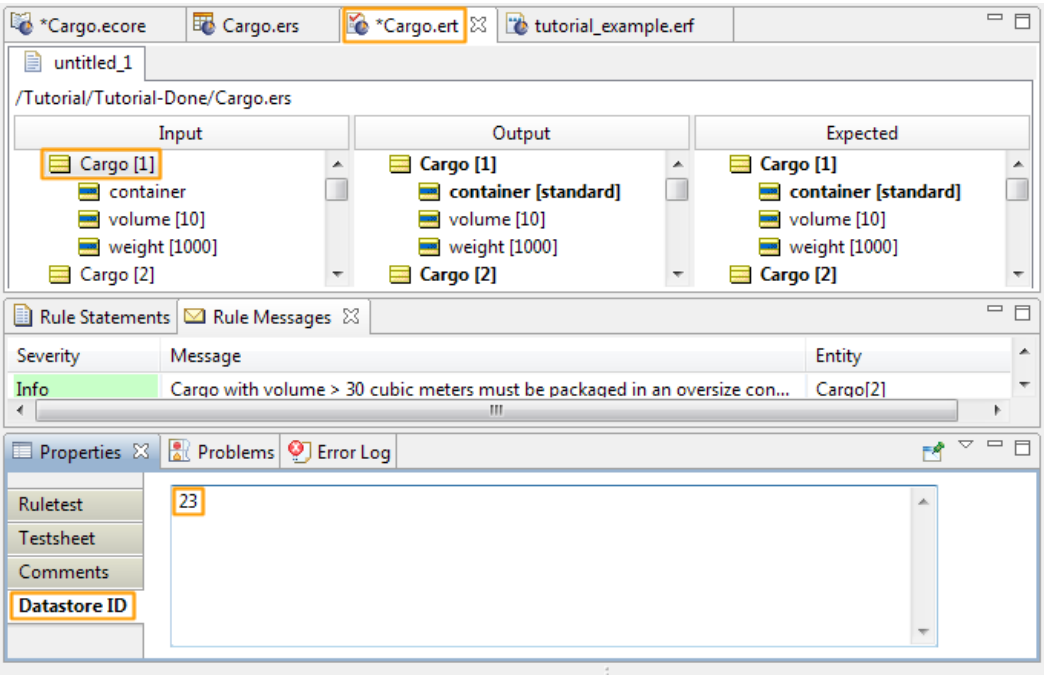
Primary Key

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 35: 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 36: Datastore Identity inside the XML Request

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <CorticonRequest xmlns="urn:Corticon" xmlns:
3     <WorkDocuments>
4       <Cargo databaseid="23" id="Cargo_id_1">
5         <weight>1000</weight>
6         <volume>10</volume>
7         <container xsi:nil="true" />
8       </Cargo>
9       <Cargo id="Cargo_id_2">
10        <weight>1000</weight>
11        <volume>40</volume>
12        <container xsi:nil="true" />
13      </Cargo>
14      <Cargo id="Cargo_id_3">
19      <Cargo id="Cargo_id_4">
25    </WorkDocuments>
26  </CorticonRequest>
```

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.

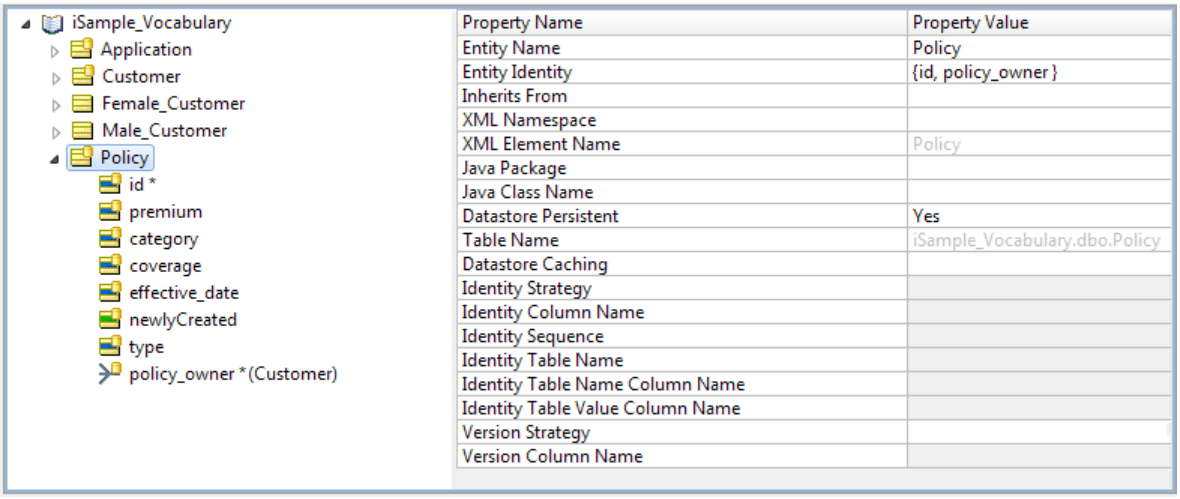
Some tables are not independent. Take `Customer` and `Policy` tables – if each policy record must have a person to whom the policy is “attached,” we say the `Policy` table is dependent upon the `Customer` table. A person may or may not have a policy, but each policy must have a person.

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 37: Primary Key of a Dependent Table Includes the Role Name



Inferred property values

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

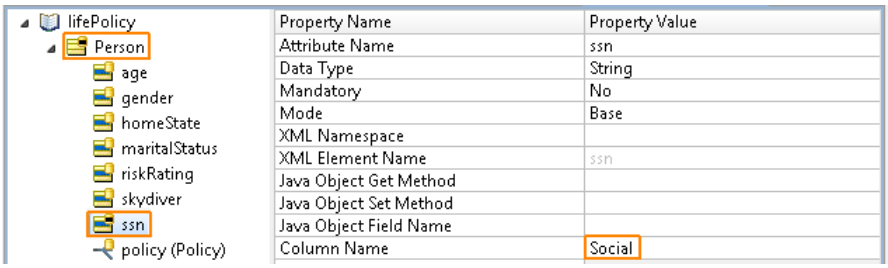
The ability to match entity names with table names is a key capability, because many of the other matching strategies will indirectly rely on this capability. Generally speaking, the system will locate the first table in database metadata that matches the entity name (ignoring case). For the purpose 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 8: Corticon inference rules

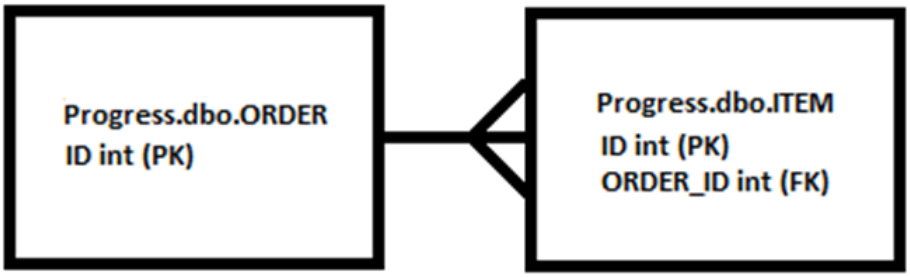
Vocabulary Property	Automatic Inference Rules
`Table Name`	Derived from table metadata. The first table located in database metadata that matches the entity name (ignoring case) will be chosen. This matching process will ignore catalog, schema and domains. The inferred value will be displayed as a fully-qualified 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:

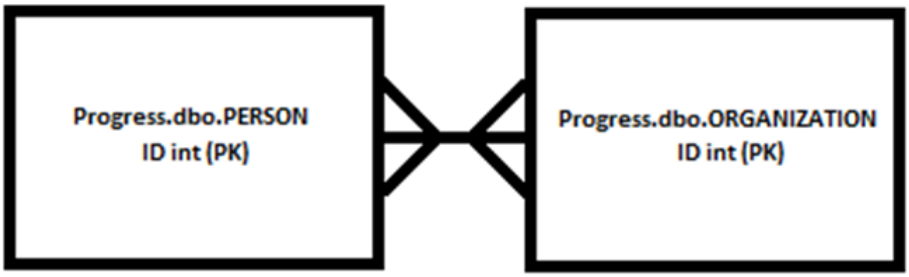
Vocabulary Association	Join Expression
`Order.item`	`{ Progress.dbo.ORDER.ID1 = Progress.dbo.ITEM.ORDER_ID1, Progress.dbo.ORDER.ID2 = Progress.dbo.ITEM.ORDER_ID2 }`
`Item.order`	`{ Progress.dbo.ITEM.ORDER_ID1 = Progress.dbo.ORDER.ID1, Progress.dbo.ITEM.ORDER_ID2 = Progress.dbo.ORDER.ID2 }`

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/`.

9

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 EDC tutorials, Modeling Progress Corticon Rules to Access a Database using EDC and Connecting a Progress Corticon Decision Service to a Database using EDC, provide a walk through setting up Microsoft SQL Server 2014 to create tables that Corticon Studio and Corticon Server then use to create, read, update, and delete data. We recommend completing or reviewing both tutorials before reading this chapter. This chapter extends the information covered by that guide.
- *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 full set of options and parameters when using EDC, and then takes a close look at mappings and validations in a Corticon connection to an RDBMS.
- Topics in Packaging and deploying Decision Services on page 31 in this guide describe the parameters for deploying 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 the available fields for Entities, Attributes, and Associations.

For details, see the following topics:

- Overview of the Enterprise Data Connector
- Working with EDC in Corticon Studio
- Working with EDC in Corticon Server

Overview of the Enterprise Data Connector

The following concepts and features are the basic tenets of Corticon EDC:

- **Technologies** - Corticon enables mapping to supported RDBMS brands using Progress Software's enterprise-grade Data Direct drivers, object/relational mapping in Hibernate, and the algorithms in C3P0's open-source JDBC connection pooling. Corticon creates Corticon Data Objects (CDOs) that are Hibernate-ready, and requests Hibernate to validate them against the database. These tools are embedded in the Corticon products, and maintained as product upgrades are applied. There is no initial need for configuring any of these supporting technologies, yet common tuning parameters are documented.
- **Corticon's Vocabulary binds to specified RDBMS metadata** - The Corticon Vocabulary stores the database connection and database metadata (tables, columns, primary keys, and foreign keys) that Corticon loads into its working memory as Corticon Data Objects (CDOs) for use in rule execution. Database metadata is used to infer the values of database-related fields based on rules. The import of database metadata imported into the Corticon Vocabulary can be constrained to specified entities to minimize the amount of metadata stored inside the Vocabulary asset. Dynamic validation tries to ensure that the Vocabulary always makes sense with respect to imported metadata. All Corticon assets that share a Vocabulary are relating to the same database.
- **Database Keys** - Entities can use either *application identity* (that is, primary keys) or *datastore identity*. With application identity, the application supplies the key values to create new instances, whereas with datastore identity the key values can be established through predefined identity strategies (or the default `AutoGenID` mechanism.)

- **Persistence** - Vocabulary entities can be *transient* or *datastore persistent* (that is, database-bound). Transient entities can have associations to persistent entities and vice versa. The user must supply instances of transient entities in the input message; conversely, datastore persistent entities are retrieved/updated in the database.
- **Database caching** - The Enterprise Data Connector supports *database caching*, the ability of a database to locally persist data that was retrieved from a database to expedite subsequent retrieval. This advanced concept's enablement in Studio is discussed in *"Using Database Caching" in the Rule Modeling Guide* and related menu and property settings in the *Quick Reference Guide*, and its management in production is discussed in *"Database Caching" in the Integration and Deployment Guide*.

Working with EDC in 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.

For a detailed example, see the EDC tutorials, Modeling Progress Corticon Rules to Access a Database using EDC and Connecting a Progress Corticon Decision Service to a Database using EDC.

How Corticon Vocabulary terms relate to a database

A Corticon Vocabulary is fundamentally relational in nature, and conceptually equivalent to the elements of a typical relational database.

Table 9: Equivalent Corticon — Relational Database Concepts

Corticon	Relational Database
Vocabulary	Schema
Vocabulary: Entity	Table
Vocabulary: Attribute	Table Column or Field
Vocabulary: Association	Relationship between Tables
Ruletest Output	Table Row(s) or Record(s)

Your Corticon EDC design can be defined from either perspective:

- Start from the business perspective by converting a Vocabulary into a database. Because Studio is a powerful modeling environment, users often build Vocabularies "on-the-fly" in order to support their rule modeling. Assuming the Vocabulary design is acceptable to IT as the basis for a database, then Studio can be used to export schema information directly to a database engine and generate the necessary table structure within a defined tablespace.
- Start from an IT perspective by abstracting a data model from an existing database and using its terms and structure to create a Vocabulary for rule modelers to use in their rule building and testing.

Validation of names of entities, attributes and associations against SQL keywords and database restrictions

Commercial databases, such as Microsoft SQL Server and Oracle, use specific words for defining, manipulating, and accessing databases. These reserved keywords are part of the grammar used to parse and understand statements. Do not use database reserved words for Corticon Entity, Attribute, and Association names when creating the schema in Corticon. Your database support pages list reserved words -- for example, SQL Server 2012 -- that you should review as you prepare your Vocabulary for enterprise data connection.

Corticon makes a best-effort to validate the names against the SQL keywords against the database restrictions for column and table naming (such as length of a table name), and then validates generated column names (such as Foreign Key (FK) columns) against SQL keywords and table/column name restrictions.

Note: It is good practice to ensure that database columns not have hyphens, spaces or other special characters (even though some databases and SQL parsers allow them). The generally accepted valid values are all alphanumeric characters and the underscore character. It is a plus to use all-lowercase names to avoid platform case inconsistencies. For more information on Corticon's accepted names, see the topic *"Vocabulary node naming restrictions" in the Quick Reference Guide.*

Connecting a Vocabulary to a database

Defining a table namespace in the database

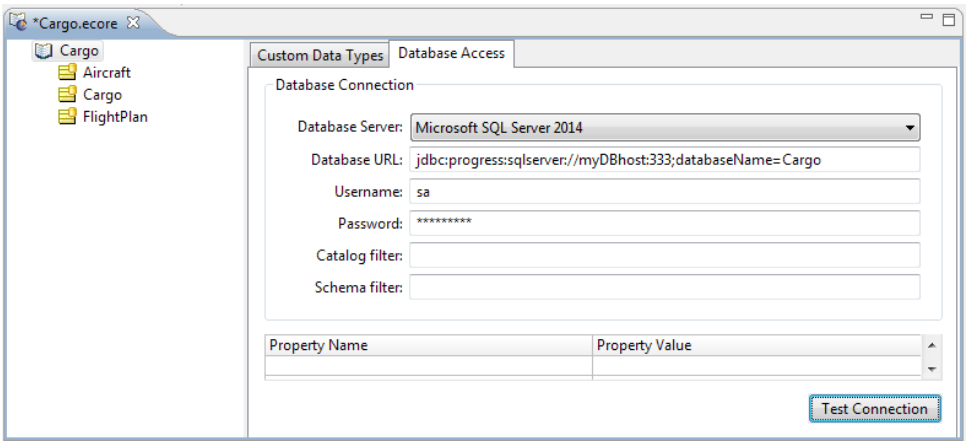
Set up your database product in a network-accessible location, and then define a database name. Note the database URL and port as well as the new database name. You also need credentials that will permit connection. These parameters are all that is typically required to connect the Vocabulary to the database, create the schema for the persistent entities, and then bring the database metadata back to the Vocabulary

---

Note: Refer to the Progress Software web page Progress Corticon 5.6 - Supported Platforms Matrix to review the currently supported database brands and versions.

Defining the database connection

To connect a Vocabulary to a newly defined database , the Vocabulary's **Database Access** tab provides essential and optional parameters, as follows:



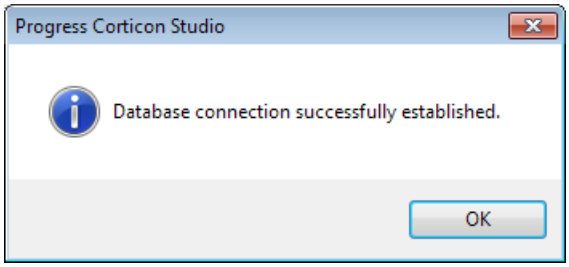
where:

- **Database Server**: The database product. Click the dropdown menu on the right side of the entry area to list the available database brands. Corticon embeds Progress DataDirect JDBC Drivers for each database. These drivers 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 Progress Applicaton Server. The drivers are pre-configured and do not require performance tuning.
- **Database URL**: The preconfigured URL for the selected database server. You must edit the default entry to replace (1) `<server>` with the machine's DNS-resolvable hostname or IP address and port (in this example, `myDBhost:333`), and (2) `<database name>` with the database name that was set up (typically case-sensitive).

  Note: Avoid using the loopback identity (`localhost` or `127.0.0.1`) as the server identity.

- **Username**: The user credentials that enable connection to the database. The credentials are encrypted when the database access file is exported for deployment.
- **Password**: The specified user's password.

Click **Test Connection**. The following alert indicates success:



The basic connection works. The optional parameters (catalogs and filters) and properties might be required in certain circumstances.

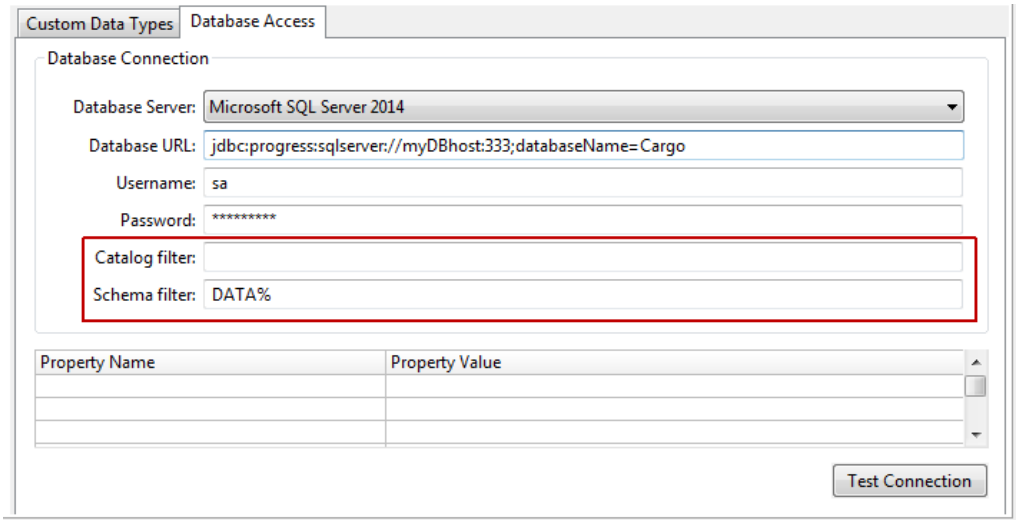
Filtering catalogs and schemas

When tables are generated to the database, they will use the default Catalog/Schema unless you specify otherwise.

Note: This feature is not available on all supported database brands. When you choose a Database Server, the feature will be enabled only if it is supported for that brand.

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 38: 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 -- you can make that choice in the **Import Database Metadata** selection panel.

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.

Setting additional database access properties

There are properties you might want to set for a database connection.

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

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.

The following properties let you tune connection pooling:

Table 10: 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.
`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`.

Note:

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

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

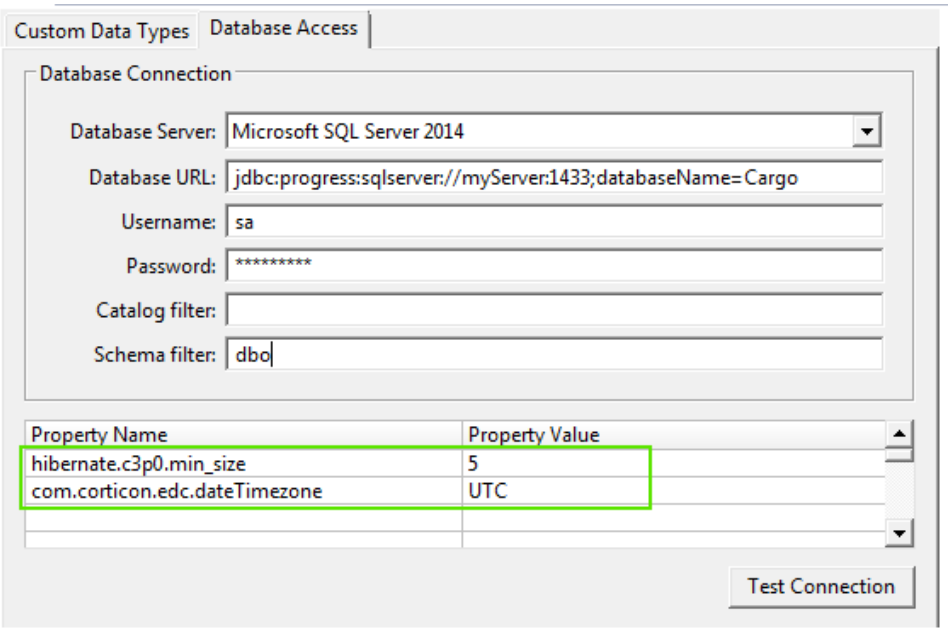
Database Time Zone

When your application stores date/time values in the database, you might need to set the following property:

`com.corticon.edc.dateTimezone`. This property pertains to only the DateTime data type, and lets you declare how DateTime values are expressed in the database:

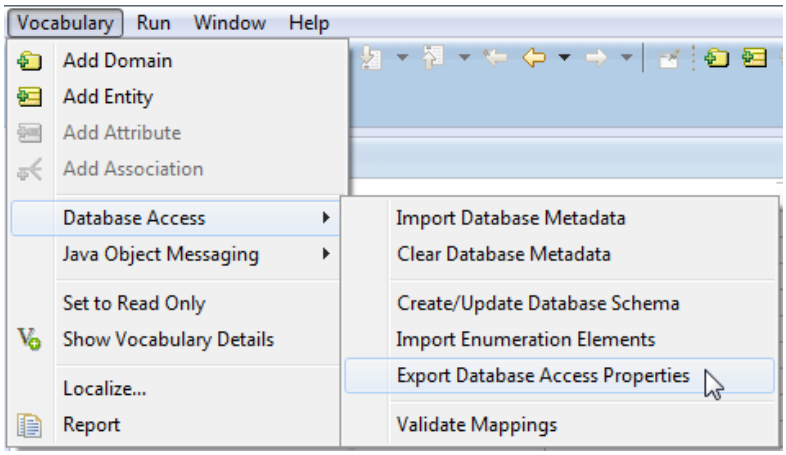
Value	Purpose
`JDK_DEFAULT_TIMEZONE`	Declares that date/time values will be expressed in the Java Virtual Machine (JVM) time zone. Use this setting if your date/time values are expressed in “local” time.
`UTC`	Declares that date/time values will be expressed in GMT. This setting is typical for internet applications that are used across time zones.
`America/Los_Angeles`	Declares that date/time values will be expressed in America/Los Angeles time.
`Europe/Paris`	Declares that date/time values will be expressed in Europe/Paris time.
`GMT+01:00`	Declares that date/time values will be expressed in time zone GMT plus one hour.

Set your override values in the **Property** table of the Vocabulary editor's **Database Access** tab, as illustrated:



Creating a database access properties file

Even though no tables have been defined for the database and nothing read or written, it is a good practice to export the successfully tested database access properties file. In the Vocabulary editor, select **Vocabulary > Database Access > Export Database Access Properties**, as shown:



You can specify a preferred name and location for the file, although colocating it within its related project folder is a good idea. The generated properties file looks like this:

```
com.corticon.database.id=com.corticon.database.id.MsSql2014
com.corticon.database.readonly.supported=true
hibernate.c3p0.max_size=100
hibernate.c3p0.max_statements=50
hibernate.c3p0.min_size=1
hibernate.c3p0.timeout=1800
hibernate.connection.driver_class=com.prgs.jdbc.sqlserver.SQLServerDriver
hibernate.connection.password=030046016058035029061039110
hibernate.connection.url=jdbc:progress:sqlserver://myDBhost:333;databaseName=Cargo
hibernate.connection.username=061046
hibernate.dialect=com.corticon.eclipse.studio.drivers.core.DDSQLServer2008Dialect
hibernate.temp.use_jdbc_metadata_defaults=false
```

Notice that the username and password values are very different from the credentials that were entered. These values were encrypted when the database access file was created, and will be decrypted when they are implemented in a decision service.

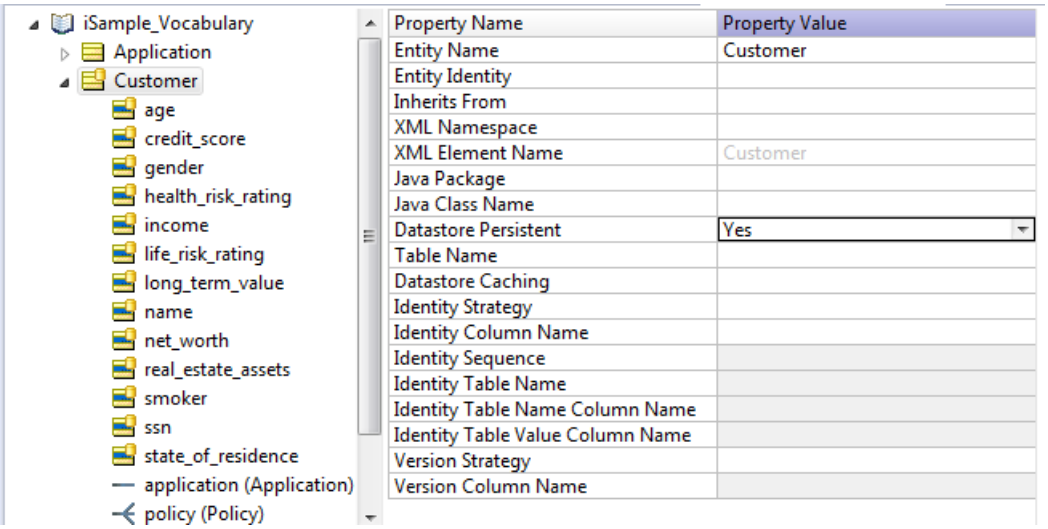
You shouldn't have to edit this file but if the Server is not colocated with the Studio, you see why it was a good idea to use the explicit host identifier rather than `localhost`.

Specifying entity properties for the database schema

In order to generate a database schema that will be mapped to a database, a few tasks are required. When none of these actions are taken, nothing happens when you attempt to create the database schema.

1. In the Vocabulary editor, select the menu toggle **Vocabulary > Show Vocabulary Details**.
2. Click on each entity that will be included in the database schema, to do the following:
   - **Set entity Database Persistence** - Click on the **Database Persistent** Property Value dropdown menu, and choose **Yes**, as shown:

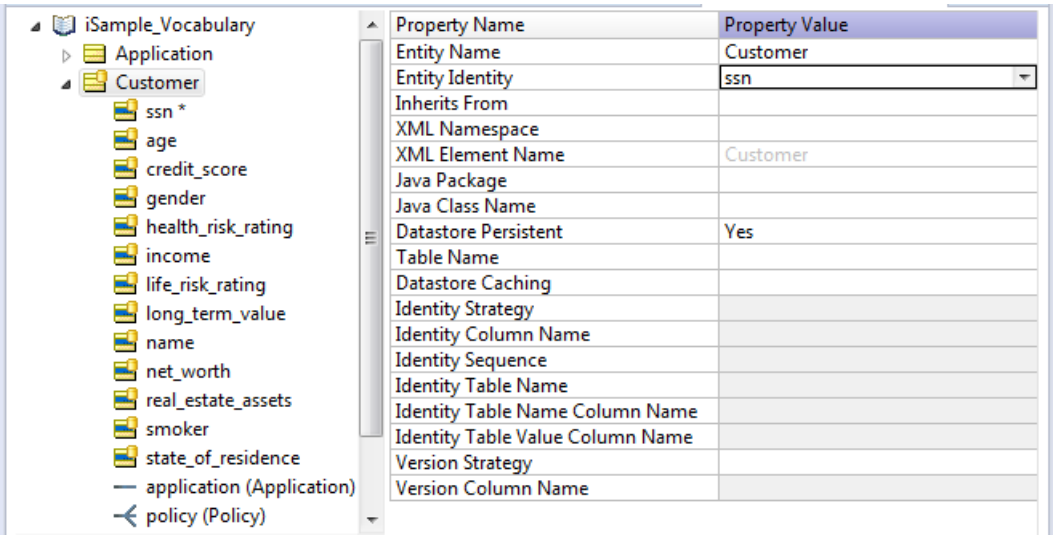
     Figure 39: Setting an entity to be database persistent



The entity and all its attributes now display their icon with a database decoration.

- **Set Entity Identity** - Considering Identity strategies on page 92, if you want to use an Application Identity, you will choose an attribute in the entity that you can enforce as unique, and then click on its **Entity Identity** Property Value dropdown menu, and choose from the listed attributes, as shown:

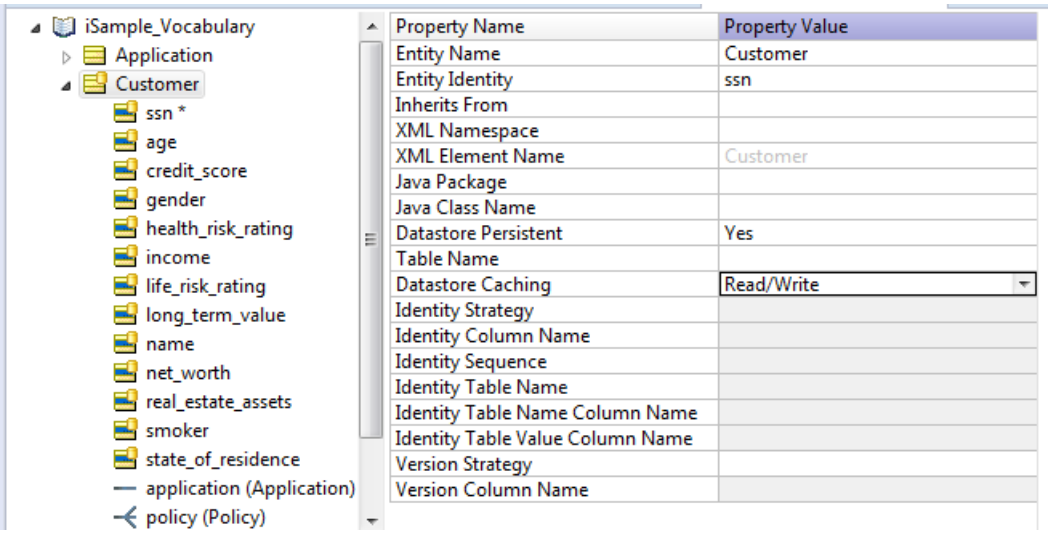
Figure 40: Setting an entity's identity



The selected attribute is now at the top of the listed attributes and marked with an asterisk (*). It will be the primary key (PK) in the generated table.

- **Set Datastore Caching** - (optional) The caching option is a level 2 cache that applies to specific settings in Rulesheets that - together with this entity property value, and enablement on a Ruletest or deployed Decision Service - provides caching of values that would be repeatedly accessed in the datastore. The setting here will apply to all use cases of caching on this entity.

Figure 41: Setting an entity's datastore caching preference



The Datastore Caching options are:

- `No Cache` or blank (default) - Disable caching.
- `Read Only` - Caches data that is never updated.
- `Read/Write` - Caches data that is sometimes updated while maintaining the semantics of "read committed" isolation level. If the database is set to "repeatable read," this concurrency strategy almost maintains the semantics. Repeatable read isolation is compromised in the case of concurrent writes.

- `Nonstrict Read/Write` - Caches data that is sometimes updated without ever locking the cache. If concurrent access to an item is possible, this concurrency strategy makes no guarantee that the item returned from the cache is the latest version available in the database.

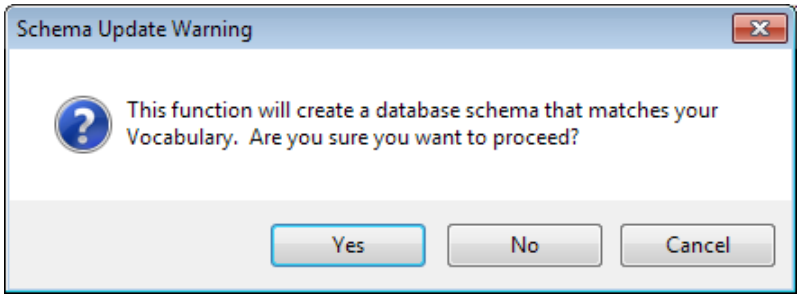
For more information on datastore caching, see the topic *"Defining database caching in Studio" in the Rule Modeling Guide* and the Working with database caches on page 142 in this guide.

Creating a schema in the database

Once a Vocabulary has database persistence, entity identity, datastore caching specified on each participating entity, the Vocabulary can create a corresponding schema in the database.

In the Vocabulary editor, choose the menu command **Vocabulary > Database Access > Create/Update Database Schema**. The following alert window opens:

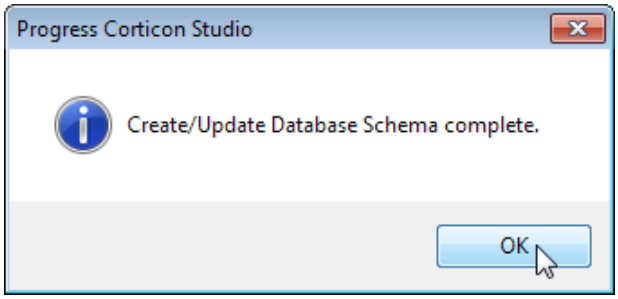
Figure 42: Schema Update Warning Message



The alert warns if a schema already exists in the database connected to this Vocabulary, as the **Create/Update Database Schema** function will change it.

Click **Yes** to proceed. When the create/update process succeeds, you see the following alert:

Figure 43: Database Schema Update Success Message



Click **OK** to dismiss the notification.

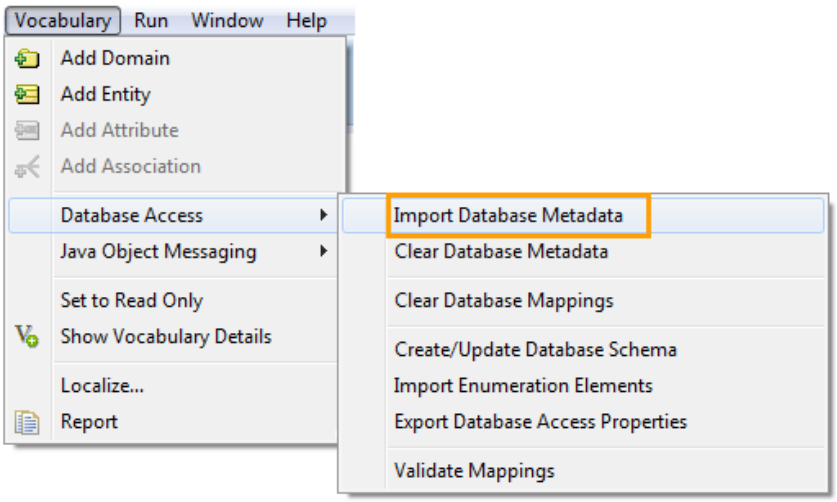
The database schema has been setup. Review the database to see the tables and their key assignments.

Importing database metadata into a Vocabulary

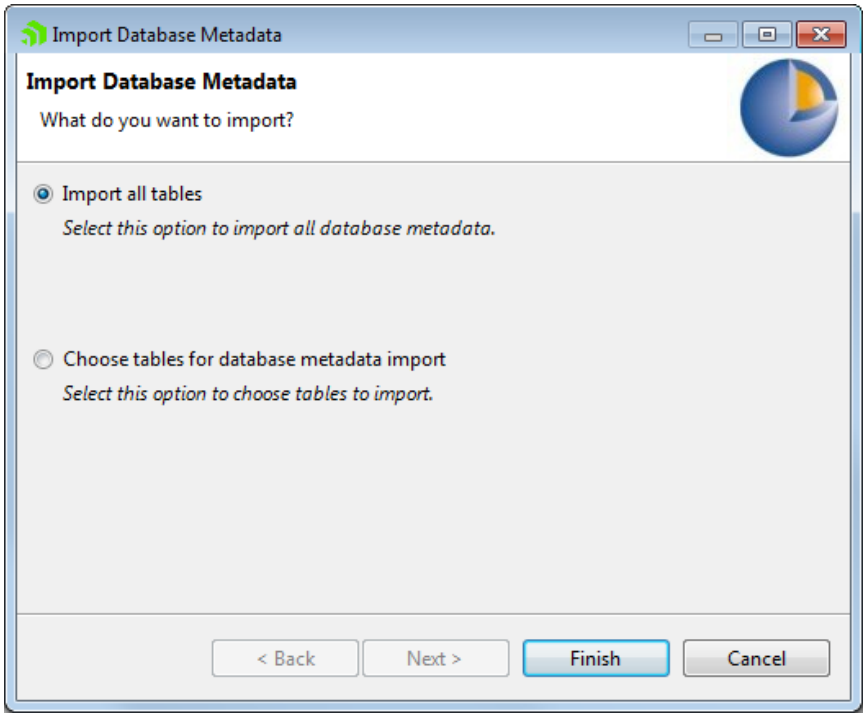
Once the database schema exists, the metadata it created can be imported back into Corticon Studio to refine and complete the mappings between the Vocabulary and the metadata.

You can control the tables that are accessed to transfer metadata to the Vocabulary. When only a small subset of tables will supply the metadata that is needed, the time and space overhead of the process is reduced by delimiting the tables.

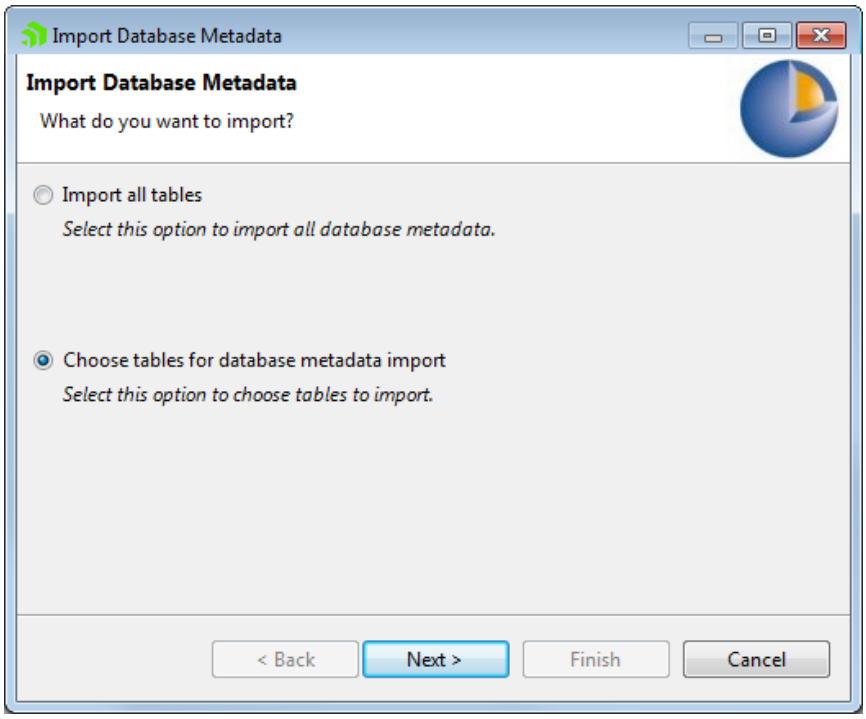
In the Vocabulary editor with a database connection established, select the menu command **Vocabulary > Database Access > Import Database Metadata**, as shown:



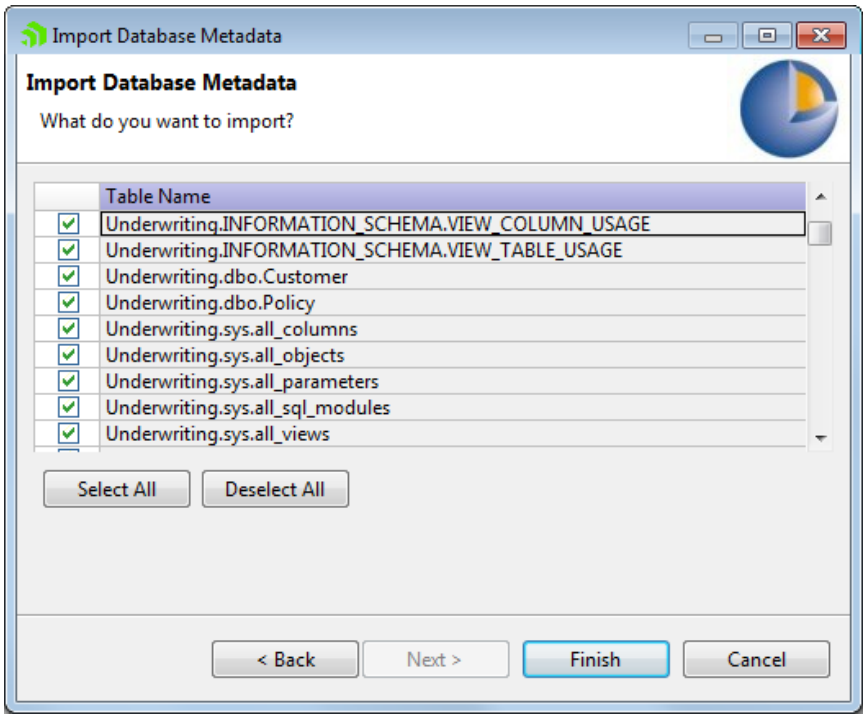
In the dialog, accept **Import all tables**, and click **Finish**, or...



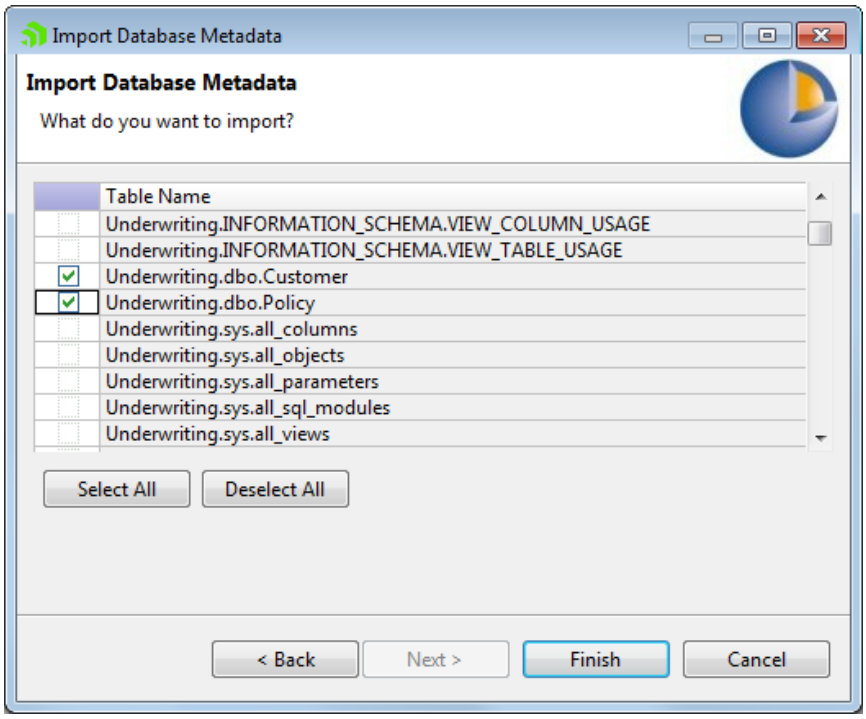
... or click **Choose tables for database metadata import**, and click **Next**.



The dialog lists all the tables in the connected database.



Use **Select All**, **Deselect All**, and individual selection boxes to refine the preferred tables.



In this example, just two tables are selected. Click **Finish** to perform the task.

As 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.

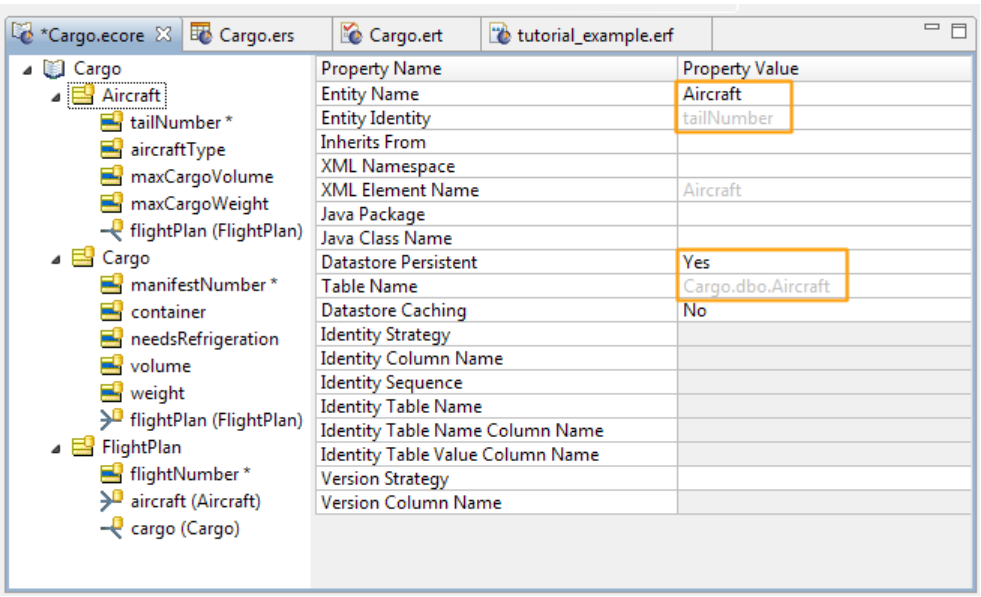
Mapping and validating database metadata

Mapping data between a Corticon Vocabulary and relational database is not always perfect. When there are issues, you need to review the mappings to resolve incomplete or conflicting mapping data.

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 44: Automatic Mapping of Vocabulary Entity

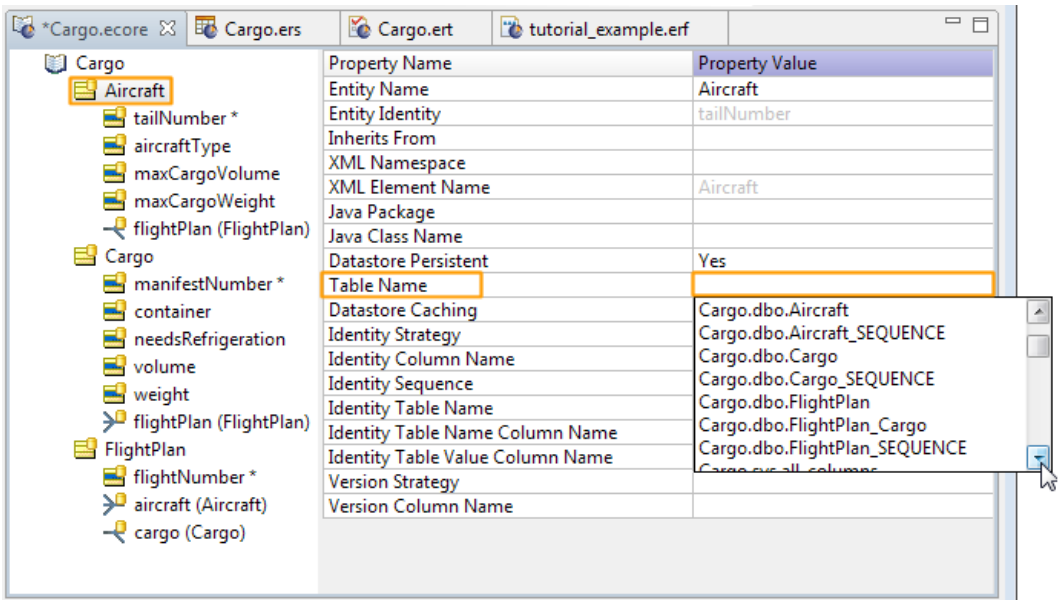


It is also possible for an external database to contain tables or fields not mapped to Vocabulary entities and attributes - these terms are simply excluded from the Vocabulary.

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:

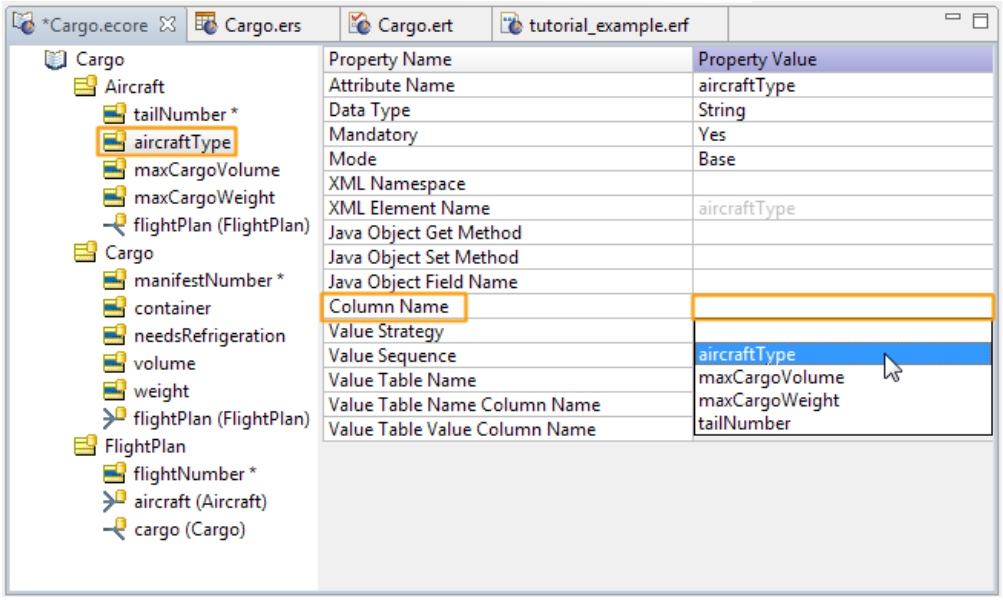
Figure 45: Manual Mapping of Vocabulary Entity



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 46: Manual Mapping of Vocabulary Attribute



Note: Handling data in a `CHAR` database column

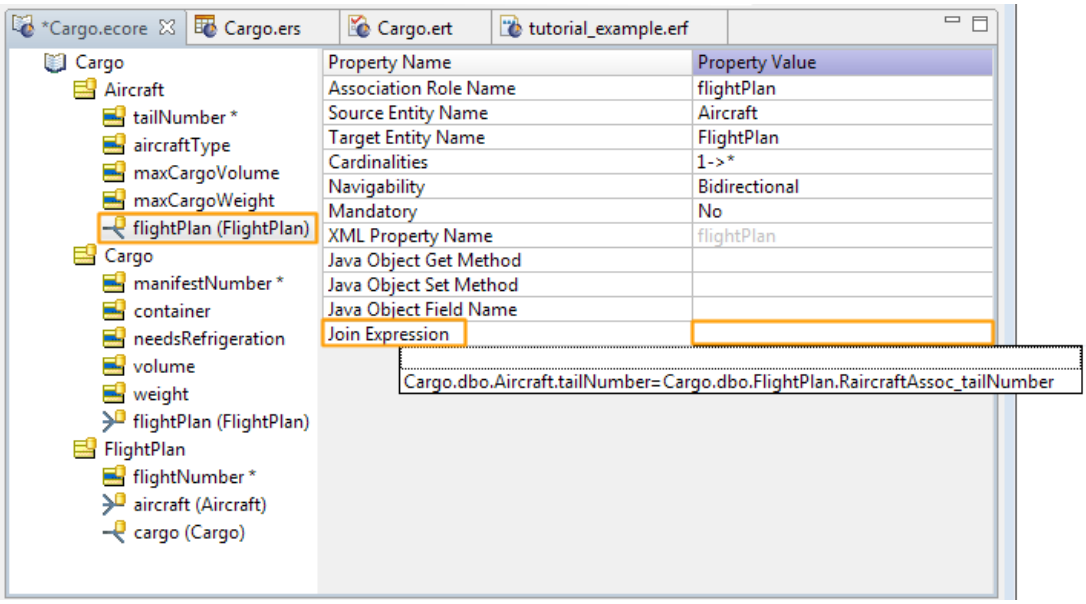
The database column type `CHAR` has a constant length. When a Corticon string attribute is mapped to such a column, the string retrieved from the database always has the length that is specified in the database definition. When a string shorter than the specified length is assigned to the attribute, the database adds spaces at the end of the string before storing it in the database. When the attribute is retrieved from the database, the value returns with the padded spaces at the end of the string.

If this is not the intended behavior, change the database type for the column from `CHAR` to variable-length character data type. If the database schema cannot be changed, either use a `trimSpace` operator to strip the trailing spaces from the returned attribute value, or redefine the query string to allow for its full length including added spaces.

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:

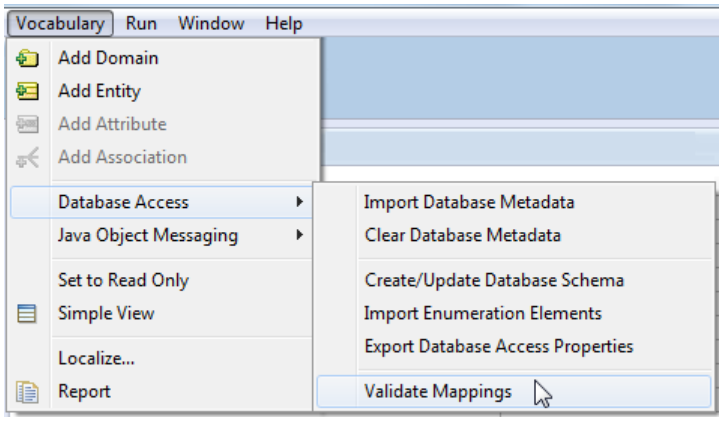
Figure 47: Manual Mapping of Vocabulary Association



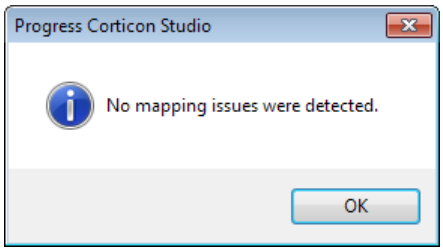
Validating database mappings

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:

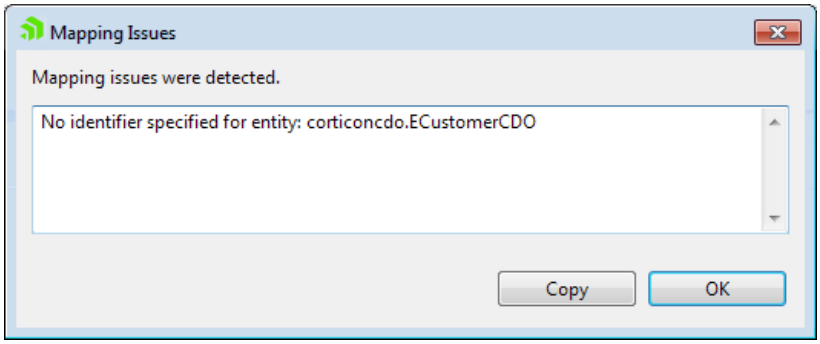
Figure 48: Validate Database Mappings Option



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*

Metadata for Datastore Identity in XML and JSON Payloads

When Element attributes have extra information at the Entity Element level, data such as the datastore identity requires special handling as metadata because it is not an attribute in the Vocabulary. It is invalid to declare the datastore identity as an Element, as shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest decisionServiceName="MyDS">
  <WorkDocuments>
    <TestEntity1 id="TestEntity1_id_1">
      <databaseid>1</databaseid>   this is incorrect
      <testBoolean xsi:nil="true" />
      <testDate xsi:nil="true" />
      <testDateTime xsi:nil="true" />
      <testDecimal xsi:nil="true" />
      <testInteger xsi:nil="true" />
      <testString xsi:nil="true" />
      <testTime xsi:nil="true" />
    </TestEntity1>
  </WorkDocuments>
</CorticonRequest>
```

Adding Datastore Identity to an XML Payload

For an XML payload, `databaseid` is placed inside the Element, as shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest decisionServiceName="MyDS">
  <WorkDocuments>
    <TestEntity1 databaseid="1" id="TestEntity1_id_1">
      <testBoolean xsi:nil="true" />
      <testDate xsi:nil="true" />
      <testDateTime xsi:nil="true" />
      <testDecimal xsi:nil="true" />
      <testInteger xsi:nil="true" />
      <testString xsi:nil="true" />
      <testTime xsi:nil="true" />
    </TestEntity1>
  </WorkDocuments>
</CorticonRequest>
```

Adding Datastore Identity to a JSON Payload

In JSON formatting, the `#datastore_id` is placed in the `__metadata` section of the Entity, as shown:

```
{
    "Objects": [{
        "testDate": null,
        "testDecimal": null,
        "testDateTime": null,
        "testString": null,
        "testBoolean": null,
        "testInteger": null,
        "testTime": null,
        "__metadata": {
            "#id": "TestEntity1_id_1",
            "#type": "TestEntity1",
            "#datastore_id": "1"
        }
    }]
}
```

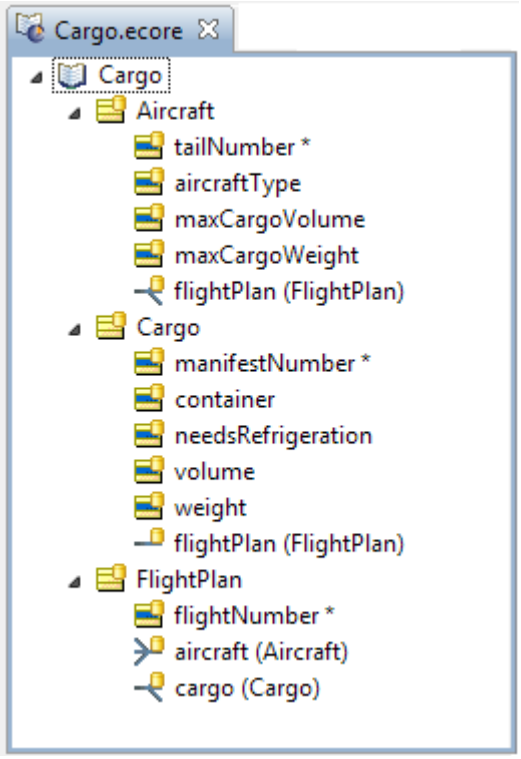
For more information about datastore identity, see the topic Identity strategies on page 92.

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.

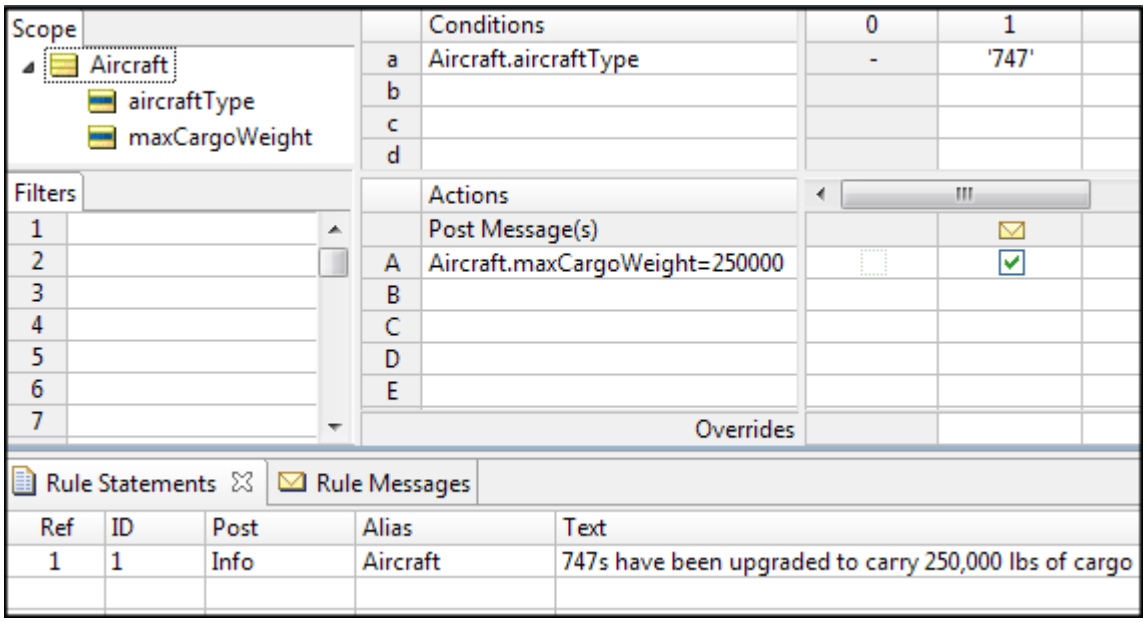
However, when EDC is used, data management becomes more complicated. How is data in the database synchronized with the data contained in the request payload and data produced or updated by Decision Service execution? Using several scenarios, this section describes the algorithms used by Corticon Server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

Figure 49: `Cargo.ecore` with Database Connection and Mappings



The sample Rulesheet we will use is defined as shown:

Figure 50: Sample Rulesheet for Synchronization Examples



Note: The RDBMS data in this section was established in the *EDC Tutorial* into a Microsoft SQL Server 2014 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:

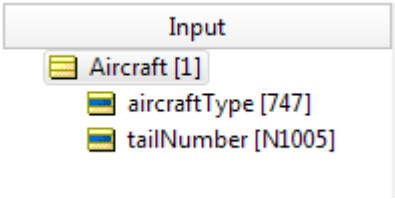
Figure 51: Initial State of Database Table dbo_Aircraft

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

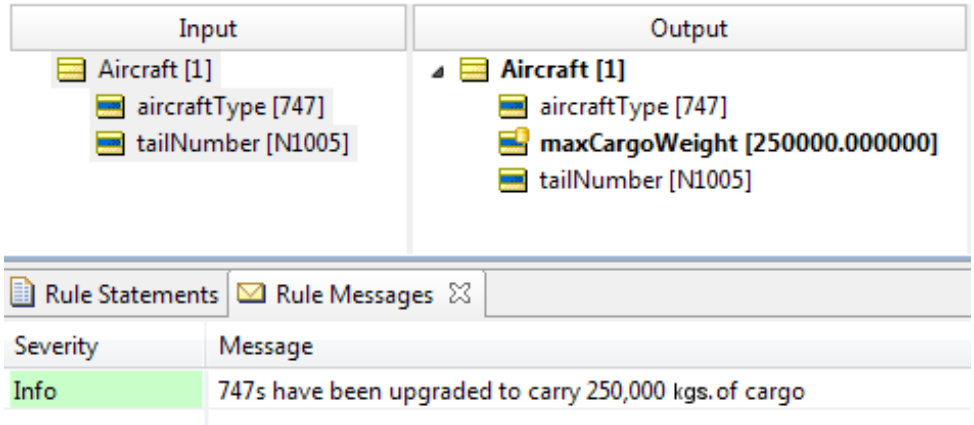
And the Studio Input Ruletest is as shown in the following figure.

Figure 52: 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 53: 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:

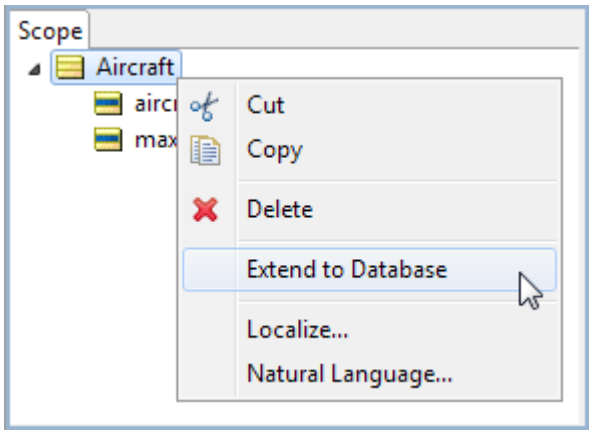
Figure 54: Final State of Database Table AIRCRAFT

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

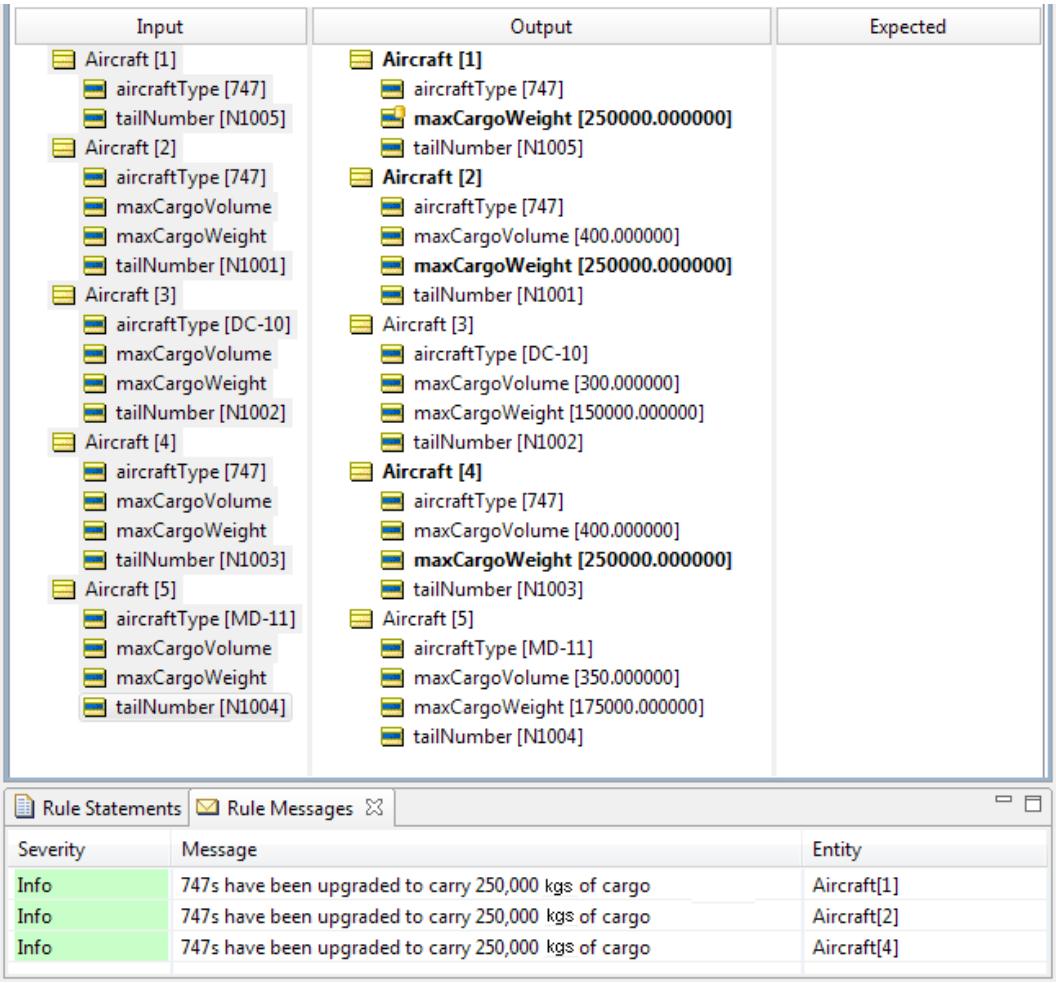
This scenario assumes the rule shown in Sample Rulesheet for Synchronization Examples makes use of an alias extended to the database. By placing the `Aircraft` Entity in the Scope of Rulesheet, we can right-click on `Aircraft` and then choose **Extend to Database** as shown:



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 55: Results Ruletest Showing Re-Binding

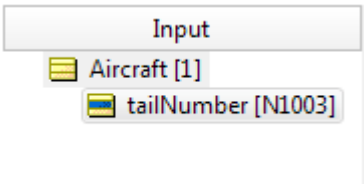


Payload Contains Existing Database Record

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.

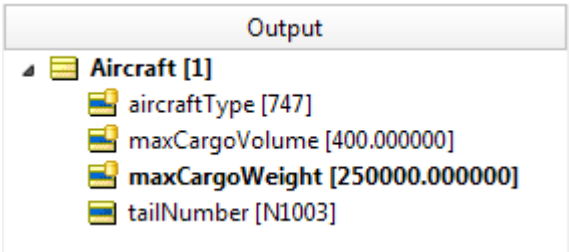
Fortunately, a record with this value exists in the database table, so when the Test is executed, a query to the database successfully retrieves the necessary data.

Figure 56: Ruletest Input with Existing Record



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

Figure 57: 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 58: Final State of Database Table AIRCRAFT

dbo.Aircraft

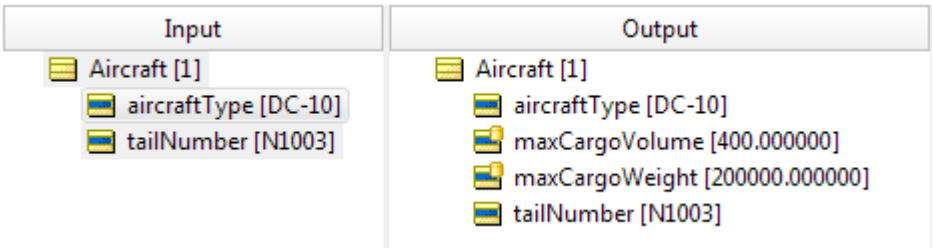
tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
N1001	747	400.00	200000.00
N1002	DC-10	300.00	150000.00
N1003	747	400.00	200000.00
N1004	MD-11	350.00	175000.00
NULL	NULL	NULL	NULL

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?

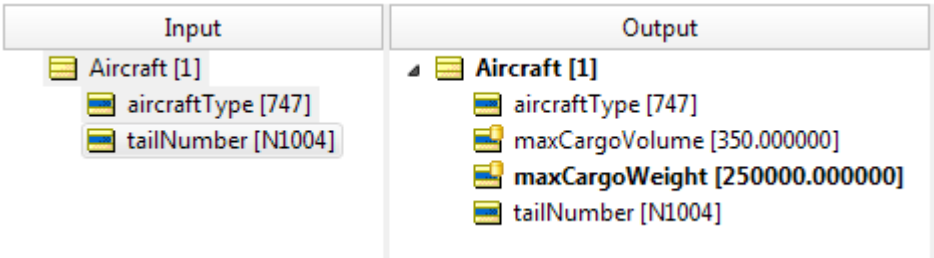
Studio still performs a query to the database because it has the necessary key information in the provided `tailNumber`. When the query returns with an `aircraftType` of `747`, the Synchronization algorithm decides that the data in the Input Ruletest has priority over the retrieved data – for the purposes 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 59: 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 60: 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*.

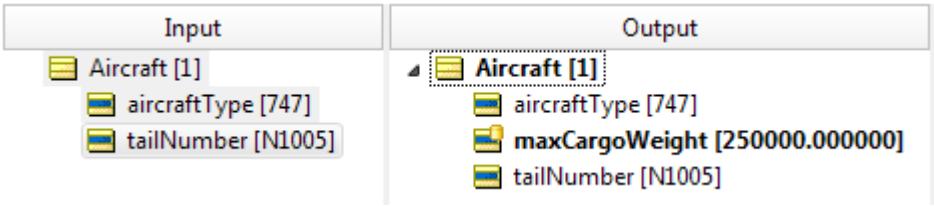
We'll use the same batch of examples as before to discuss the synchronization performed by Studio and Server when set to **Read/Update** mode for a Ruletest by selecting **Ruletest > Testsheet>Database Access>Read/Update** from the Studio's menubar (the Ruletest must be the active Studio window).

Payload contains a New Record not in the Database

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

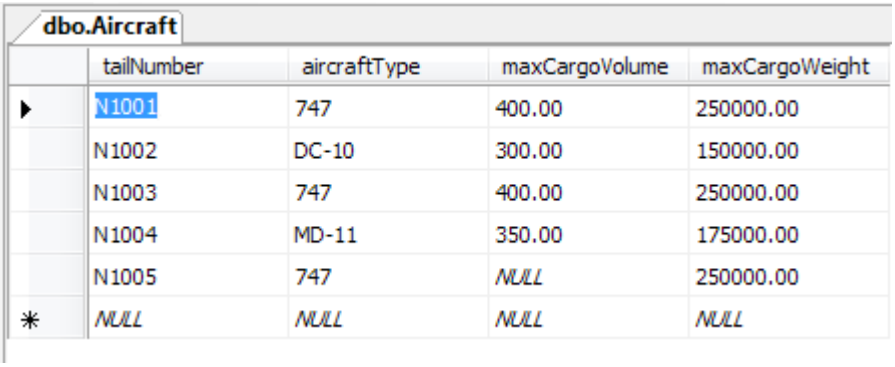
As before, no such record is present in the table so all the data required by the rule must be present in the Input section. Fortunately, in this case, the required `aircraftType` data is present, and the rule fires, as shown:

Figure 61: 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 62: Final State of Database Table Aircraft



dbo.Aircraft

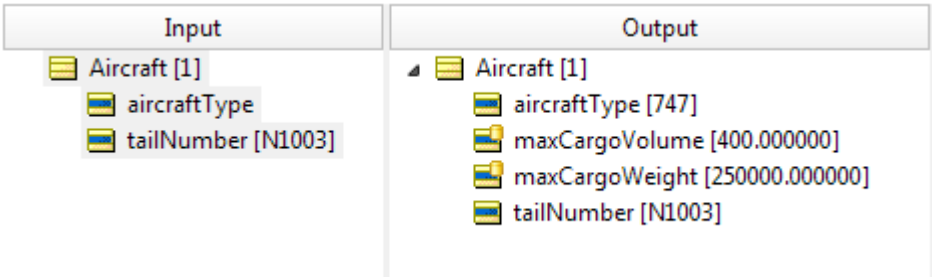
tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
N1001	747	400.00	250000.00
N1002	DC-10	300.00	150000.00
N1003	747	400.00	250000.00
N1004	MD-11	350.00	175000.00
N1005	747	NULL	250000.00
NULL	NULL	NULL	NULL

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 63: 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 64: 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 65: 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 66: Ruletest with Existing Record

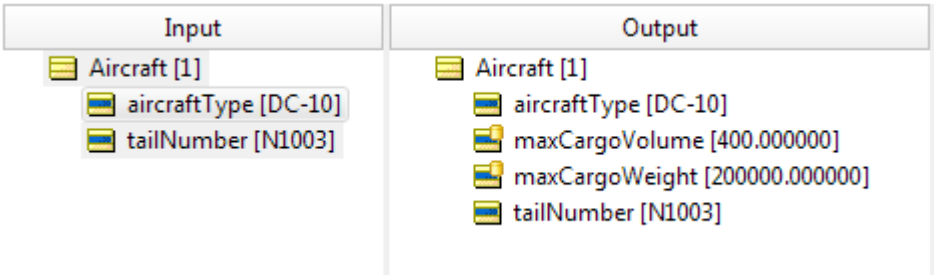
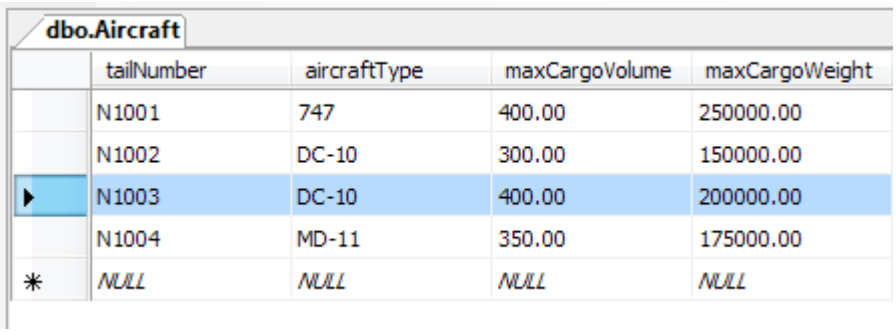


Figure 67: Final State of Database Table `Aircraft`



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 68: Ruletest with Existing Record

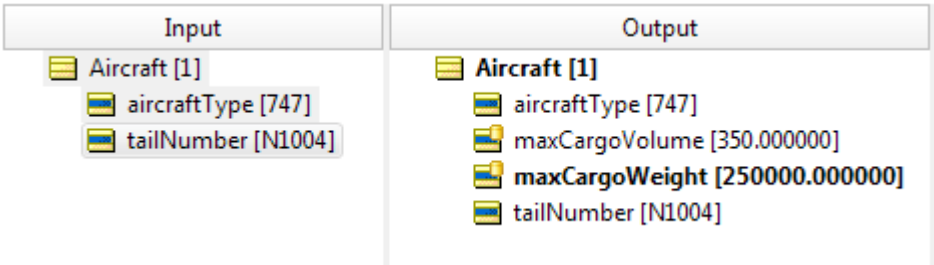
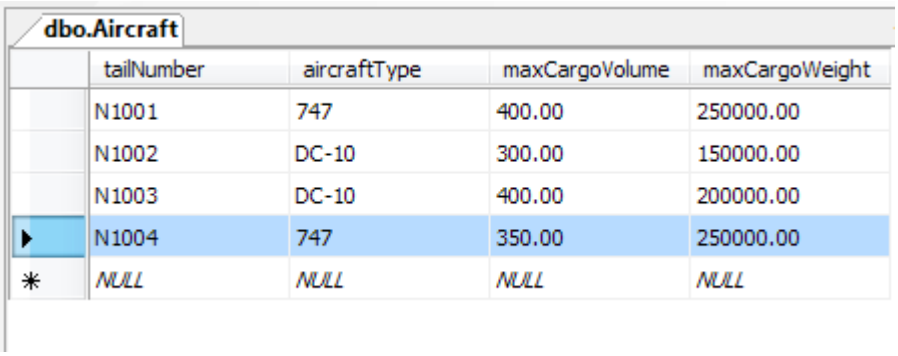


Figure 69: Final State of Database Table `Aircraft`



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.

Importing an attribute's possible values from database tables

A database connection can also provide designers and testers of Rulesheets and Ruletests with lists of enumerations, also known as *possible values*. While these lists can be created and maintained by hand on the Custom Data Types tab of a Vocabulary, you can retrieve lists from the connected database.

Consider the general behavior of enumerations, especially when retrieving labels and values from a database:

- There can be only one instance of any label and any value in the list, whether created manually or imported. An exception will make the Vocabulary invalid. The database retrieval will work as expected but you will have to groom the results to make the lists valid. You can get optimal results when your database source prevents duplicates in the table columns you are using for your values or label-value pairs.
- If you chose a label in a Rulesheet and that label is no longer available after an update, an error will occur. Any Rulesheet expressions that refer to the defunct label will be flagged as invalid. You must update the Rulesheet expressions to correct the problem.
- If you chose a label in a Rulesheet and that label takes on a different value after an update, the current value is what is evaluated.
- The value assigned - whether directly or as the label's value - at the time of deployment does not change thereafter on the server.

It is good practice to ensure that the data types of the retrieved values in the database are consistent with the Custom Data Type, and then extend the corresponding base data value in the attribute.

Procedures

The steps to implement custom data types retrieved from a database are, in summary, as follows:

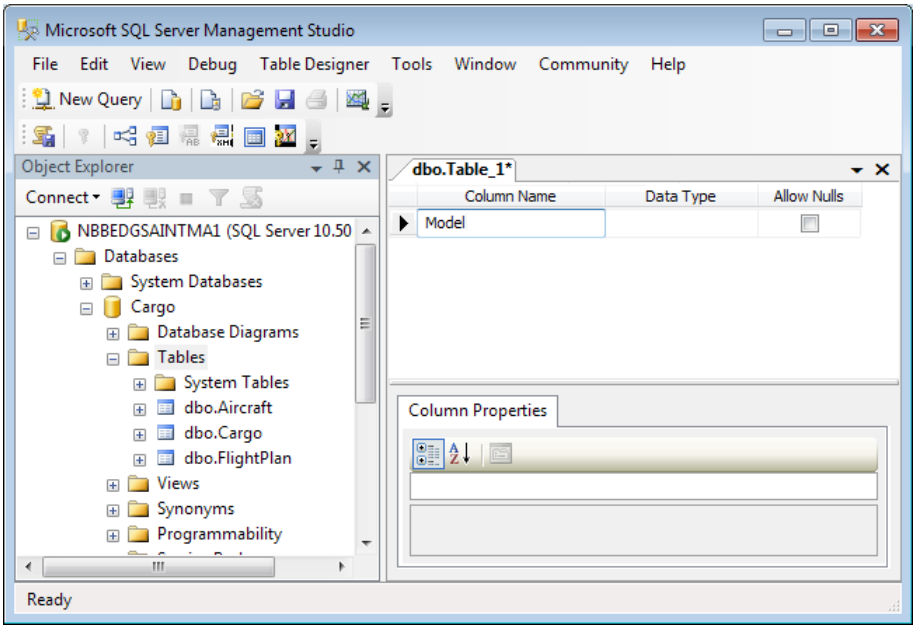
- A - Create or locate the database table and columns you want to retrieve.
- B - Verify the database connectivity, and then import its metadata.
- C - Define the Custom Data Type lookup information.
- D - Import the enumeration elements.
- E - Check the lists for duplicates.
- F - Set the Data Type of appropriate attributes to the Custom Data Type.
- G - Verify that the list functions correctly.

A - Create or locate the database table and columns you want to retrieve.

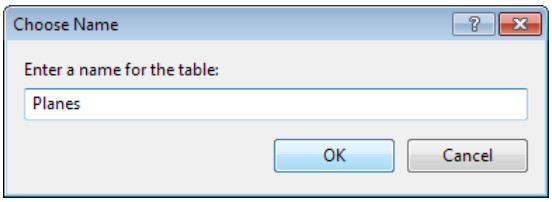
Note: This step uses the procedures detailed in Populating the database when we set up the database.

Continuing with our `Cargo` database in SQL Server, we'll add two tables to demonstrate both value-only and label+value enumerations:

1. Start the SQL Server Management Studio, and then expand the tree for **Databases : Cargo : Tables**. Right-click on **Tables** and choose **New Table**. Enter `Model` as the only column name, as shown:



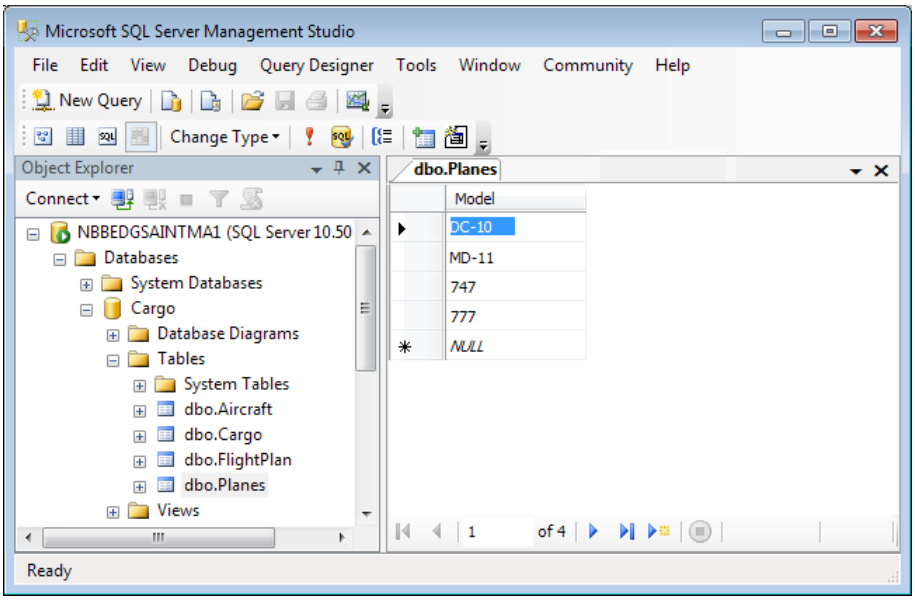
2. Choose the menu command **File > Save Table_1**, enter the name `Planes`, and then click **OK**.



3. Create another table, now with two columns named `planeCarrier` and `planeID`, saving it as `Carrier`.
4. Click **New Query**, copy/paste the following text, and then click **Execute**.

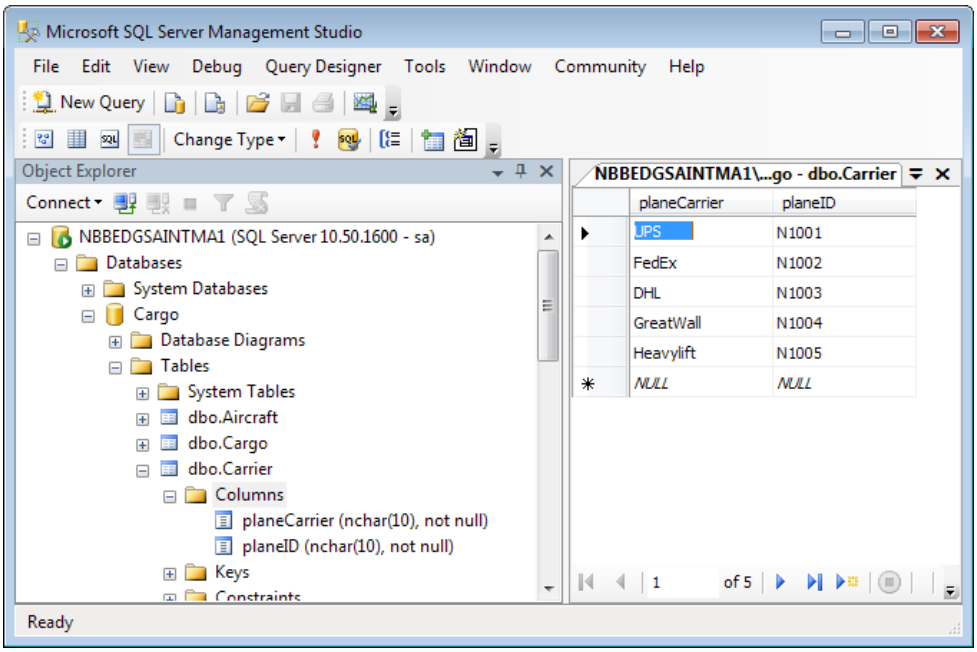
```
INSERT INTO Cargo.dbo.Planes (Model) VALUES ('DC-10');
INSERT INTO Cargo.dbo.Planes (Model) VALUES ('MD-11');
INSERT INTO Cargo.dbo.Planes (Model) VALUES ('747');
INSERT INTO Cargo.dbo.Planes (Model) VALUES ('777');
INSERT INTO Cargo.dbo.Carrier (planeCarrier,planeID) VALUES ('UPS','N1001');
INSERT INTO Cargo.dbo.Carrier (planeCarrier,planeID) VALUES ('FedEx','N1002');
INSERT INTO Cargo.dbo.Carrier (planeCarrier,planeID) VALUES ('DHL','N1003');
INSERT INTO Cargo.dbo.Carrier (planeCarrier,planeID) VALUES ('GreatWall','N1004');
INSERT INTO Cargo.dbo.Carrier (planeCarrier,planeID) VALUES ('Heavylift','N1005');
```

5. In the tree, right-click on **dbo.Planes**, and then choose **Edit Top 200 Rows**.



The **Planes** data is as we intended. It is ready for our use in the Corticon Studio.

6. Similarly, right-click on **dbo.Carrier**, and then choose **Edit Top 200 Rows**.

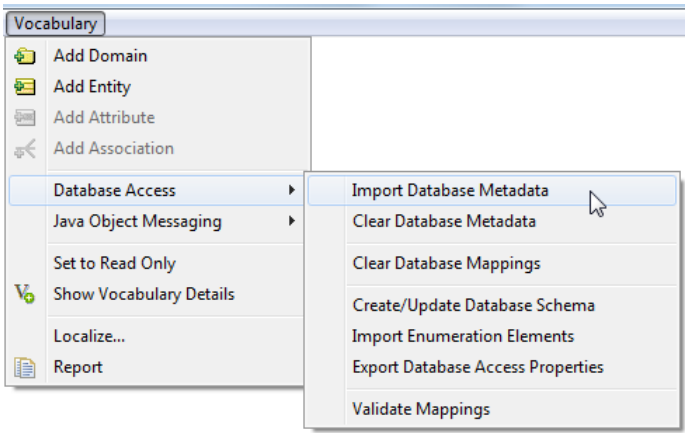


The **Carrier** data is as we intended. It is ready for our use in the Corticon Studio.

B - Verify the database connectivity, and then import its metadata.

We want to bring the information about the table definitions into the Studio:

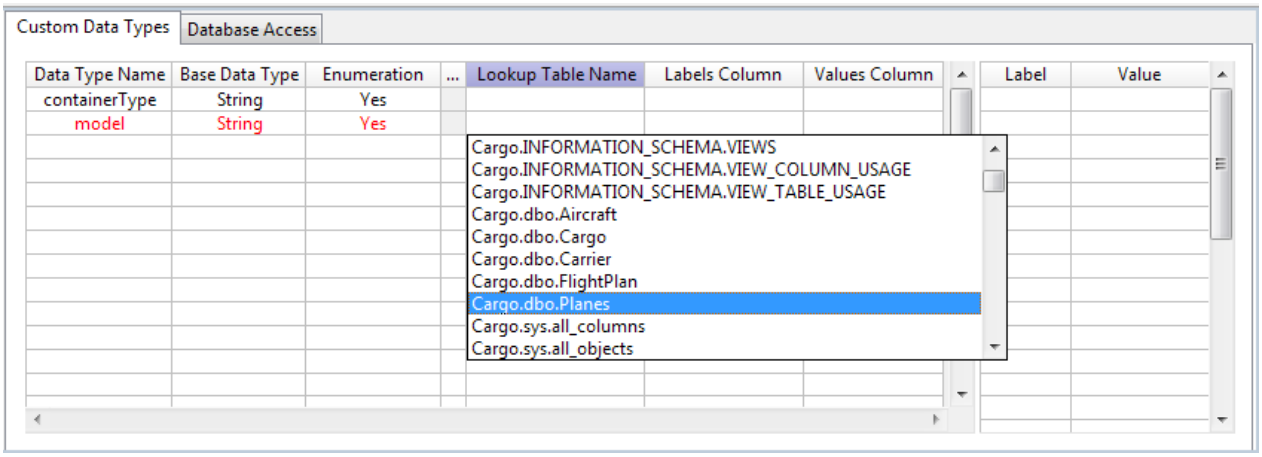
1. In Corticon Studio, confirm that you have the same good connection you achieved in Connecting a Vocabulary to a database on page 108
2. With `Cargo.ecore` open its editor, choose the menu command **Database Access > Import Database Metadata**, as shown:



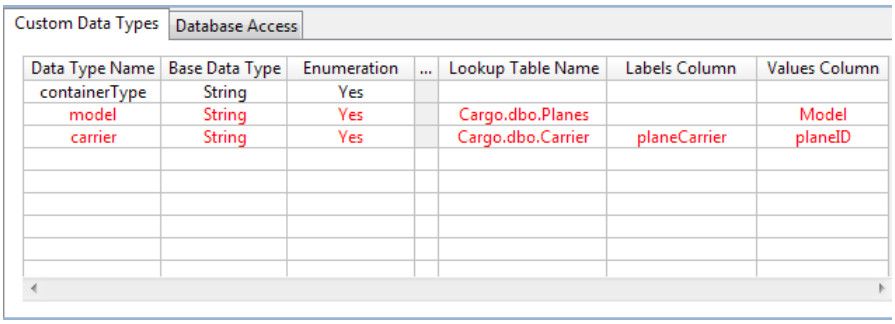
C - Define the Custom Data Type lookup information.

We now can specify how we want to use the data and then bind it to the appropriate database table and columns:

1. Click on `Cargo` to get to its top level, and then select the **Custom Data Types** tab.
2. Click on the next empty row, enter `model` as the Data Type Name, select `String` as the Data Type, and `Yes` as the Enumeration.
3. Click on the Lookup column in the row to expose its dropdown, and then choose `Cargo.dbo.Planes` that we imported in the database metadata.



4. We are using a values-only lookup, click on the row's Values Column to select its one database column, `Model`:
5. For the other table, click on the next empty row, enter `carrier` as the Data Type Name, select `String` as the Data Type, and `Yes` as the Enumeration.
6. Click on the Lookup Table Name in the row to expose its dropdown, and then choose `Cargo.dbo.Carrier` that we imported in the database metadata.
7. We are using a label-values lookup, so click on the row's Labels Column to select planeCarrier, and then in the Values Column to select `planeID`:

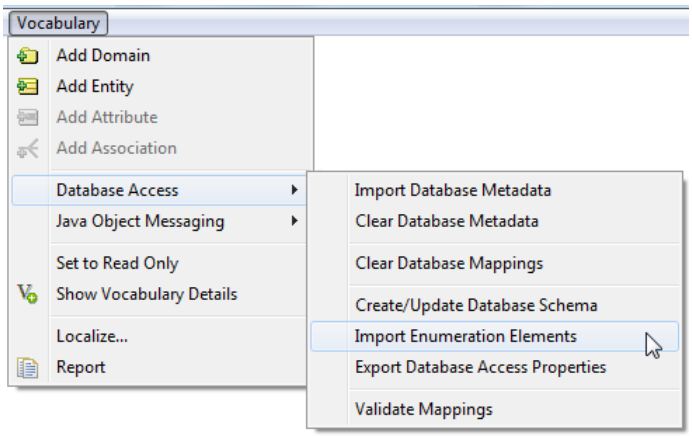


Everything we have entered is red! That's because Studio has no data for either of these enumeration sets.

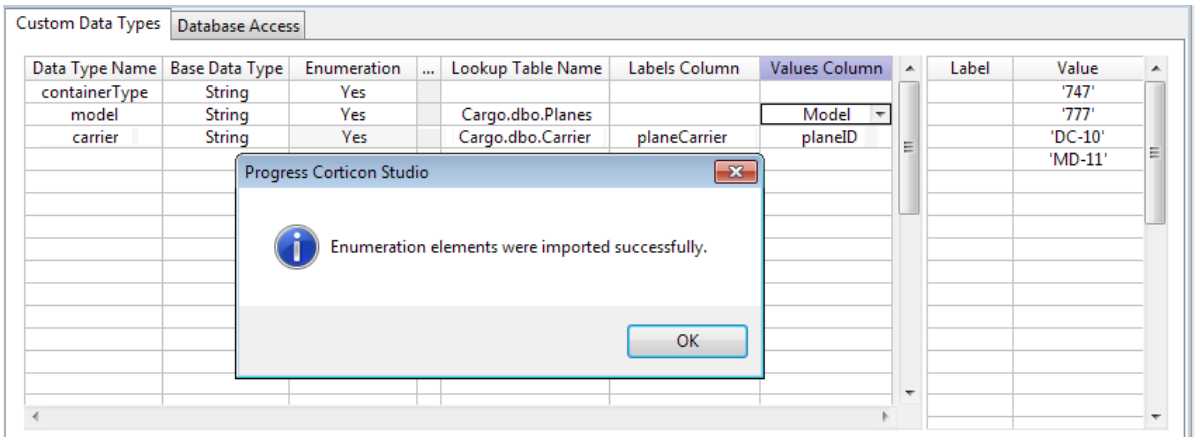
D - Import the enumeration elements.

Once you have defined the database table and columns you want, you can retrieve the data:

1. Choose the menu command **Database Access > Import Enumeration Elements**, as shown:



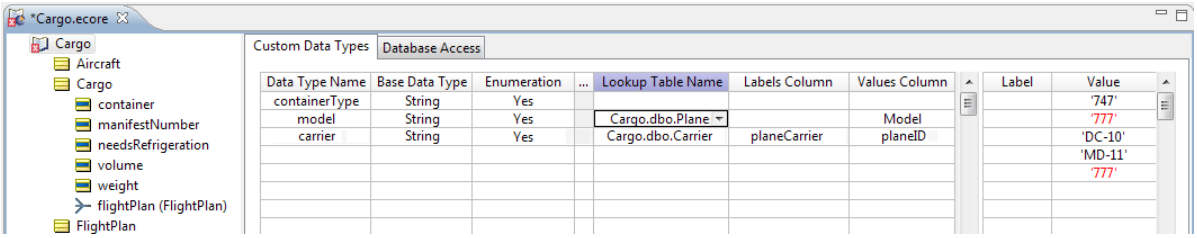
2. The retrieved values are displayed in the associated Labels and Values window to the right, as shown for the `model`:



E - Check the lists for duplicates.

Unless you enforced uniqueness in the source database. To demonstrate what happens, we'll add an existing value to the `model` enumerations.

1. In the Values retrieved column, enter a new value that is already there, such as `777`, as shown:



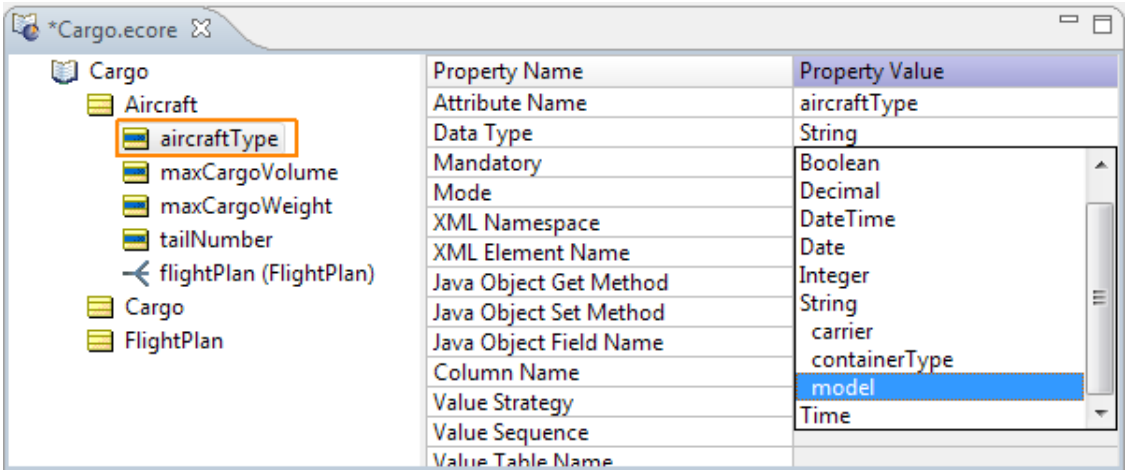
The duplicates are both highlighted in red, and the `Cargo.ecore` file is marked as being in an error state.

2. Remove the line (or change it to something unique) and the Vocabulary is again valid.

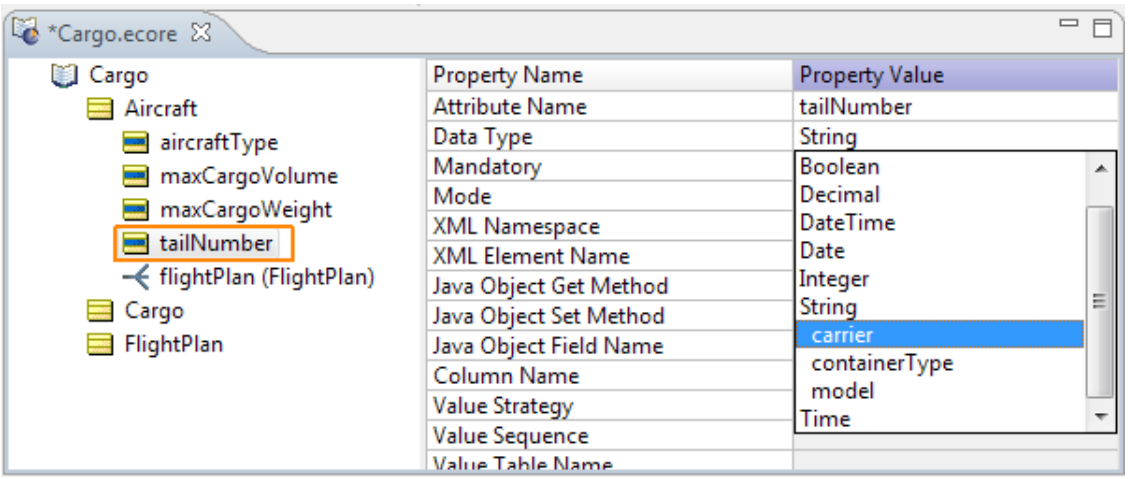
F - Set the Data Type of appropriate attributes to the Custom Data Type.

With our enumeration lists imported from the database and verified as free of duplicate labels or values, we can link them to the attributes that will use them:

1. Aircraft.aircraftType:



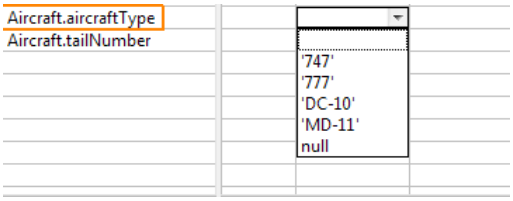
2. Aircraft.tailNumber:



G - Verify that the list functions correctly.

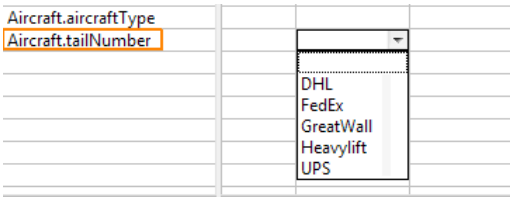
To verify that the lists perform as expected, use them in a Rulesheet or Ruletest :

1. In a Rulesheet Actions area, enter two new lines, one with the attribute syntax `Aircraft.aircraftType` and the other with `Aircraft.tailNumber`, as shown:
2. Click on the `aircraftType` where it intersects with column 1, as shown:



The pulldown displays our imported values, as well as blank and `null`.

3. Click on the `tailNumber` where it intersects with column 1, as shown:



The pulldown displays our imported label, as well as blank. The label is a place holder for its value.

Note: For more information about enumerations and retrieving values from databases, see:

- *"Enumerated values" in the Quick Reference Guide.*
- *"Enumerations retrieved from a database" in the Rule Modeling Guide*

Working with EDC in Corticon Server

Corticon Server requires that you have a copy of a license that enables EDC on each in-process and web service server location that intends to use EDC.

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

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

See the tutorials *Modeling Progress Corticon Rules to Access a Database using EDC* and *Connecting a Progress Corticon Decision Service to a Database using EDC*, available at https://www.progress.com/tutorials/corticon, for more information.

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.

How EDC handles transactions and exceptions

Here are a few points to note about Corticon's Enterprise Data Connector:

- Each Decision Service call is one database transaction. Transactions are not per operation, per Rulesheet, or per Ruleflow. Corticon does not currently provide for configuration of transaction management.

- The default transaction isolation level in Corticon EDC is the same as the default transaction isolation level of the database to which it is connected.
- When an exception occurs, the database transaction is rolled back, and the database reverts to the same state as before the Decision Service was called.

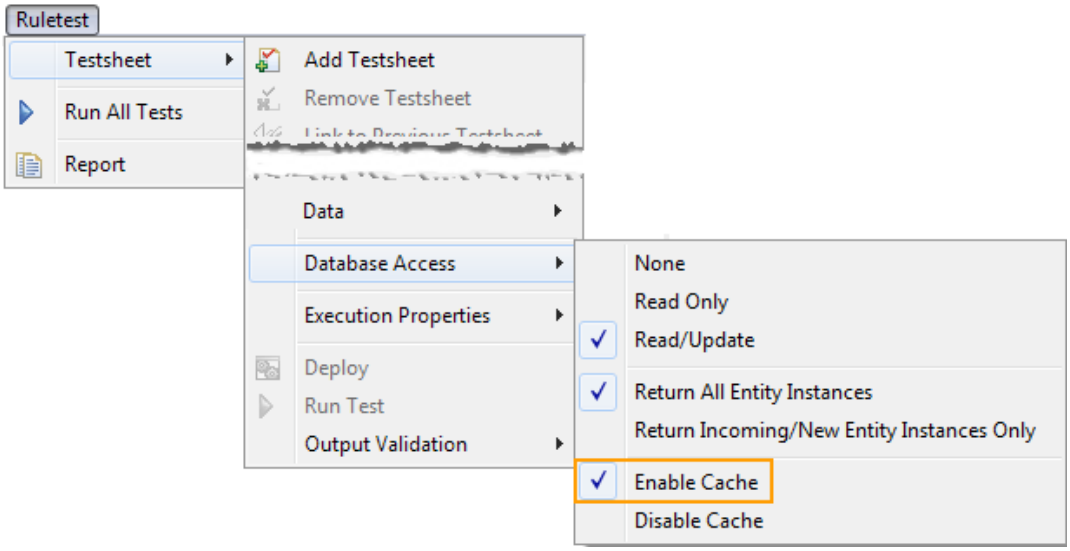
Working with database caches

Corticon's EDC provides functionality for enhanced database caching at runtime. Its cache is temporary data that duplicates data located in a database so that it can be repeatedly accessed with minimal costs in terms of time and resources. If an application must be certain not to get stale data, then it should not use caching. Caching is best used for reference data such as tax or actuarial tables.

What gets cached is based on settings in a project's Vocabulary and Rulesheets (see the topic *"Defining database caching in Studio" in the Rule Modeling Guide*). Ruletests and deployed Decision Services let you choose to enable the requested caching. The first cache usage takes some overhead to establish the cache so that subsequent test runs get the benefit of very fast performance. When Studio or Server restarts, its in-memory cache(s) and on-disk cache files are cleared.

Enabling caching on a Studio Ruletest

Once you have Rulesheets and a Vocabulary that are prepared for database caching, choosing to enable cache on the Studio will perform the caching functions. To enable caching on the Ruletest, choose the **Ruletest** menu command **Testsheet > Database Access > Enable Cache**, as shown:



When the Ruletest runs, it records its default configuration in the Studio's one DB Cache properties file, `[CORTICON_WORK_DIR]\etc\ehcache.xml`, which it creates during the first run. If the memory cache overflows, the cache files are created on the Studio machine. While the default settings should provide good performance, you can adjust the cache configuration as discussed later in this topic.

Enabling caching on deployed Decision Services

On Corticon Servers, no Decision Services are enabled for database caching by default. You set the preference to enable caching on each Decision Service during its deployment process, as available in these various deployment techniques:

- **Using Corticon Web Console** - Adding or editing a Decision Service in an application, as described in the topic *"Decision Service and Applications" in the Web Console Guide.*
- **Using scripts testServerAxis and testServer** - Command 103 includes a parameter for enabling caching. Those commands use the underlying **Using the Corticon Server APIs**. For more information, see the topic Using Server API to compile and deploy Decision Services on page 60.

On Corticon Servers, each Decision Service records its respective default configuration in its properties file, `[CORTICON_WORK_DIR]/etc/ehcache_<DSName>_v<M.m>.xml` where *<DSName>_v<M.m>* is the named and versioned Decision Service. For example, **`[CORTICON_WORK_DIR]\etc\ehcache_Cargo_v0.16.xml`**. Each Decision Service will also maintain its own cache, and cached data is never shared between Decision Services. Undeploying a Decision Service immediately clears its cache in memory and on disk.

Setting properties in a cache configuration

Once the cache configuration file has been created for Studio or for a deployed Decision Service, you can adjust its properties such as controlling the refresh intervals. The default properties and values are:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
  <diskStore path="[CORTICON_WORK_DIR]\Server\etc\ehcache.xml" />
  <defaultCache
       overflowToDisk="true"
  timeToLiveSeconds="120"
  timeToIdleSeconds="120"
  eternal="false"
  maxElementsInMemory="1000" />
</ehcache>
```

where:

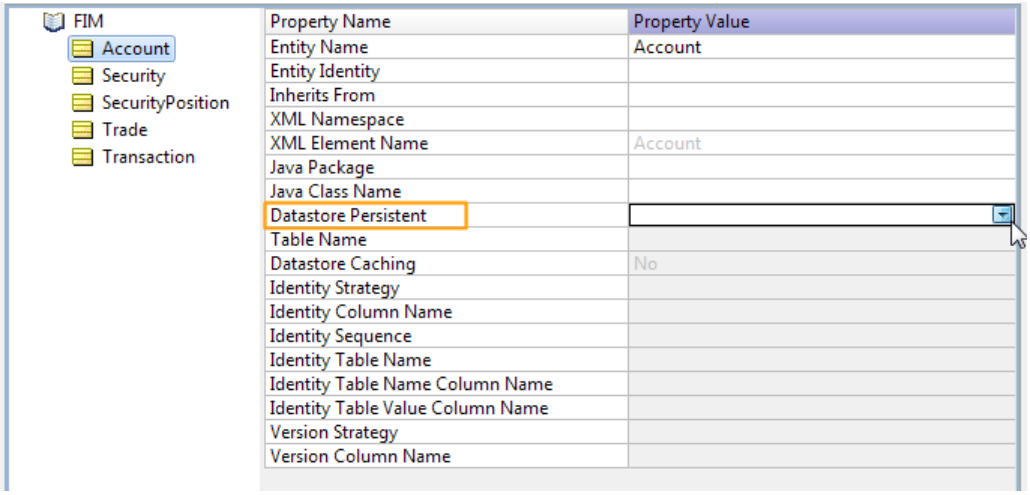
- `diskStore path` is the location where overflows to disk are written.
- `overflowToDisk` sets whether elements can overflow to disk when the in-memory cache has reached the `maxElementsInMemory` limit.
- `timeToLiveSeconds` is the maximum number of seconds an element can exist in the cache regardless of use. The element expires at this limit and will no longer be returned from the cache. If the value is 0, no TTL eviction takes place (infinite lifetime).
- `timeToIdleSeconds` is the maximum number of seconds an element can exist in the cache without being accessed. The element expires at this limit and will no longer be returned from the cache. If the value is 0, no TTI eviction takes place (infinite lifetime).
- `eternal` sets whether elements are eternal. When eternal is `true`, timeouts are ignored and elements are never expired.
- `maxElementsInMemory` is the maximum number of objects that will be created in memory. When set to 0, there is no limit.

Note: For more information about the settings and behaviors of Corticon's advanced EDC caching, see the EHCACHE documentation.

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`.

10

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
- About Working Memory
- Ruleflow compilation into an EDS file
- Multi-threading, 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 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, each of which might reference a different Vocabulary.

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 installation options described in Choose the deployment architecture on page 11, Option 1 and 2 express the payload as an XML or JSON document. In Option 3, the 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.

Ruleflow compilation into an EDS file

Ruleflows 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\CcServerSandbox\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 Ruleflows 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, Ruleflows were compiled during the Corticon Studio's **Save** processing of `.ers` and `.erf` assets .

Multi-threading, concurrency reactors, and server pools

Multiple Decision Services place their requests in a queue for processing. Server-level thread pooling is implemented by default, using built-in Java concurrent pooling mechanisms to control the number of concurrent executions. This design allows the server to determine how many concurrent requests are to be processed at any one time.

Implementation of an Execution Queue

Each thread coming into the Server gets an available Reactor from the Decision Service, and then the thread is added to the Server's Execution Thread Pooling Queue, or, simply put, the ***Execution Queue***. The Execution Queue guarantees that threads do not overload the cores of the machine by allowing a specified number of threads in the Execution Queue to start executing, while holding back the other threads. Once an executing thread completes, the next thread in the Execution Queue starts executing.

The Server will discover the number of cores on a machine and, by default, limit the number of concurrent executions to that many cores, but a property can be set to specify the number of concurrent executions. Most use cases will not need to set this property. However, if you have multiple applications running on the same machine as Corticon Server, you might want to set this property lower to limit the system resources Corticon uses. While this tactic might slow down Corticon processing when there is a heavy load of incoming threads, it will help ensure Corticon does not monopolize the system. Conversely, if you have Decision Services which make calls to external services (such as EDC to a database) you may want to set this property higher so that a core is not idle while a thread is waiting for a response.

Ability to Allocate Execution Threads

The Corticon Server takes each thread (regardless of which Decision Service the thread is executing against), and then adds the thread to the Execution Queue in a first-in-first-out strategy. While that generally satisfies most use cases, you might want more control over which Decision Services get priority over other Decision Services. For that, you first set the Server property `com.corticon.server.decisionservice.allocation.enabled` to `true`, and then set the maximum number of execution threads (`maxPoolSize`) for each specified Decision Service in the Execution Queue. Once you set the allocation on *every* Decision Service, the Server will try to maintain corresponding allocations of execution threads from the Decision Services inside the Execution Queue.

Once the property is set to `true`, the Decision Services will allocate based on the `maxPoolSize` that was assigned when the Decision Service was deployed. You can then dynamically change a Decision Service's `maxPoolSize`, depending on how you deployed that Decision Service:

- If deployed using the API method `ICcServer.addDecisionService(..., int aiMaxPoolSize, …)`, then the `maxPoolSize` can be updated using `ICcServer.modifyDecisionServicePoolSizes(int aiMinPoolSize, int aiMaxPoolSize)`.
- If deployed using a CDD file, then change the value of `max-size` in the CDD. When the `CcServerMaintenanceThread` detects the change, it will update the Decision Service.

Memory management

Allocation means that you could allocate hundreds of execution threads for one Decision Service. The way Reactors are maintained in each Decision Service, the Server can re-use cached data across all Reactors for the Decision Service. Runtime performance should reveal only modest differences in memory utilization between a Decision Service that contains just one Reactor and another that contains hundreds of Reactors. Because each Reactor reuses cached data, the Server can dynamically create a new Reactor per execution thread (rather than creating Reactors and holding them in memory.) Even when an allocation is set to 100, the Server only creates a new Reactor (with cached data) for every incoming execution thread, up to 100. If there are only 25 execution threads against Decision Service 1, then there are just 25 Reactors in memory. Large request payloads are more of a concern than the number of concurrent executions or the number of Reactors.

Note: In prior releases, you set minimum and maximum pool sizes that the Server would use based on load. As load increased, the Server would allocate more Reactors to the Decision Service Pool (up to Max Pool Size), then, as load decreased, the Server would remove Reactors (down to Min Pool Size). This mechanism attempted to throttle the Server so that it would not run out of memory. Starting in this release, there is no need to decrease the number of Reactors in the Pool because extra Reactors are not actually sitting in the Pool. A new Reactor is created for every execution thread, and -- when the execution is done -- the Reactor is not put back into the Pool for reuse (as it was in previous versions), it just drops out of scope and garbage collection releases its memory.

Related Server properties

The following Server properties let you adjust server executions and allocations:

- Determines how many concurrent executions can occur across the Server. Ideally, this value is set to the number of Cores on the machine. By default, this value is set to `0`, which means the Server will auto-detect the number of Cores on the server.

  ```
  com.corticon.server.execution.processors.available=0
  ```

- This is the timeout setting for how long an execution thread can be inside the Execution Queue. The time starts when the execution thread enters the Execution Queue and ends when it completes executing against a Decision Service. A `CcServerTimeoutException` will be thrown if the execution thread fails to complete in the allotted time. The value is in milliseconds. Default value is `180000` (180000ms = 3 minutes)

  ```
  com.corticon.server.execution.queue.timeout=180000
  ```

- In some cases, you might want to enable Decision Service level allocations to control the number of Decision Service instances that can be added to the queue at a particular time. This will cause prioritizing of one Decision Service over another, making more resources available to that type. To do this, set the property's value to `true`. Default value is `false`

  ```
  com.corticon.server.decisionservice.allocation.enabled=false
  ```

- Once Decision Service allocation is turned on, prioritization of one Decision Service may occur in the Execution Queue. If a particular Decision Service is fully allocated in the Execution Queue, other execution threads for that Decision Service will have to wait until one of the allocated execution Threads completes its execution. The wait time, in getting into the Execution Queue varies, based on load and other Decision Service allocations. You can allow those waiting Threads to timeout if they wait longer than specified. A `CcServerTimeoutException` will be thrown if the execution thread fails to complete in the allotted time. The value is in milliseconds. Default value is `180000` (180000ms = 3 minutes)

  ```
  com.corticon.server.decisionservice.allocation.timeout=180000
  ```

API methods that set and maintain queue allocation

The following `CcServer` API methods let you specify the queue allocation on a Decision Service:

- `addDecisionService` methods that have the `aiMaxPoolSize` parameter.

  Note: The `addDecisionService` methods that previously had `int aiMinPoolSize, int aiMaxPoolSize` parameters were deprecated, and replaced by corresponding `addDecisionService` methods that set only the `int aiMaxPoolSize` parameter. Existing methods that have these signatures will be valid but will ignore the `aiMinPoolSize` parameter. Check any such existing usage to ensure that the `aiMaxPoolSize` value is appropriate as the allocation queue value.

- `modifyDecisionServicePoolSize` methods to have the `int aiMaxPoolSize` parameter.

  Note: The `modifyDecisionServicePoolSizes` methods (note the plural "`Size`**`s`**)" that had the `int aiMinPoolSize, int aiMaxPoolSize` parameters were deprecated.

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 contains multiple Rulesheets, state is preserved across those Rulesheets as the Rulesheets successively execute within the Ruleflow. However, no interaction with the client application occurs between or within Rulesheets. 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 nulled out (garbage collection will remove it from memory). A new Reactor will be created for the next incoming `CorticonRequest`.

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.

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. The file `ServerState.xml`, located in `[CORTICON_WORK_DIR]\{INP|SER}\CcServerSandbox\DoNotDelete`, maintains a record of the Ruleflows 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 Web Console. For example, any properties settings made or `.eds` files deployed using the Console will be lost. If an `autoloaddir` property has been set in your `brms.properties` file, 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, set the Server property `com.corticon.server.serverstate.persistchanges` in your `brms.properties` file.

To allow/prevent Corticon Server reading `ServerState.xml`, set the Server property `com.corticon.server.serverstate.load` in your `brms.properties` file.

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 Server property `com.corticon.ccserver.serviceIntervals`. 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 can be shut off by setting the Server property`com.corticon.ccserver.dynamicupdatemonitor.autoactivate` to `false` in your `brms.properties` override file.

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*.

11

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
- 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 Deployment related files on page 32 for a refresher.

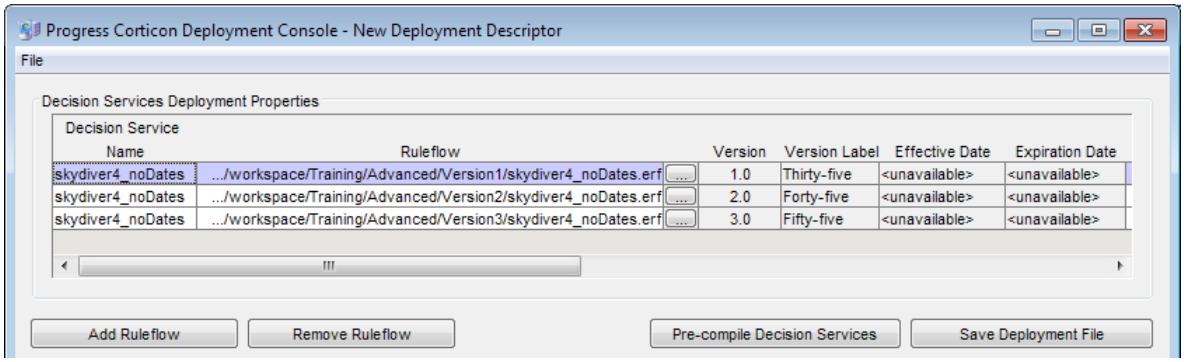
However, the Decision Service Versioning and Effective Dating feature makes an exception to this rule. Decision Services with identical Decision Service Names may be deployed on the same Corticon Server if and only if:

- 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 70: 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.

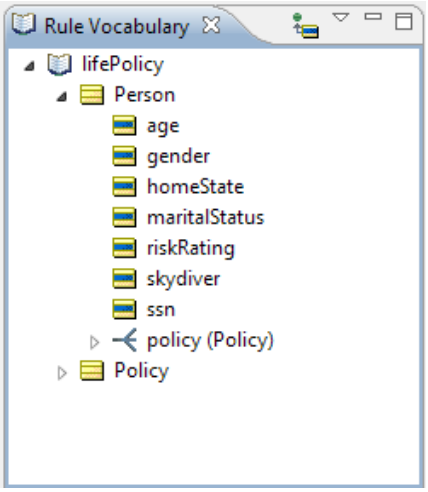
Invoking a Decision Service by version number

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

Creating samples of versioned Ruleflows

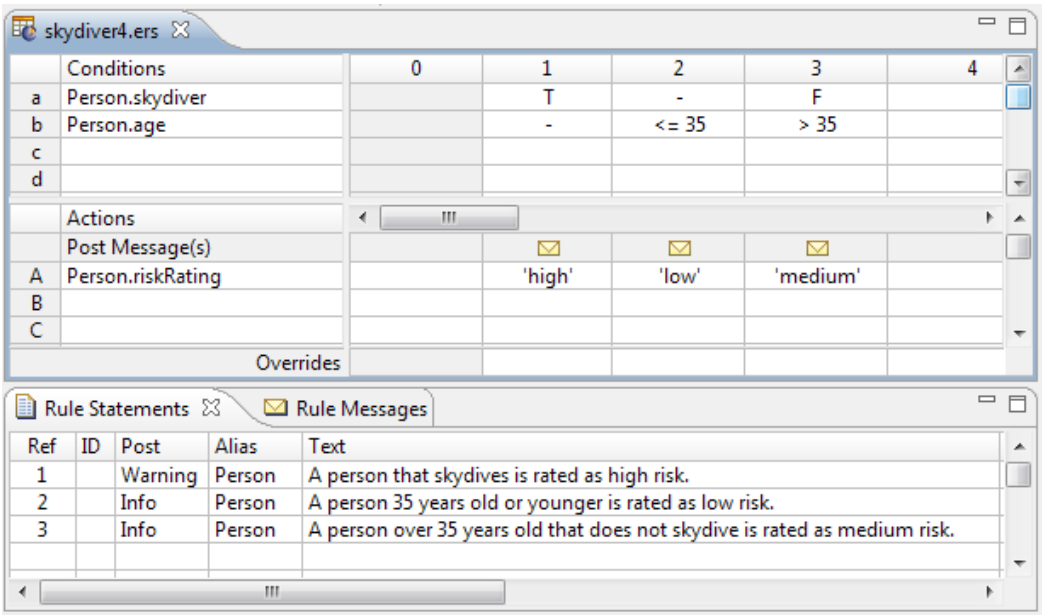
The Ruleflows we will use in this section are based on Rulesheet variations of a single rule. Notice that the only difference between the three Rulesheets is the threshold for the age-dependent rules (columns 2 and 3 in each Rulesheet). The age threshold is `35`, `45`, and `55` for Version 1, 2 and 3, 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 71: Sample Vocabulary for demonstrating versioning



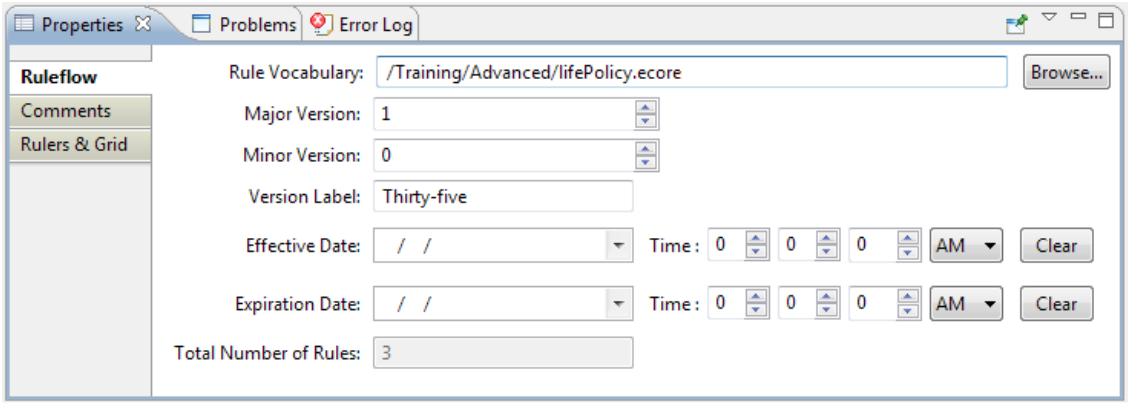
We know we want to have more than one Ruleflow with the same name and differing versions, so we first used **File > New Folder** to place a `Version1` folder in the project. Then we created a Rulesheet for defining our policy risk rating that considers age 35 as a decision point, as shown:

Figure 72: Rulesheet skydiver4.ers in folder Version1



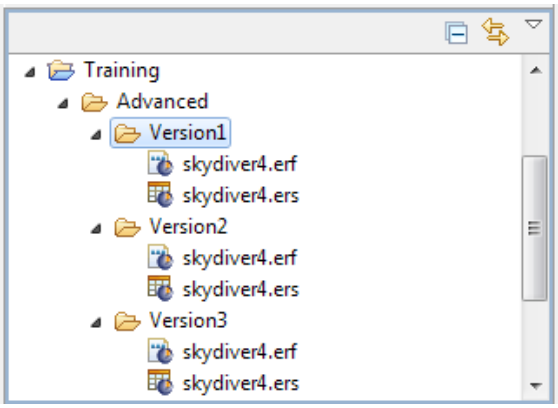
We created a new Ruleflow and added the Version1 `skydiver4.ers` Rulesheet to it. Then we set the Major version to `1` and the Minor version to `0`. The label `Thirty-five` was entered to express the version in natural language.

Figure 73: 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 74: 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 75: Rulesheet skydiver4.ers in folder Version2

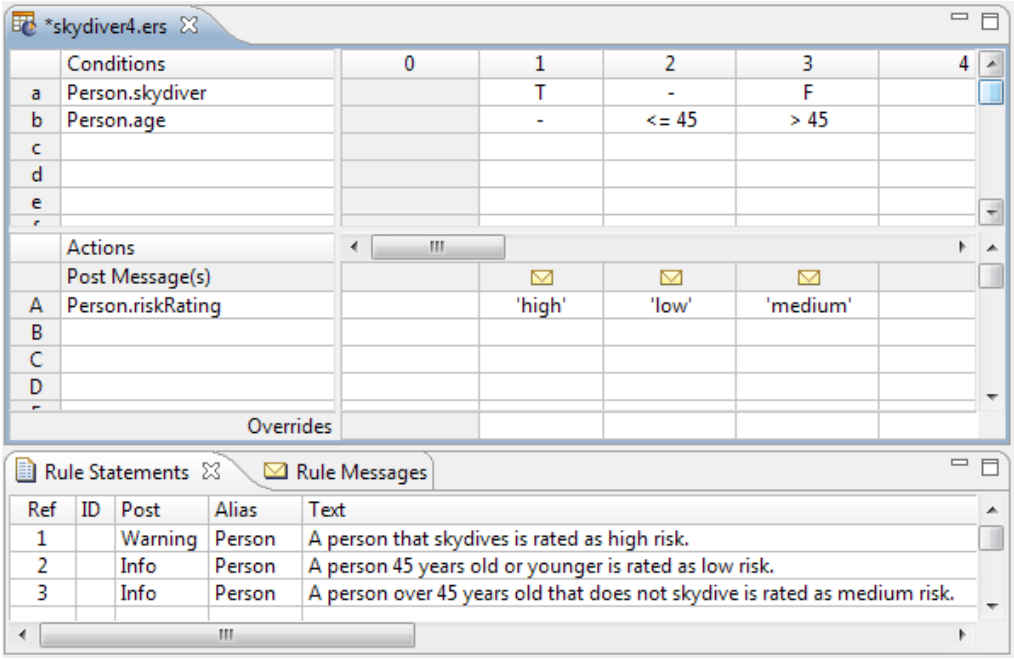
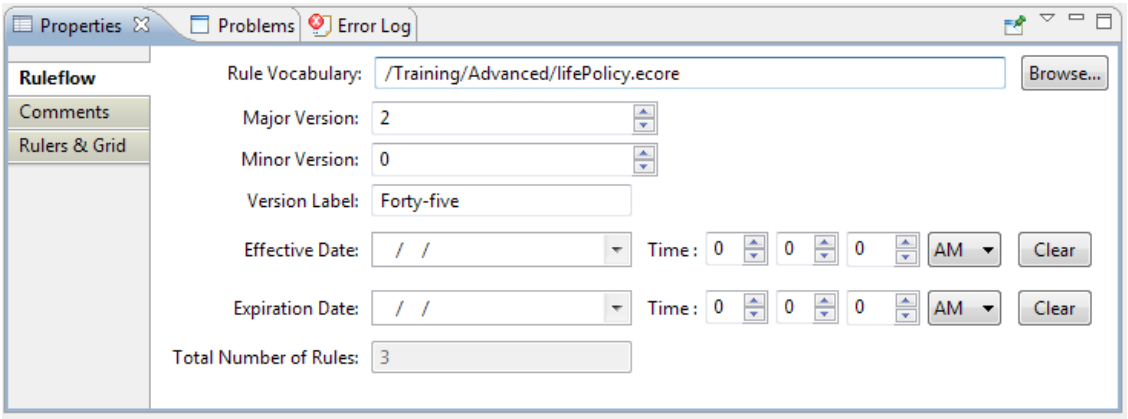


Figure 76: Ruleflow in folder Version2



And then for Version 3:

Figure 77: Rulesheet skydiver4.ers in folder Version3

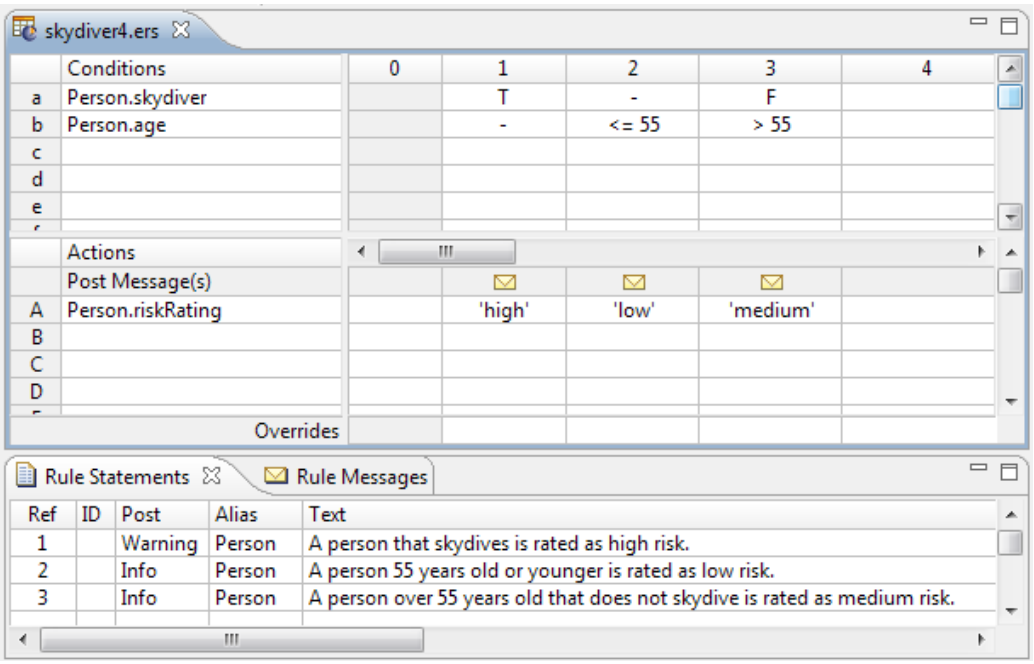
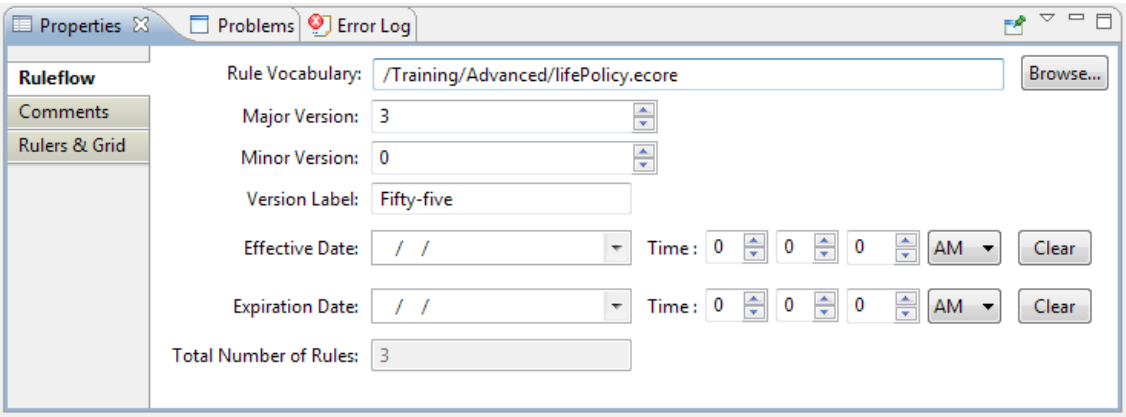


Figure 78: Ruleflow in folder Version3



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 examples on page 231 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 with no target version

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
```

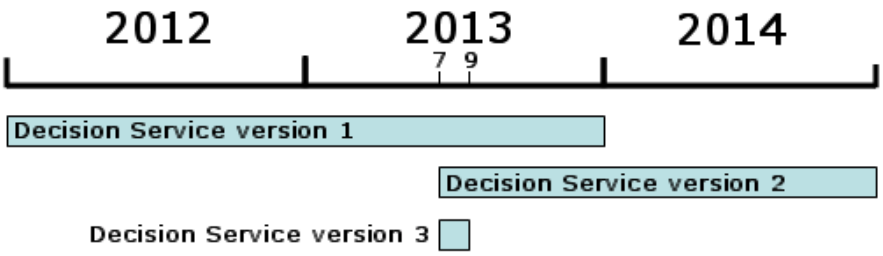
When no Version Number or EffectiveTimestamp is specified, the Server executes against v2.1 (if its Effective/Expiration range is valid). However, when Major Version 1 is passed in without an EffectiveTimestamp specified, the Server executes against v1.2 (if its Effective/Expiration range is valid).

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 79: DS Effective and Expiration Date Timeline



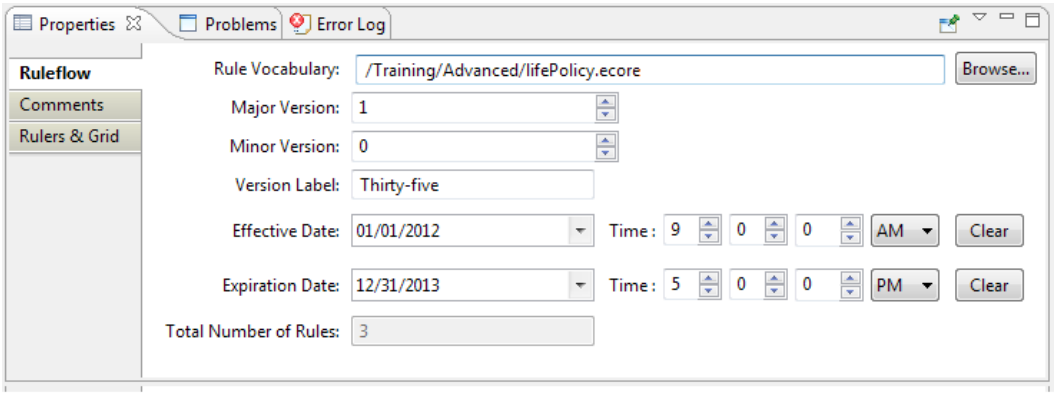
As illustrated, our three deployed Decision Services have Effective and Expiration dates that overlap in several date ranges: Version 1 and Version 2 overlap from July 1, 2013 through December 31, 2013. And Version 3 overlaps with both 1 and 2 in July-August 2013. To understand how Corticon Server resolves 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 80: Ruleflow in folder Version1 with dateTime set



We proceed to edit the other two Ruleflows as shown:

Figure 81: Ruleflow in folder Version2 with dateTime set

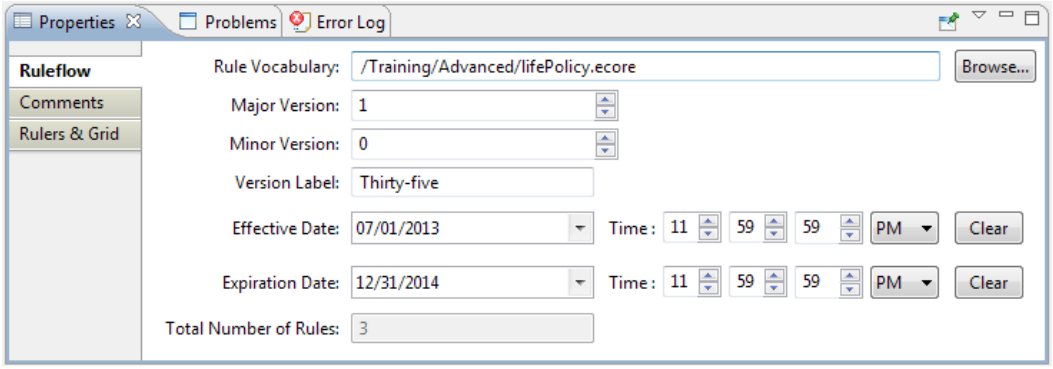
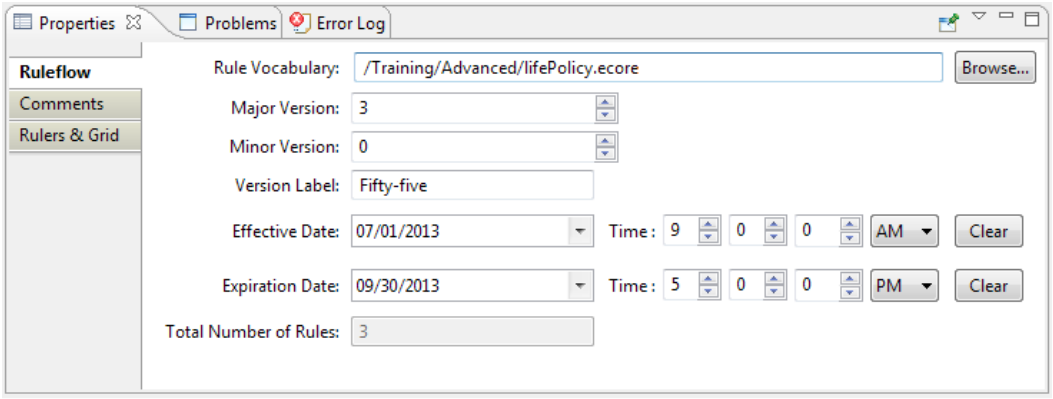


Figure 82: Ruleflow in folder Version3 with dateTime set



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 examples on page 231 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 with no timestamp

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">
```

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

Request Specifies Major Version?	Request Specifies Minor Version?	Request Specifies Timestamp?	Server Behavior
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
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

12

Using Corticon Server logs

Logging functions have to be able to cover the range between getting enough information to understand how to resolve general processing issues, and not getting so much information that the time and space for server operations is compromised. In development environments, more detailed settings can be helpful, while production systems need to capture significant events yet be able to tolerate short-term application of detailed logging.

Logging server activities is an important part of administering Corticon Server deployments. It is also a feature of the built-in Server in Corticon Studios. You can set logs to be as detailed or as brief as your needs for information and performance change.

For details, see the following topics:

- How users typically use logs
- Changing logging configuration
- Troubleshooting Corticon Server problems

How users typically use logs

How users typically use logs

Here are some ways you might use and manage logs.

- Configure logs to expose information:
  - Audit rule execution with log files (`logFiltersAccept` list includes `RULETRACE`)
  - Prevent sensitive data in log files (`logFiltersAccept` does not include`RULETRACE`)

  - Record diagnostic performance data (`logFiltersAccept` list includes `DIAGNOSTIC`), and then transform log data into CSV data for analysis tools

- Resolve problems:
  - Assess problems at server startup (Logs not created)
  - Assess problems with malformed requests
  - Assess problems with Corticon licensing
  - Assess problems with deployments
  - Assess problems with object translations

- Administer log files:
  - Produce Decision Service specific logs
  - Specify a preferred log path
  - Change retention policy for log file archives

Changing logging configuration

The default logging configuration enables basic logging without any need for further tailoring. This section describes the settings that let you adjust the configuration to suit your needs. The several properties that control logging features are described in the top section of the `brms.properties` file as these properties are often adjusted by users. These properties and how you can change them are described in two sections:

- **Log content** - The types and volume of information to be logged that are set in the `loglevel` and `logFiltersAccept` properties.
- **Log files** - The persistence techniques for log data that are set in the logpath, log rollover features, and option to log each Decision Service.

Configuring log content

What gets entered into logs is up to you. There are two dimensions to what produces log content. The `loglevel` records ascending levels of operational information from nothing to everything. The `logFiltersAccept` let you control whether each information type created by each of the process reporting mechanisms is accepted into the logs. The resulting logs meld entries from both dimensions sequentially, and record all the information into log files.

Log Level

The `loglevel` specifies the depth of detail in standard logging. When set to `OFF`, no log entries are produced. Each higher level enables log entries triggered for that level, as well as each lower level. Set your preferred log level by uncommenting the line `# loglevel=` in `brms.properties`, and then setting the value to exactly one of:

- `OFF` - Turn off all logging
- `ERROR` - Log only errors
- `WARN` - Log all errors and warnings
- `INFO` - Log all info, warnings and errors (Default value)
- `DEBUG` - Log all debug information and all messages applicable to `INFO` level
- `TRACE` - Equivalent to DEBUG plus some tracing logs
- `ALL` - Maximum detail

Note: The loglevel can be changed using the method `ICcServer.setLogLevel(String)`.

Log Filters

The `logFiltersAccept` setting lets you include specified types of information emanating from running services in the logs. When the log level is set to `INFO` or higher, this property acepts logging of information types that are listed. Set your preferred log filters by uncommenting the line `# logFiltersAccept=` in `brms.properties`, and then listing functions you want to have in logs as comma-separated values from the following:

- `RULETRACE` - Records performance statistics on rules
- `DIAGNOSTIC` - Records of service performance diagnostics at a defined interval (default is 30 seconds)
- `TIMING` - Records timing events
- `INVOCATION` - Records invocation events
- `VIOLATION` - Records exceptions
- `INTERNAL` - Records internal debug events
- `SYSTEM` - Records low-level errors and fatal events

The default `logFiltersAccept` setting is: `DIAGNOSTIC,SYSTEM`

Examples:

- Accept all: `logFiltersAccept=RULETRACE,DIAGNOSTIC,TIMING,INVOCATION,VIOLATION,INTERNAL,SYSTEM`
- Accept none: `logFiltersAccept=`
- Accept just ruletracing, diagnostics, and timing: `logFiltersAccept=RULETRACE,DIAGNOSTIC,TIMING`

Configuring log files

The files that record log entries can be relocated, created for each Decision Service, and archived for a specified number of cycles.

Log path

All log files are placed in a common location:

```
logpath=%CORTICON_WORK_DIR%/logs
```

The target folder can be changed to a preferred network-accessible location by uncommenting the line `# logpath=` in `brms.properties`, and then entering your location as the value. For example:

```
logpath=H:/CorticonLogs/Server
```

If the folder structure does not exist, it will be created. You must use forward slashes as the separator; if you do not, the level preceding a backslash and all lower levels will be ignored.

Note: The logpath can be changed using the method `ICcServer.setLogPath(String)`.

Log for each Decision Service

com.corticon.server.execution.logPerDS - This property enables the sifting of the logs into execution log files specific to each Decision Service. Default value is `false`.

The default logging approach is a single log for a running server. In server deployments, you can choose to maintain a log for each Decision Service. When Decision Service logging is enabled, log entries specific to a Decision Service are written only to that Decision Service's log file. Edit the `brms.properties` file to uncomment the following property and set it to `true`:

```
com.corticon.server.execution.logPerDS=true
```

When you save the file and restart the server, every transaction specific to a Decision Service is recorded in a file in the logs directory with the name pattern `Corticon-`*`DSname`*`.log`.

Archiving log histories

Logs 'rollover' on regular basis, compressing each current `.log` file at the log path -- the general log and every Decision Service log -- into a separate dated archive. The default action is to do this. If you decide that you do not want to rollover or archive logs, uncomment the line `# logDailyRollover` in `brms.properties`, and then set the value to `false`.

When log rollover is in effect, the `logRolloverMaxHistory` setting specifies the number of rollover logs to keep. (If the server is not always running or has low traffic, this might not be a number of days). The default is five archives. You can set your preferred number of archives by uncommenting the line `# logRolloverMaxHistory=` in `brms.properties`, and then entering your preferred positive integer value. For example:

```
logRolloverMaxHistory=21
```

Troubleshooting Corticon Server problems

When the Corticon Server has issues, the primary troubleshooting tool is the set of log files produced during Corticon Server operation, whether as Studio test server or as deployed Server.

By default, Corticon Server produces one log that records all the activities of running Decision Services at the specified log level. While higher detail levels produce a more comprehensive basis for analysis, the details of all running Decision Services will also generate detail information into the log. When diagnosing problems, you might want to use Corticon Server's ability to generate logs for each Decision Service so that you can produce detailed logs for each service.

The following procedure shows how to set the Server log to expose additional information, as well as to expand its data capture to detailed debugging mode.

Note: Consider backing up and then deleting all the log files and archives in the `\logs` folder so that you get only what is logged from your tests under the log settings. It is good to start with a fresh logs that record only the problematic transaction. The next time Corticon Server processes a transaction, a new log file will be created and entries recorded in it.

Setting logging content

See Configuring log content on page 170 for more information.

1. Edit the installation's `brms.properties` file.
2. Change the `loglevel` to a level that should bring in the operational activities that will reveal the problem, perhaps `DEBUG`.
3. Change `logFiltersAccept` to list activity elements that you think will be relevant.
4. Save the file.
5. Stop, then restart the server.

Setting logging for each Decision Service

See Log for each Decision Service on page 172 for more information.

1. Edit the `brms.properties` file.
2. Change the value of `com.corticon.server.execution.logPerDS` to `true`.
3. Save the file.
4. Stop, then restart the server.

Examining log files

1. Once you have restarted the server, rerun your tests.
2. In a text editor, navigate to `[CORTICON_WORK_DIR]\logs` to open the appropriate current logfile: Studio's is `CcStudio.log`, Server's is `CcServer.log`, and individual Decision Service (*DSname*) logs are `Corticon-`*`DSname`*`.log`.
3. Look for the indicators of problems that are described in the following sections.

Logs not created

Logging does not start until the server is invoked. Even though started, as viewed in its startup window, logs are not initialized until an activity occurs.

However, if you have routed a Corticon Request to the server and no log is produced, it is likely that the invocation/request is not even reaching the server. The most common causes of a non-responsive (invocation produces no log file entry) Corticon Server include:

- Incorrect Corticon Server deployment. Review your deployment procedures to confirm the deployment files and paths.
- Incorrect Corticon Server invocation
  - **Incorrect URL** - If using a web services deployment, ensure the SOAP message is addressed correctly, and that no firewalls or port configurations prevent the SOAP message from reaching Corticon Server.
  - **Incorrect API** - Review the Administrative APIs on page 88 for details on Java APIs available for Corticon Server invocation.

Note: See the complete Java Server JavaDocs in Studio and Java Server installations at `[CORTICON_HOME]\JavaDoc`.

- Even though Corticon Server may not respond to an incorrect invocation, the host server or container (app server, web server, and similar) may respond either at a command line or log level. Check to see if the host server has responded to your invocation.

Response containing errors

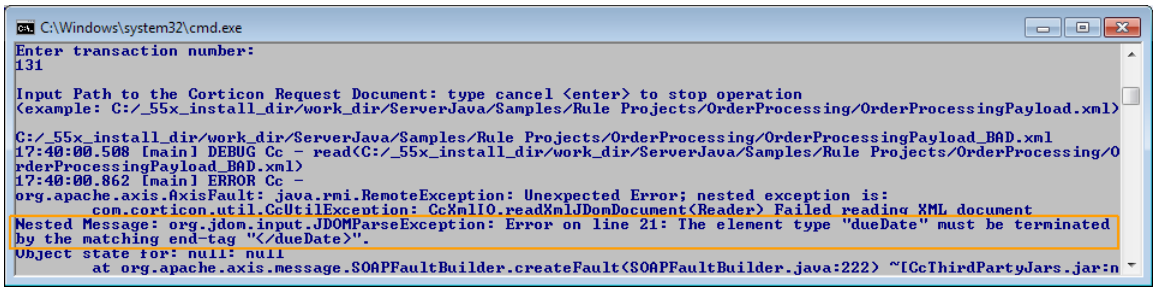
The most common causes of erroneous Corticon Server responses include:

Problems with a request

Invalid syntax and misnamed targets are common problems with requests:

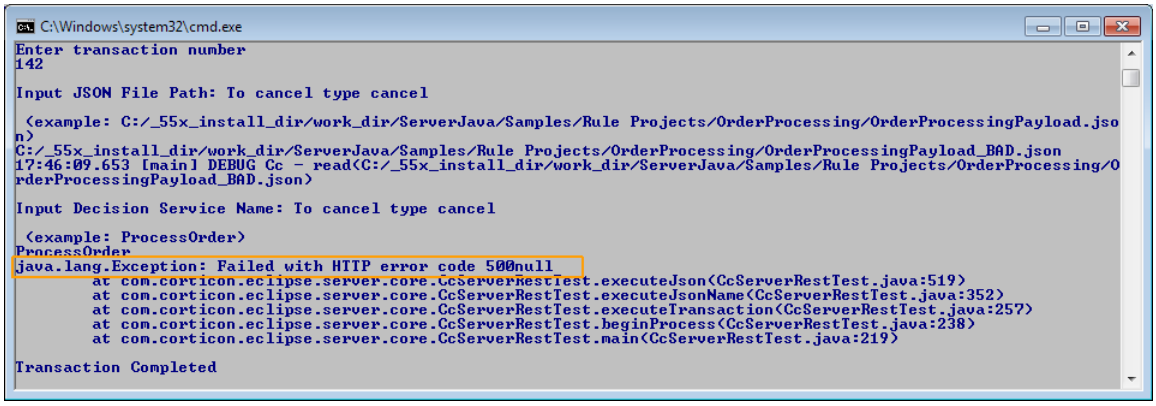
Message payload does not conform to service contract - Compare your SOAP message to the service contract produced by the Deployment Console to ensure compliance. Many third-party tools are available that automatically validate an XML document (in this case, the SOAP message) to its schema (in this case, the WSDL service contract). Notice that if Corticon Server cannot even parse the inbound SOAP message, no entry will be made in Corticon Server's log. Instead, the error message will be displayed directly in the web server window, as shown:

Figure 83: Server Window Message Highlighting Incorrect SOAP/XML Request Structure



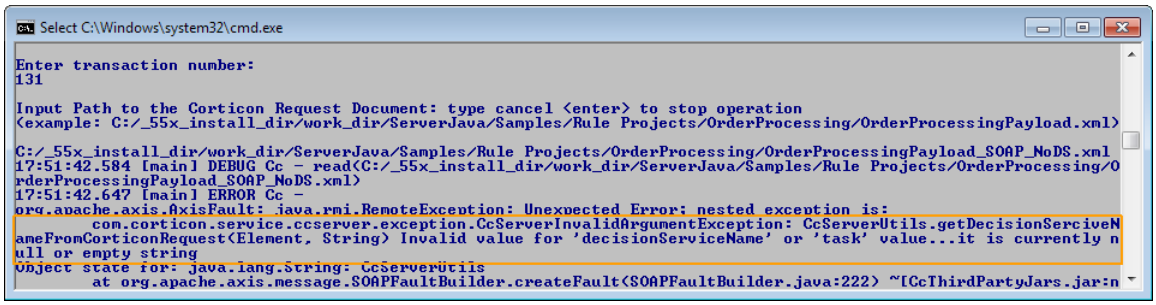
Poorly formed JSON request - Syntax errors in a JSON message generate an error message in the web server window, as shown:

Figure 84: Server Window Message Highlighting Incorrect JSON Request Structure



Incorrect or missing Decision Service Name - Ensure the SOAP/XML message's Decision Service Name attribute matches the name of the Decision Service as it was deployed by either a deployment descriptor file or an API method call, as shown:

Figure 85: Log Excerpt Highlighting Missing Decision Service Name in SOAP/XML Request



Review Decision Service metadata in different environments

Corticon log files add metadata on each Decision Service as it is loaded so that you can confirm consistent loading of a Decision Service instance in different environments. For example, the log header will have lines similar to the following::

```
2015-08-20 14:39:52.784 INFO DIAGNOSTIC [localhost-startStop-1] Cc
com.corticon.eclipse.server.core.impl.CcServerPool -
    ADD DECISION SERVICE :::
DecisionServiceName=Cargo,Version=1.0,CompiledVersionNumber=5.5.1,
    CompiledBuildNumber=7300,EDSTimestamp=08/03/15 4:04:03 PM,

RuleCount=2,MaxPoolSize=1,AutoReload=true,CddPath=C:/_55x_install_dir/work_dir/Server/cdd/Cargo.cdd,

    DatabaseAccessMode=null,ReturnEntities=ALL,ContainsServiceCallouts=false
...
2015-08-20 14:39:52.815 INFO DIAGNOSTIC [localhost-startStop-1] Cc
com.corticon.eclipse.server.core.impl.CcServerPool -
  ADD DECISION SERVICE :::
DecisionServiceName=ProcessOrder,Version=1.10,CompiledVersionNumber=5.5.1,
  CompiledBuildNumber=7300,EDSTimestamp=08/03/15 4:04:07 PM,

RuleCount=6,MaxPoolSize=1,AutoReload=true,CddPath=C:/_55x_install_dir/work_dir/Server/cdd/OrderProcessing.cdd,

  DatabaseAccessMode=null,ReturnEntities=ALL,ContainsServiceCallouts=false
```

The metadata is recorded in the log for each Decision Service at every server startup, at every log rollover, and whenever a Decision Service is added, updated, or deleted.

Corticon Server license problem

The basic license issues are as follows:

License not installed - The `CcLicense.jar` license file must be located in the same directory as your server installation's `CcServer.jar` file. In the default installation, `CcServer.jar` is located in `[CORTICON_WORK_DIR]\Server\pas\server\webapps\axis\WEB-INF\lib`, so ensure your valid license file is there.

See the topic *"Updating your Corticon Server license" in the Corticon Installation Guide* for more information.

Note: If you are using one of the `.war` or `.ear` packages, then be sure that those packages also include valid copies of `CcLicense.jar`.

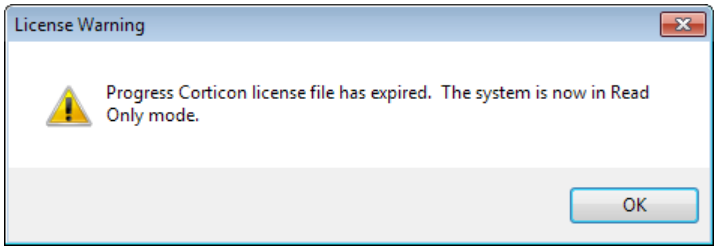
License expired - If your license indicates that it has expired, contact your Progress Corticon representative to obtain an updated license file.

Corticon Server logs this information as:

```
This Product is licensed to: {name} - {use}
Progress Corticon Server license has expired.
Please contact Customer Support to receive a valid Key.
```

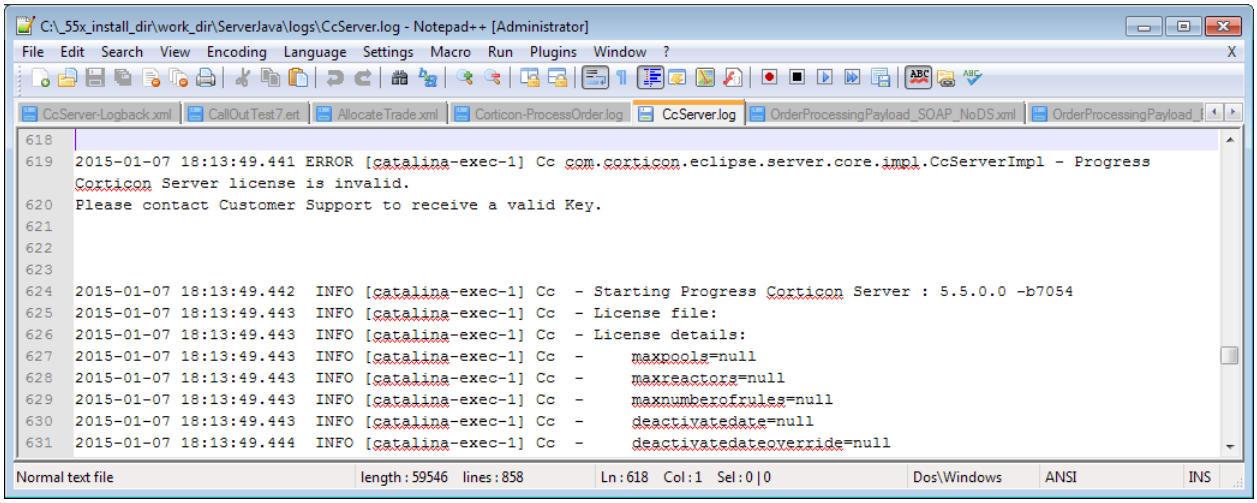
Corticon Studio alerts the user at startup, and then limits functionality:

Figure 86: License Expiration alert at Studio startup



License Invalid - If your license indicates that it is invalid, contact your Progress Corticon representative to obtain a valid license file.

Figure 87: Log Excerpt Highlighting License Invalid Message



License capacity exceeded - License capacity is measured in several ways:

- Number of unique Decision Services that may run concurrently in Corticon Server. Make sure your license can support the total number of unique `.erf` files referenced by deployed `.cdd` files.
- Number of rules allowed for all Decision Services deployed. Make sure your license can support the total number of rules contained in all the deployed `.erf` files.

Deployment Descriptor (`.cdd`) file problems

The following items are common Deployment Descriptor file problems:

Missing Ruleflow (`.erf`)file - The `.erf` file has been moved and is no longer located in the directory referenced by the `.cdd` file. For example, the log might record:

```
Failed -> Cdd C:\Users\Admin\Progress\CorticonWork5.5\OrderProcessing.cdd
Could not find a valid Ruleflow located at
C:/Users/Admin/Progress/Samples/Rule Projects/OrderProcessing/Order.erf
```

Missing Deployment Descriptor (`.cdd`) file - The `.cdd` file is missing from the `\cdd` directory, or the taskname contained in the SOAP request message does not match any of the tasknames in any of the `.cdd` files deployed to Corticon Server. For example, the log might record:

```
...
|ERROR|com.corticon.service.ccserver.exception.CcServerDecisionService.notRegisteredException:
CcServerDecisionService.lookupCcServerPoolForExecution () Decision Service:
OrderProcess is not registered. Update failed. (Missing pool manager)
```

Missing `\cdd` directory - The default location of the `cdd` directory in a server installation is `[CORTICON_WORK_DIR]\cdd`. For example, the log might record:

```
java.rmi.RemoteException: Unexpected Error; nested exception is:
com.corticon.service.ccserver.exception.CcServerInvalidArgumnentException:
CcServer.loadDromCddDir (String)   Directory does not exist.
```

Object translation errors due to incorrect Vocabulary external name mappings

- External names mapped incorrectly
- External data types specified incorrectly
- ALL entities must be mapped, even those where all attributes are transient.

13

Performance and tuning guide

This section discusses aspects of Corticon Server performance.

For details, see the following topics:

- Rulesheet performance and tuning
- Server performance and tuning
- Optimizing pool settings for performance
- Single machine configuration
- Cluster configuration
- Capacity planning
- The Java clock
- Diagnosing runtime performance of server and Decision Services

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 Rulesheets, 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 Rulesheets 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.

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 and allocation is enabled, 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 maximum 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 machine 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 maximum pool size settings for each deployed Decision Service that has enabled allocation. 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.

Note: The `CcServer.ear` and `CcServer.war` files are available from the Progress download site in the `PROGRESS_CORTICON_5.6_SERVER.zip` package. The unpackaged files are typically installed in the Corticon directory `[CORTICON_HOME]\Server\Containers\{EAR/WAR}`. Refer to the Progress Software web page Progress Corticon 5.6 - Supported Platforms Matrix to review the currently supported UNIX/Linux platforms and brands of Application Servers. Also see the Corticon KnowledgeBase entry Corticon Server 5.X sample EAR/WAR installation for different Application Servers for detailed instructions on configuring Apache Tomcat, JBoss, WebSphere, WebLogic on all supported platforms.

The sample EJB code in Corticon's default installation sets the number of wrappers in the `weblogic-ejb-jar.xml file` (located in `meta-inf` of the EAR location's `\lib\CcServerAdminEJB.jar` and `\lib\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 number of available CPUs on the server. In version 5.5, the Corticon Server implemented its own Execution Queue to control how many threads can simultaneously execute inside the Corticon Server. By default, the Execution Queue will be configured to match the number of available CPUs on the machine, but this can be overridden by changing the following Corticon property in the `brms.properties` file, as follows:

```
##########################################################################################
#  This property will be used by the CcServer to determine how many concurrent executions
 can occur across
#  the CcServer.  Ideally, this value will be set to the number Cores on the machine.
By default, this value
#  is set to 0, which means the CcServer will auto-detect the number of Cores on the
server.
#
#  Default value is 0 (CcServer will auto-detect the number of Cores on machine and use
 that number)
##########################################################################################
```

```
com.corticon.server.execution.processors.available=0
```

Maximum pool sizes

This value in the Decision Service is utilized when the CcServer is configured with Decision Service Allocation turned on. The following property needs to be set to true (default is `false`):

```
com.corticon.server.decisionservice.allocation.enabled=true
```

With the allocation property set to true, the CcServer can control how many incoming execution threads from each Decision Service will be added to the CcServer's Execution Queue. If additional execution threads come into for a particular Decision Service, and that Decision Service has already allocated its maximum to the Execution Queue, then the incoming execution thread waits until an execution thread for that Decision Service leaves the Execution Queue.

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.

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.

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.

Diagnosing runtime performance of server and Decision Services

When performance issues arise, analyzing usage characteristics might reveal the performance bottlenecks. Corticon servers always start a diagnostic thread, and a snapshot of key diagnostic metrics is taken at a regular interval. The diagnostic data is forwarded to the logging system where they are recorded. You can then run a utility that extracts diagnostic lines, and transforms them to a standard comma-separated value format you can use in a data analysis product such as Tableau or Excel to create visualizations.

Properties that control diagnostic logging

The user can control the starting of the diagnostic thread, the intervals at which diagnostic snapshots are taken, and whether diagnostic lines are accepted into logs. Add or edit the following properties in the `brms.properties`:

- To enable (`true`) or disable (`false`) automatic startup and configuration of the server monitor thread when an `ICcServer` is created in the `CcServerFactory`. Default value is `true`.

  ```
  com.corticon.server.startDiagnosticThread=true
  ```

- To specify the wait time in milliseconds of the Server Diagnostic Monitor. Default is `30000` - 30 seconds.

  ```
  com.corticon.server.DiagnosticWaitTime=30000
  ```

- To set the log level and/or filters to accept diagnostic entries. The default loglevel, `INFO`, and higher loglevels enable filters that are set to accept `DIAGNOSTIC` entries. Choosing a lower loglevel and/or removing `DIAGNOSTIC` from the `logFiltersAccept` list denies generated diagnostic data entry into the log.

Note: See the topic Changing logging configuration on page 170 for more information.

Example of diagnostic log entries for a server and four Decision Services

```
2015-05-13 14:02:34.831 INFO DIAGNOSTIC [CcDiagnosticsThread] Cc  -
    id=1431540154831,sthp=509.6875,shp=356.08567810058594,sex=0,stq=1564,
    sec=1564,sfc=0,saex=3.9801790281329925,sawt=0
2015-05-13 14:02:34.831 INFO DIAGNOSTIC [CcDiagnosticsThread] Cc  -
    id=1431540154831,ds=ProcessOrder.1.10,ec=1564,aex=3,awt=0,fc=0
2015-05-13 14:02:34.831 INFO DIAGNOSTIC [CcDiagnosticsThread] Cc  -
    id=1431540154831,ds=Candidates.1.14,ec=0,aex=0,awt=0,fc=0
2015-05-13 14:02:34.831 INFO DIAGNOSTIC [CcDiagnosticsThread] Cc  -
    id=1431540154831,ds=AllocateTrade.1.14,ec=0,aex=0,awt=0,fc=0
2015-05-13 14:02:34.831 INFO DIAGNOSTIC [CcDiagnosticsThread] Cc  -
    id=1431540154831,ds=Cargo.1.0,ec=0,aex=0,awt=0,fc=0
2015-05-13 14:03:04.842 INFO DIAGNOSTIC [CcDiagnosticsThread] Cc  -
    id=1431540184842,sthp=509.6875,shp=371.9812469482422,sex=0,stq=1564,
    sec=1564,sfc=0,saex=3.9801790281329925,sawt=0
```

To generate DIAGNOSTIC data into the server log:

The default Corticon Server settings generate diagnostic data and capture the diagnostic entries in the log.

1. In `brms.properties`, set the monitor interval so that unusual spikes of activity in a key metric (such as heap size) are captured. If the window is large, the heap size might go from normal to the server crashing with OutOfMemory exceptions without leaving any trail in the diagnostics entries in the log. The default value is 30000 milliseconds. You could change it to, say, 10 seconds by locating the line: `#com.corticon.server.DiagnosticWaitTime=30000`and then changing its value to `com.corticon.server.DiagnosticWaitTime=10000`
2. Confirm that the `logLevel` is `INFO` or higher and that `logFiltersAccept` includes `DIAGNOSTICS`.
3. Save the file.
4. Start Corticon Server.
5. Execute some requests or activities through the server.
6. Examine the server log at `[CORTICON_WORK_DIR]\logs\` to note lines that contain "`DIAGNOSTIC`".

You can now run the utility that extracts only the diagnostic lines and transforms each from *name=value* pairs to comma-separated integer and string values. The Server and Decision Service Diagnostic data and procedures are discussed separately.

SERVER DIAGNOSTICS

The server diagnostic log entries provide general server health metrics. The following metrics are logged:

Table 11: Content of a Server diagnostic entry

Item	Description
Diagnostic set ID (`id`)	Grouping of log lines from a common time slice
Date Time	Timestamp of log entry
Server Total Heap size (`sthp`)	Corticon server JVM total memory in MB
Server Heap size (`shp`)	Corticon server JVM allocated memory in MB
Server Executing threads (`sex`)	Current count of threads actively executing Decision Services
Server Threads in Queue (`stq`)	Count of Decision Service invocations waiting in the queue to be executed
Server Execution Count (`sec`)	Total number of Decision Services executed since Corticon server startup
Server Failure Count (`sfc`)	Total execution failure count since Corticon server startup
Server Average Executions time (`saex`)	Average Decision Service execution time measured across all Decision Services,and provides the average since the Corticon server startup
Server Average Wait Time (`sawt`)	Average Decision Service wait time measured across all Decision Services, and provides the average since the Corticon server startup.

Once you have a log file with diagnostic entries, running a Corticon management utility takes an input file and produces each `DIAGNOSTIC` line as CSV data in its output file. To run the utility against a log that has been archived, extract the log file to a temporary location.

To extract server diagnostic data from a Corticon Server log:

1. Open a command prompt window at `[CORTICON_HOME]\Server\bin`.
2. Enter:

```
corticonManagement.bat -e -s -i {input_file}  -o {output_file}
```

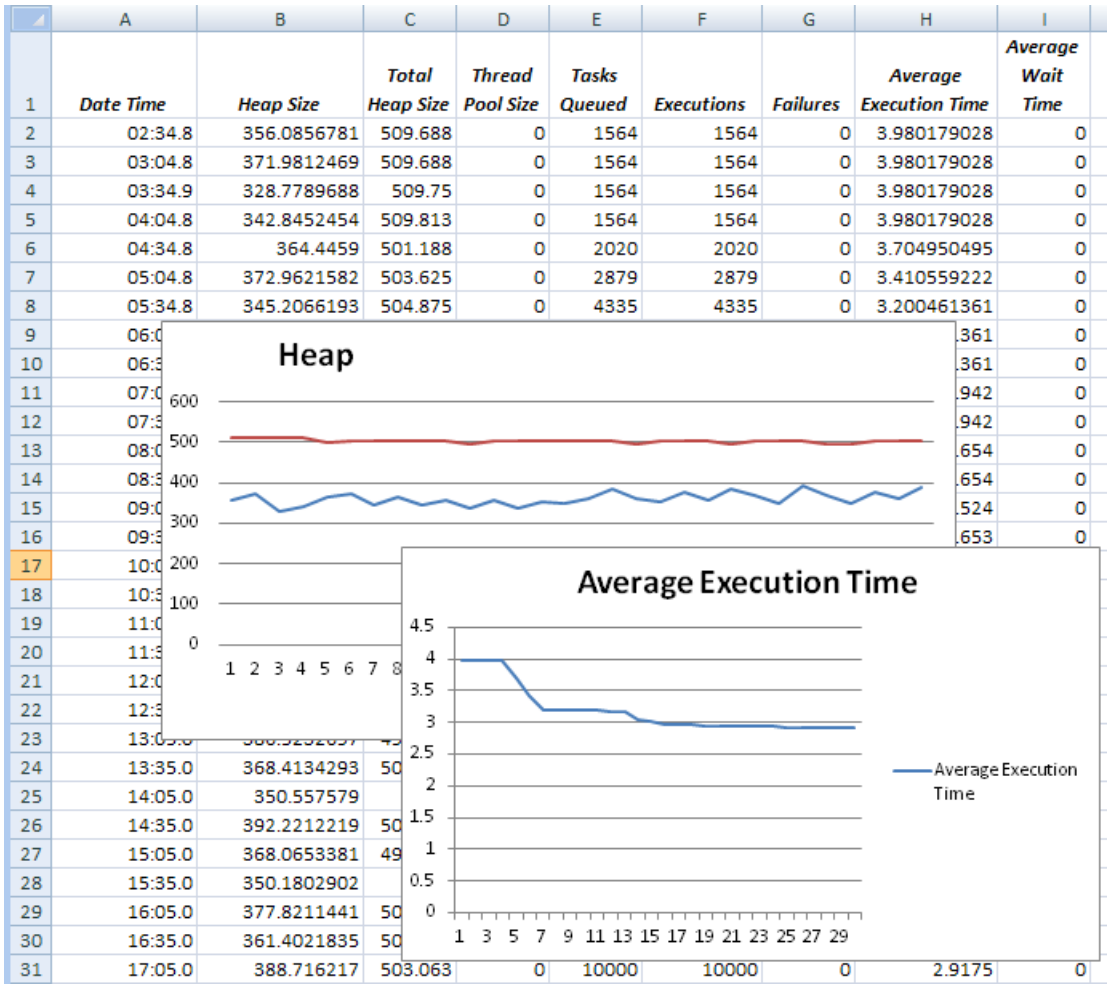
For example:

```
corticonManagement.bat -e -s -i C:\CcServer.log  -o C:\CcServer_20150513.csv
```

When the processing completes, the input file is unchanged. The output file extracts only Server diagnostic lines, transforming each line into CSV values and a header line as shown for the log example above:

```
Date Time,Heap Size,Total Heap Size,Thread Pool Size,Tasks Queued,
    Executions,Failures,Average Execution Time,Average Wait Time
2015-05-13 14:03:04.842,371.9812469482422,509.6875,0,1564,1564,0,3.9801790281329925,0
2015-05-13 14:03:34.854,328.77896881103516,509.75,0,1564,1564,0,3.9801790281329925,0
2015-05-13 14:04:04.827,342.8452453613281,509.8125,0,1564,1564,0,3.9801790281329925,0
2015-05-13 14:04:34.829,364.4458999633789,501.1875,0,2020,2020,0,3.704950495049505,0
2015-05-13 14:05:04.831,372.962158203125,503.625,0,2879,2879,0,3.4105592219520666,0
2015-05-13 14:05:34.833,345.2066192626953,504.875,0,4335,4335,0,3.200461361014994,0
2015-05-13 14:06:04.835,366.8616409301758,503.4375,0,4335,4335,0,3.200461361014994,0
2015-05-13 14:06:34.837,345.76478576660156,502.9375,0,4335,4335,0,3.200461361014994,0
2015-05-13 14:07:04.839,358.3552551269531,503.5,0,4448,4448,0,3.1913219424460433,0
2015-05-13 14:07:34.841,337.6990051269531,495.75,0,4448,4448,0,3.1913219424460433,0
```

The Server Diagnostic CSV data is compatible with analytic and visualization products such as Excel and Tableau, as illustrated:



	A	B	C	D	E	F	G	H	I
1	Date Time	Heap Size	Total Heap Size	Thread Pool Size	Tasks Queued	Executions	Failures	Average Execution Time	Average Wait Time
2	02:34.8	356.0856781	509.688	0	1564	1564	0	3.980179028	0
3	03:04.8	371.9812469	509.688	0	1564	1564	0	3.980179028	0
4	03:34.9	328.7789688	509.75	0	1564	1564	0	3.980179028	0
5	04:04.8	342.8452454	509.813	0	1564	1564	0	3.980179028	0
6	04:34.8	364.4459	501.188	0	2020	2020	0	3.704950495	0
7	05:04.8	372.9621582	503.625	0	2879	2879	0	3.410559222	0
8	05:34.8	345.2066193	504.875	0	4335	4335	0	3.200461361	0
9	06:0							.361	0
10	06:3							.361	0
11	07:0							.942	0
12	07:3							.942	0
13	08:0							.654	0
14	08:3							.654	0
15	09:0							.524	0
16	09:3							.653	0
17	10:0								
18	10:3								
19	11:0								
20	11:3								
21	12:0								
22	12:3								
23	13:0								
24	13:35.0	368.4134293	50						
25	14:05.0	350.557579							
26	14:35.0	392.2212219	50						
27	15:05.0	368.0653381	49						
28	15:35.0	350.1802902							
29	16:05.0	377.8211441	50						
30	16:35.0	361.4021835	50						
31	17:05.0	388.716217	503.063	0	10000	10000	0	2.9175	0

DECISION SERVICE DIAGNOSTICS

A diagnostic entry is created for each Decision Service that is deployed to the Corticon server. If you are creating separate logs for each Decision Service, the utility runs against its corresponding log; otherwise, the Decision Service diagnostic is captured into the server log where the utility will, in turn, extract the data for one specified Decision service at a time against the same server log. The following metrics are logged for each Decision Service.

Table 12: Content of a Decision Service diagnostic entry

Item	Description
Diagnostic set ID (`id`)	Grouping of log lines from a common time slice
Date Time	Timestamp of log entry
Decision Service name (`ds`)	The name and Major.minor version of the deployed Decision Service
Execution Count (`ec`)	The total execution count for this Decision Service since the Corticon server startup
Average Execution time (`aex`)	Average execution time in the designated Decision Service since the Corticon server startup
Average Wait Thread time (`awt`)	The Average execution wait time for threads entering the designated Decision Service since the Corticon server startup
Failure Count (`fc`)	Number of failures recorded in the designated Decision Service since the Corticon server startup

Once you have a log file with diagnostic entries, running a Corticon management utility takes an input file and produces each `DIAGNOSTIC` line as CSV data in its output file.

To extract Decision Service diagnostic data from a Corticon Server log:

1. Open a command prompt window at `[CORTICON_HOME]\Server\bin`.
2. Enter:

   ```
   corticonManagement.bat -e -ds {DSname} -dsv {major.minor} -i {input_file}  -o
   {output_file}
   ```

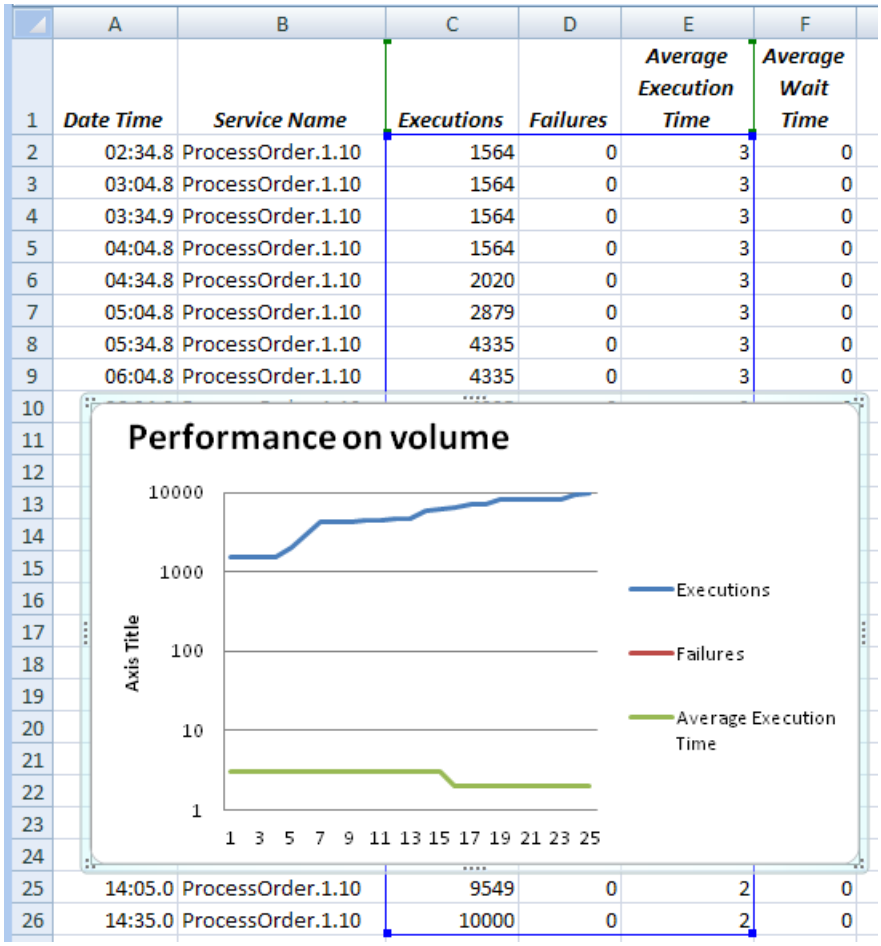
   For example:

   ```
   corticonManagement.bat -e -ds ProcessOrder -dsv 1.10
           -i C:\CcServer.log  -o C:\ProcessOrder_1.10_20150513.csv
   ```

When the processing completes, the input file is unchanged. The output file extracts only diagnostic lines, transforming each line into CSV values and a header line as shown for the log example above:

```
Date Time,Service Name,Executions,Failures,
  Average Execution Time,Average Wait Time
2015-05-13 14:02:34.831,ProcessOrder.1.10,1564,0,3,0
2015-05-13 14:03:04.842,ProcessOrder.1.10,1564,0,3,0
2015-05-13 14:03:34.854,ProcessOrder.1.10,1564,0,3,0
2015-05-13 14:04:04.827,ProcessOrder.1.10,1564,0,3,0
2015-05-13 14:04:34.829,ProcessOrder.1.10,2020,0,3,0
2015-05-13 14:05:04.831,ProcessOrder.1.10,2879,0,3,0
2015-05-13 14:05:34.833,ProcessOrder.1.10,4335,0,3,0
2015-05-13 14:06:04.835,ProcessOrder.1.10,4335,0,3,0
2015-05-13 14:06:34.837,ProcessOrder.1.10,4335,0,3,0
2015-05-13 14:07:04.839,ProcessOrder.1.10,4448,0,3,0
```

The Decision Service Diagnostic CSV data is compatible with analytic and visualization products, as illustrated in Excel:



	A	B	C	D	E	F
1	*Date Time*	*Service Name*	*Executions*	*Failures*	*Average Execution Time*	*Average Wait Time*
2	02:34.8	ProcessOrder.1.10	1564	0	3	0
3	03:04.8	ProcessOrder.1.10	1564	0	3	0
4	03:34.9	ProcessOrder.1.10	1564	0	3	0
5	04:04.8	ProcessOrder.1.10	1564	0	3	0
6	04:34.8	ProcessOrder.1.10	2020	0	3	0
7	05:04.8	ProcessOrder.1.10	2879	0	3	0
8	05:34.8	ProcessOrder.1.10	4335	0	3	0
9	06:04.8	ProcessOrder.1.10	4335	0	3	0
25	14:05.0	ProcessOrder.1.10	9549	0	2	0
26	14:35.0	ProcessOrder.1.10	10000	0	2	0

Interpreting diagnostic data

Here is a guide to what changes in performance diagnostic values might indicate:

- When the number of waiting threads (`wt`) goes up, it is an indication that the request demand is greater than the server capacity. Some wait time may be necessary in periods of high demand.
- The average wait time (`awt`) can be used to determine whether server capacity should be expanded (or if consistently low, might indicate contracting server resources).

- The number of executions (`ex`) combined with the average wait time help pinpoint whether there is a need to expand server resources or just to accept a slower response in small, high demand windows.
- The number of failures (`fl`) is an indication that expert analysis/maintenance is needed.
- The average execution time (`aex`) can be used to determine if there are configuration/resource issues. If this rate is not stable, it might indicate that the resource configuration is not optimal. However, this value can be dependent upon data size -- if the input data size is not stable the execution size will not be stable.

14

Enabling Server handling of locales, languages, and timezones

When deploying Decision Services that will be consumed by users or services running in different locales, you often need to address issues with locale-dependent data formats and localized messages. Corticon now has the ability to specify a "locale" when calling a Decision Service. When locale is specified, Corticon uses that locale when parsing and formatting locale-dependent data types such as Decimals and Dates. In addition, Corticon returns localized Rule Messages if you defined localizations for the messages when creating the Rulesheets for your Decision Service.

In prior releases, you would have needed to deploy a Decision Service multiple times--once for each locale supported--to have localized Rule Messages returned. This is no longer necessary. A single deployed Decision Service can support multiple locales.

Localizing your rule modeling and processing environment can implement five related functions:

1. Displaying the Studio **program** in your locale of choice. This means switching the Corticon Studio user interface (menus, operators, system messages, etc.) to a new language. Consult with Progress Corticon support or your Progress representative to learn more about available and upcoming Corticon localization packages.
2. Displaying your Studio **assets** in your locale of choice. This means switching your Vocabularies, Rulesheets, Ruleflows, and Ruletests to a new language. See *"Localizing Corticon Studio" in the Rule Modeling Guide*.
3. Displaying your localized Rulesheet's **rule statements** in your locale of choice. Rulesheets can specify rule statements in another language that are returned to requestors when the server is set to that language. This is not a new feature in this release. However, as of this release, when a request's execution property specifies a language that has defined appropriate rule statements, the locale-specific statements are included in the response. See *"Localizing Corticon Studio" in the Rule Modeling Guide*.
4. Enabling requests submitted to a Corticon Server to set an execution property that indicates the locale of the incoming payload so that the server can transform the payload's **locale-specific decimal and date**

literal values to the decimal delimiter and month literal names of the server, run the rules, and return the output formatted for the submitter's specified locale. This function is described in this section.

5. Enabling requests submitted to a Corticon Server to set an execution property that indicates the **timezone** of the incoming payload so that the server can transform the payload's time calculations to the timezone of the server, run the rules, and return the output formatted for the submitter's specified timezone. This function is described in this section.

For details, see the following topics:

- Character sets supported
- Handling requests and replies across locales
- Examples of cross-locale processing
- Example of cross-locale literal dates
- Example of requests that cross timezones

Character sets supported

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
```

Handling requests and replies across locales

When a Corticon service request document provides data formats that are unsupported by the Server, the request throws an exception. The two most common issues are:

- Inconsistent parsing of the decimal delimiter - For example, a message is supplying a comma (such as "157,1") and the Server is expecting a period ("157.1")
- Inconsistent name of a literal month name - For example, a message is supplying a French name (such as "avril") and the Server is expecting an English name ("April")

An inbound message can provide the locale of the message payload in the form:

```
<ExecutionProperties>>
  <ExecutionProperty name="PROPERTY_EXECUTION_LOCALE" value="language-country" />
</ExecutionProperties>>
```

where *language-country* is the JVM standard identifier, such as `en-US` for **English-United States**.

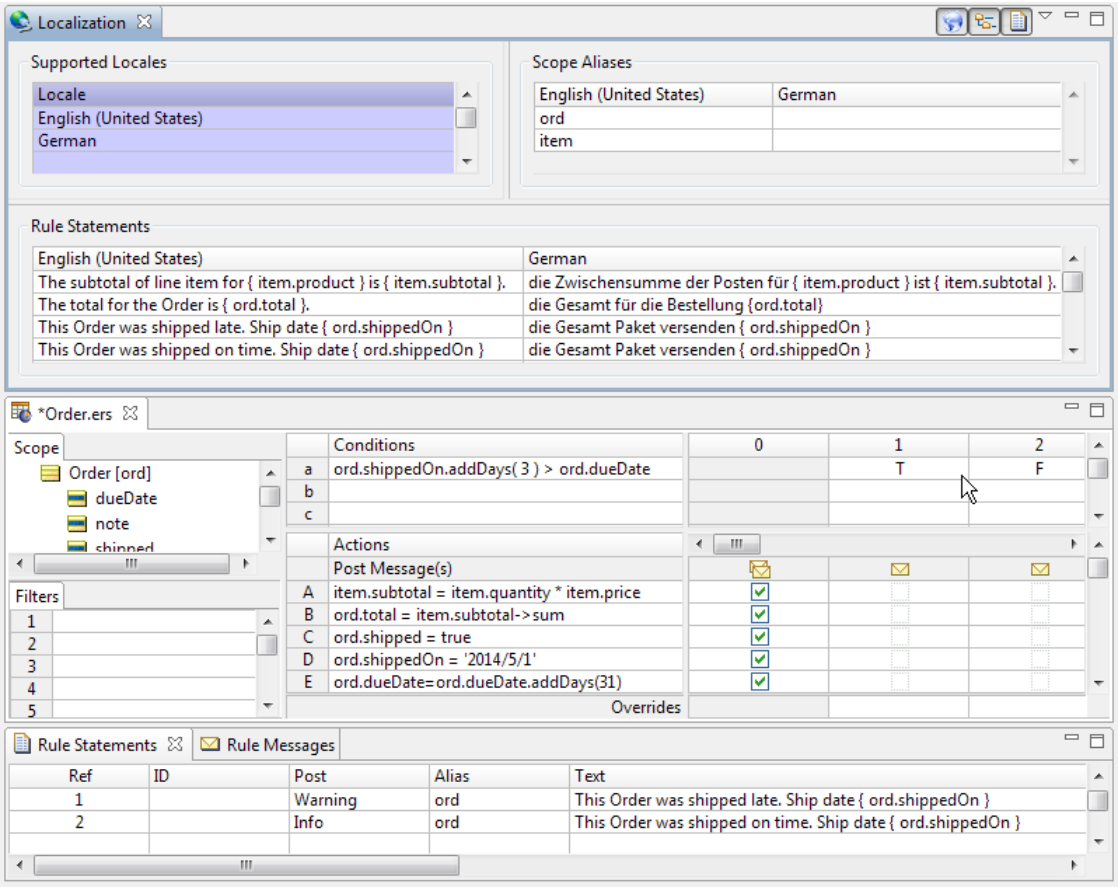
When the message's locale is specified, it is used at rule execution time regardless of the Server's default locale. If the Rulesheet has a matching locale, those rule statement messages are used. Whether or not there is a match, the JVMs functionality enables it to map the input request's decimal delimiters and the literal month names to the server locale's corresponding format. When rule processing is complete, the output response maps the results to the formats of the requestor's locale, and--when rule statement messages are available for the requestor's locale--messages for that locale are included.

Note: Matching a literal month name must have the appropriate case and diacritical marks, such as août, décembre and März.

Note: When this property is not set on an inbound request, the Corticon Server assumes the locale of the server machine, or the language that is set as an override in the Java startup of the server. That setting will use locale settings in Corticon Rulesheets for rulestatement messages so that a server running the Rulesheet's Decision Service would get rule statements that are specified for that locale.

Examples of cross-locale processing

The following examples use the installed Progress Application Server and the API test scripts. It also presents a sample of the OrderProcessing sample Rulesheet enhanced to show localization to German rule statements and some test conditions and actions that expose the features of cross-locale processing.



The internationalization feature uses the English rule statements in replies to requests. When the Server is set to German, it uses the German rule statements in replies to requests.

When a request does not indicate its language and locale, and the request has decimal values or literal dates that are not consistent with the server's format, the request message throws an exception.

```
<ns1:Messages version="1.10">
  <ns1:Message>
    <ns1:severity>Violation</ns1:severity>
    <ns1:text>An unexpected error occurred in Input Data:
                java.lang.NumberFormatException</ns1:text>
  </ns1:Message>
</ns1:Messages>
```

Note: If the request has no decimal values or literal dates, the response contains rule statements in the server's locale.

When a request includes the execution property `PROPERTY_EXECUTION_LOCALE` and a valid value, the provided locale is used to parse data values in the request document and to produce the response document. In the response document, the provided locale is used to format data values and to select the localized rule messages to return. Data types with locale dependencies are decimal and literal dates. If an invalid locale is provided, an exception is thrown. If localized rule messages were not defined, the default rule messages are used.

Using the example of the English-German rulesheet, and assuming that the Decision Service is running on a `en-US` system, consider the following messages:

The following request specifies German, `de-DE`, as its locale:

```
<CorticonRequest xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 decisionServiceName="Order_localeAware">
  <ExecutionProperties>
 <ExecutionProperty name="PROPERTY_EXECUTION_LOCALE" value="de-DE" />
  </ExecutionProperties>
  <WorkDocuments>
    <Order id="Order_id_1">
      <dueDate>08/25/14</dueDate>
      <total xsi:nil="1" />
      <myItems id="Item_id_1">
        <price>10,250000</price>
        <product>Ball</product>
        <quantity>20</quantity>
      </myItems>
    </Order>
    <Order id="Order_id_2">
      <dueDate>07/27/14</dueDate>
      <myItems id="Item_id_4">
        <price>0,050000</price>
        <product>Pencil</product>
        <quantity>100</quantity>
      </myItems>
    </Order>
  </WorkDocuments>
</CorticonRequest>
```

The response specifies German, `de-DE`, as its locale. The messages are in German and the decimal values are delimited correctly:

```
<?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:Corticon"
            xmlns="urn:Corticon" decisionServiceName="Order_localeAware">
      <ns1:ExecutionProperties>
        <ns1:ExecutionProperty name="PROPERTY_EXECUTION_LOCALE"
                               value="de-DE" />
      </ns1:ExecutionProperties>
      <ns1:WorkDocuments>
        <ns1:Order id="Order_id_1">
          <ns1:dueDate>2014-09-25</ns1:dueDate>
          <ns1:shipped>true</ns1:shipped>
          <ns1:shippedOn>2014-04-30T23:00:00.000-05:00</ns1:shippedOn>
          <ns1:total>205,000000</ns1:total>
          <ns1:myItems id="Item_id_1">
            <ns1:price>10,250000</ns1:price>
            <ns1:product>Ball</ns1:product>
            <ns1:quantity>20</ns1:quantity>
            <ns1:subtotal>205,000000</ns1:subtotal>
          </ns1:myItems>
        </ns1:Order>
        <ns1:Order id="Order_id_2">
          <ns1:dueDate>2014-08-27</ns1:dueDate>
          <ns1:shipped>true</ns1:shipped>
          <ns1:shippedOn>2014-04-30T23:00:00.000-05:00</ns1:shippedOn>
          <ns1:total>5,000000</ns1:total>
          <ns1:myItems id="Item_id_4">
            <ns1:price>0,050000</ns1:price>
            <ns1:product>Pencil</ns1:product>
            <ns1:quantity>100</ns1:quantity>
            <ns1:subtotal>5,000000</ns1:subtotal>
          </ns1:myItems>
```

```
        </ns1:Order>
      </ns1:WorkDocuments>
      <ns1:Messages version="1.10">
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>die Zwischensumme der Posten für Pencil ist 5,000000.</ns1:text>
          <ns1:entityReference href="#Item_id_4" />
        </ns1:Message>
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>die Zwischensumme der Posten für Ball ist 205,000000.</ns1:text>
          <ns1:entityReference href="#Item_id_1" />
        </ns1:Message>
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>die Gesamt für die Bestellung 5,000000</ns1:text>
          <ns1:entityReference href="#Order_id_2" />
        </ns1:Message>
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>die Gesamt für die Bestellung 205,000000</ns1:text>
          <ns1:entityReference href="#Order_id_1" />
        </ns1:Message>
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>die Gesamt Paket versenden 05/01/14 12:00:00 AM</ns1:text>
          <ns1:entityReference href="#Order_id_2" />
        </ns1:Message>
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>die Gesamt Paket versenden 05/01/14 12:00:00 AM</ns1:text>
          <ns1:entityReference href="#Order_id_1" />
        </ns1:Message>
      </ns1:Messages>
    </ns1:CorticonResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

This request specifies French, `fr-FR`, as its locale:

```
   <ns1:CorticonResponse xmlns:ns1="urn:Corticon" xmlns="urn:Corticon"
decisionServiceName="Order_localeAware">
      <ns1:ExecutionProperties>
        <ns1:ExecutionProperty name="PROPERTY_EXECUTION_LOCALE"
                               value="fr-FR" />
      </ns1:ExecutionProperties>
     ...
```

The response specifies French as its locale but, while the messages default to English, the decimal values are processed and then delimited correctly:

```
  <ns1:Messages version="1.10">
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>The subtotal of line item for Ball is 205,000000.</ns1:text>
          <ns1:entityReference href="#Item_id_1" />
        </ns1:Message>
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>The subtotal of line item for Pencil is 5,000000.</ns1:text>
          <ns1:entityReference href="#Item_id_4" />
        </ns1:Message>
        ...
```

Example of cross-locale literal dates

When a request provides dates in literal format, the date is transformed into a standard (or default format) `YYYY-MM-DD` form for processing, and is returned in the same format; in other words, the date format in the request is lost. A `dateTime` attribute is returned in Zulu format.

If it is a requirement that the date format in the response be the same as it was in the request, you can stop the server from forcing `dateTime` request values in the response to Zulu format. You can set a server option that specifies that the `date` and `dateTime` formats in the response must be the same as those in the request.

Note: Attributes in a response that were not specified in its request message will have the standard `date` and `dateTime` formats for the locale.

To use literal names for input dates echoed in the response:

1. Stop the server.
2. Locate and edit the `brms.properties` text file.
3. Add (or update) the line
   `com.corticon.ccserver.ensureComplianceWithServiceContract.lenientDateTimeFormat=true`
4. Save the edited file.
5. Start the server.

The following request from `de-DE` is similar to the one in the previous topic except that is submits literal month names, in this case `Sep` and `Okt`:

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 decisionServiceName="Order_localeAware">
  <ExecutionProperties>
    <ExecutionProperty name="PROPERTY_EXECUTION_LOCALE" value="de-DE" />
  </ExecutionProperties>
  <WorkDocuments>
    <Order id="Order_id_1">
      <dueDate>Sep 25, 2014</dueDate>
      <total xsi:nil="1" />
      <myItems id="Item_id_1">
        <price>10,250000</price>
        <product>Ball</product>
        <quantity>20</quantity>
      </myItems>
    </Order>
    <Order id="Order_id_2">
      <dueDate>Okt 9, 2014</dueDate>
      <myItems id="Item_id_4">
        <price>0,050000</price>
        <product>Pencil</product>
        <quantity>100</quantity>
      </myItems>
    </Order>
  </WorkDocuments>
</CorticonRequest>
```

The response handles not only the decimal delimiter and German rule statements, it also adds a month to the dates so it calculates and then replies with `Okt` and `Nov`:


```
<?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:Corticon" xmlns="urn:Corticon"
decisionServiceName="Order_localeAware">
      <ns1:ExecutionProperties>
        <ns1:ExecutionProperty name="PROPERTY_EXECUTION_LOCALE"
                               value="de-DE" />
      </ns1:ExecutionProperties>
      <ns1:WorkDocuments>
        <ns1:Order id="Order_id_1">
          <ns1:dueDate>Okt 26, 2014</ns1:dueDate>
          <ns1:shipped>true</ns1:shipped>
          <ns1:shippedOn>05/01/14 12:00:00 AM</ns1:shippedOn>
          <ns1:total>205,000000</ns1:total>
          <ns1:myItems id="Item_id_1">
            <ns1:price>10,250000</ns1:price>
            <ns1:product>Ball</ns1:product>
            <ns1:quantity>20</ns1:quantity>
            <ns1:subtotal>205,000000</ns1:subtotal>
          </ns1:myItems>
        </ns1:Order>
        <ns1:Order id="Order_id_2">
          <ns1:dueDate>Nov 9, 2014</ns1:dueDate>
          <ns1:shipped>true</ns1:shipped>
          <ns1:shippedOn>05/01/14 12:00:00 AM</ns1:shippedOn>
          <ns1:total>5,000000</ns1:total>
          <ns1:myItems id="Item_id_4">
            <ns1:price>0,050000</ns1:price>
            <ns1:product>Pencil</ns1:product>
            <ns1:quantity>100</ns1:quantity>
            <ns1:subtotal>5,000000</ns1:subtotal>
          </ns1:myItems>
        </ns1:Order>
      </ns1:WorkDocuments>
      <ns1:Messages version="1.10">
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>die Zwischensumme der Posten für Ball ist 205,000000.</ns1:text>
          <ns1:entityReference href="#Item_id_1" />
        </ns1:Message>
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>die Zwischensumme der Posten für Pencil ist 5,000000.</ns1:text>
          <ns1:entityReference href="#Item_id_4" />
        </ns1:Message>
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>die Gesamt für die Bestellung 205,000000</ns1:text>
          <ns1:entityReference href="#Order_id_1" />
        </ns1:Message>
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>die Gesamt für die Bestellung 5,000000</ns1:text>
          <ns1:entityReference href="#Order_id_2" />
        </ns1:Message>
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>die Gesamt Paket versenden 05/01/14 12:00:00 AM</ns1:text>
          <ns1:entityReference href="#Order_id_1" />
        </ns1:Message>
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
```

```
          <ns1:text>die Gesamt Paket versenden 05/01/14 12:00:00 AM</ns1:text>
          <ns1:entityReference href="#Order_id_2" />
        </ns1:Message>
      </ns1:Messages>
    </ns1:CorticonResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Similarly, the following `fr-FR` request is similar to the one in the previous topic except that is submits literal month names, in this case `avril` and `juillet`:

Note: Case is important.

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 decisionServiceName="Order_localeAware">
  <ExecutionProperties>
    <ExecutionProperty name="PROPERTY_EXECUTION_LOCALE"
                       value="fr-FR" />
  </ExecutionProperties>
  <WorkDocuments>
    <Order id="Order_id_1">
      <dueDate>avril 25, 2014</dueDate>
      <total xsi:nil="1" />
      <myItems id="Item_id_1">
        <price>10,250000</price>
        <product>Ball</product>
        <quantity>20</quantity>
      </myItems>
    </Order>
    <Order id="Order_id_2">
      <dueDate>juillet 9, 2014</dueDate>
      <myItems id="Item_id_4">
        <price>0,050000</price>
        <product>Pencil</product>
        <quantity>100</quantity>
      </myItems>
    </Order>
  </WorkDocuments>
</CorticonRequest>
```

The response handles the decimal delimiter and uses English rule statements. It adds a month to the dates so it calculates and then replies with `mai` and `août` (Note that when diacritical marks are used, they must be written appropriately in the request.) :

Note: When diacritical marks are used, they must be written appropriately in the request and are formatted correctly in replies.

```
<?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:Corticon" xmlns="urn:Corticon"
decisionServiceName="Order_localeAware">
      <ns1:ExecutionProperties>
        <ns1:ExecutionProperty name="PROPERTY_EXECUTION_LOCALE"
                               value="fr-FR" />
      </ns1:ExecutionProperties>
```

```
      <ns1:WorkDocuments>
        <ns1:Order id="Order_id_1">
          <ns1:dueDate>mai 26, 2014</ns1:dueDate>
          <ns1:shipped>true</ns1:shipped>
          <ns1:shippedOn>05/01/14 12:00:00 AM</ns1:shippedOn>
          <ns1:total>205,000000</ns1:total>
          <ns1:myItems id="Item_id_1">
            <ns1:price>10,250000</ns1:price>
            <ns1:product>Ball</ns1:product>
            <ns1:quantity>20</ns1:quantity>
            <ns1:subtotal>205,000000</ns1:subtotal>
          </ns1:myItems>
        </ns1:Order>
        <ns1:Order id="Order_id_2">
          <ns1:dueDate>août 9, 2014</ns1:dueDate>
          <ns1:shipped>true</ns1:shipped>
          <ns1:shippedOn>05/01/14 12:00:00 AM</ns1:shippedOn>
          <ns1:total>5,000000</ns1:total>
          <ns1:myItems id="Item_id_4">
            <ns1:price>0,050000</ns1:price>
            <ns1:product>Pencil</ns1:product>
            <ns1:quantity>100</ns1:quantity>
            <ns1:subtotal>5,000000</ns1:subtotal>
          </ns1:myItems>
        </ns1:Order>
      </ns1:WorkDocuments>
      <ns1:Messages version="1.10">
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>The subtotal of line item for Pencil is 5,000000.</ns1:text>
          <ns1:entityReference href="#Item_id_4" />
        </ns1:Message>
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>The subtotal of line item for Ball is 205,000000.</ns1:text>
          <ns1:entityReference href="#Item_id_1" />
        </ns1:Message>
        ...

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

To complete the permutations, an `en_US` on a corresponding system, performs no special operations doe to the locale setting:

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 decisionServiceName="Order_localeAware">
  <ExecutionProperties>
    <ExecutionProperty name="PROPERTY_EXECUTION_LOCALE"
                       value="en-US" />
  </ExecutionProperties>
  <WorkDocuments>
    <Order id="Order_id_1">
      <dueDate>May 25, 2014</dueDate>
      <total xsi:nil="1" />
      <myItems id="Item_id_1">
        <price>10.250000</price>
        <product>Ball</product>
        <quantity>20</quantity>
      </myItems>
    </Order>
    <Order id="Order_id_2">
      <dueDate>May 9, 2014</dueDate>
      <myItems id="Item_id_4">
```

```
        <price>0.050000</price>
        <product>Pencil</product>
        <quantity>100</quantity>
      </myItems>
    </Order>
  </WorkDocuments>
</CorticonRequest>
```

```
<?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:Corticon" xmlns="urn:Corticon"
decisionServiceName="Order_localeAware">
      <ns1:ExecutionProperties>
        <ns1:ExecutionProperty name="PROPERTY_EXECUTION_LOCALE"
                               value="en-US" />
      </ns1:ExecutionProperties>
      <ns1:WorkDocuments>
        <ns1:Order id="Order_id_1">
          <ns1:dueDate>June 25, 2014</ns1:dueDate>
          <ns1:shipped>true</ns1:shipped>
          <ns1:shippedOn>5/1/14 12:00:00 AM</ns1:shippedOn>
          <ns1:total>205.000000</ns1:total>
          <ns1:myItems id="Item_id_1">
            <ns1:price>10.250000</ns1:price>
            <ns1:product>Ball</ns1:product>
            <ns1:quantity>20</ns1:quantity>
            <ns1:subtotal>205.000000</ns1:subtotal>
          </ns1:myItems>
        </ns1:Order>
        <ns1:Order id="Order_id_2">
          <ns1:dueDate>June 9, 2014</ns1:dueDate>
          <ns1:shipped>true</ns1:shipped>
          <ns1:shippedOn>5/1/14 12:00:00 AM</ns1:shippedOn>
          <ns1:total>5.000000</ns1:total>
          <ns1:myItems id="Item_id_4">
            <ns1:price>0.050000</ns1:price>
            <ns1:product>Pencil</ns1:product>
            <ns1:quantity>100</ns1:quantity>
            <ns1:subtotal>5.000000</ns1:subtotal>
          </ns1:myItems>
        </ns1:Order>
      </ns1:WorkDocuments>
      <ns1:Messages version="1.10">
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>The subtotal of line item for Pencil is 5.000000.</ns1:text>
          <ns1:entityReference href="#Item_id_4" />
        </ns1:Message>
        <ns1:Message>
          <ns1:severity>Info</ns1:severity>
          <ns1:text>The subtotal of line item for Ball is 205.000000.</ns1:text>
          <ns1:entityReference href="#Item_id_1" />
        </ns1:Message>
        ...

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

Example of requests that cross timezones

Requests sent to geographically dispersed servers might sense a loss in precision when replies use the server's timezone to calculate time offsets.

Note: Timezone name strings are as presented in the TZ column of the table in Wikipedia's TZ topic. Refer to the Internet Assigned Numbers Authority (IANA) for timezone changes and updated name assignments.

Consider the following example where the request originates in New York City (-5:00 offset from GMT) to a server in Los Angeles (-8:00 offset from GMT):

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 decisionServiceName="timezonetest">
  <WorkDocuments>
    <Entity_1 id="Entity_1_id_1"/>
  </WorkDocuments>
</CorticonRequest>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonResponse xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" decisionServiceName="timezonetest">

  <WorkDocuments>
    <Entity_1 id="Entity_1_id_1">
      <Time1>16:24:35.000-08:00</Time1>
    </Entity_1>
  </WorkDocuments>
  <Messages version="1.0" />
</CorticonResponse>
```

When the request sets its timezone property, the response adjusts the time offset appropriately:

```
 <?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 decisionServiceName="timezonetest">
  <ExecutionProperties>
 <ExecutionProperty name="PROPERTY_EXECUTION_TIMEZONE"
                     value="America/New_York" />
  </ExecutionProperties>
  <WorkDocuments>
    <Entity_1 id="Entity_1_id_1"/>
  </WorkDocuments>
</CorticonRequest>
```

```
 <?xml version="1.0" encoding="UTF-8"?>
<CorticonResponse xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" decisionServiceName="timezonetest">

  <ExecutionProperties>
    <ExecutionProperty name="PROPERTY_EXECUTION_TIMEZONE"
                       value="America/New_York" />
  </ExecutionProperties>
  <WorkDocuments>
    <Entity_1 id="Entity_1_id_1">
      <Time1>16:24:35.000-05:00</Time1>
    </Entity_1>
```

```
  </WorkDocuments>
  <Messages version="1.0" />
</CorticonResponse>
```

When that same server gets a request indicating that it is using Chicago's time, that time offset (-6:00 offset from GMT) is in the reply:

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 decisionServiceName="timezonetest">
  <ExecutionProperties>
 <ExecutionProperty name="PROPERTY_EXECUTION_TIMEZONE"
                     value="America/Chicago" />
  </ExecutionProperties>
  <WorkDocuments>
    <Entity_1 id="Entity_1_id_1"/>
  </WorkDocuments>
</CorticonRequest>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonResponse xmlns="urn:Corticon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" decisionServiceName="timezonetest">

  <ExecutionProperties>
    <ExecutionProperty name="PROPERTY_EXECUTION_TIMEZONE"
                       value="America/Chicago" />
  </ExecutionProperties>
  <WorkDocuments>
    <Entity_1 id="Entity_1_id_1">
      <Time1>15:24:35.000-06:00</Time1>
    </Entity_1>
  </WorkDocuments>
  <Messages version="1.0" />
</CorticonResponse>
```

15

Request and response examples

For details, see the following topics:

- JSON/RESTful request and response messages
- XML requests and responses

JSON/RESTful request and response messages

About creating a JSON request message for a Decision Service

A standardized `JSONObject` can be passed in to `ICcServer.execute(…)`. The `JSONObject` has name-value pair of `"Objects":<JSONArray>`. The Corticon Server will translate `JSONObjects` that are inside the `JSONArray` under the name of `"Objects"`. The `JSONArray` must contain `JSONObjects` that represent Root Entities of the payload.

Basic structure of JSON payload

```
{
    "Objects": [<JSONArray>
    ],
    "__metadataRoot": {
        <execution property>: “<value”,
       .
       .
```

```
    }
}
```

__metadataRoot (optional)

This optional Attribute inside the main `JSONObject` can contain execution specific parameters. (Note that the initial characters are TWO underscores.) These parameters are used only for that execution, and will override a Decision Service or CcServer level properties. The following properties are supported:

- `#restrictInfoRuleMessages`
- `#restrictWarningRuleMessages`
- `#restrictViolationRuleMessages`
- `#restrictResponseToRuleMessagesOnly`
- `#locale`
- `#timezoneId`

The following example uses these properties in `__metadataRoot`:

```
{
    "Objects": [<JSONArray>
    ],
    "__metadataRoot": {
        "#restrictInfoRuleMessages": "true",
        "#restrictViolationRuleMessages": "true",
        "#restrictResponseToRuleMessagesOnly": "true"

    }
}
```

Root Level Entities

All `JSONObjects` inside the `JSONArray` under Attribute `"Objects"` are considered Root Level Entities.

Referred to as “Root”, because the Entity doesn’t have any Parent Entity. “Entity” is a `JSONObject` of data that maps to the Corticon Vocabulary Entity, which can be on the Root of the payload or it can be an Associated Entity, which is referred to as an Association. Associated Entities can either be Embedded Associations or Referenced Associations. The difference between these two types will be covered later in the document.

All Root Entities and embedded Association Entities must have an Attribute called `__metadata` that at minimum describes the type of the Entity. The “`__metadata`” is a String Attribute name with a value of a `JSONObject`. Inside this `JSONObject`, additional name-value pairs are added to describe that Entity.

Mandatory:

`#type` : This is the Entity type as defined in the Vocabulary.

Optional:

`#id`: A unique String value for each Entity. The `#id` field can be used in a Referenced Association where an Association can point to an existing Entity that is in the payload. If Referenced Associations are going to be used in the payload, then a `#id` must be defined for that Associated Entity. Referenced Associations will be covered later in the document.

If `#id` is not supplied in the payload, during execution of rules, a unique value will be set for each Entity. This is done during initial translation from JSON to Corticon Data Objects (CDOs). This is needed because Corticon does not know whether the rules will manipulate the Associations in such a way that `#id` values are needed. The output returned to the user will always contain `#id` value regardless if it was originally part of the “`__metadata`”.

Example of Root Level Entity with `__metadata`:

```
{
    "Objects": [{
        "dMarketValueBeforeTrade": "10216333.000000",
        "price": "950.000000",
       "__metadata": {
            "#id": "Trade_id_1",
            "#type": "Trade"
        }
    }],
}
```

JSON Entity Attribute and Association name-value pairs

All Entities can contain Attribute name-value pairs along with Association Role name-value pairs.

Attribute name-value pairs:

Each JSON Entity can have any number of Attribute name-value pairings. The Attribute names inside the JSON Entity correspond to what has been defined in the Vocabulary for that JSON Entity type. The Attribute name inside the JSON Entity is used to look up the corresponding Vocabulary Attribute for that Vocabulary Entity type. If JSON Entity Attributes don't match with any Vocabulary Entity Attribute, then the JSON Entity Attribute is ignored, and won't be used by the rules.

The "value" associate with "name" doesn't have to be a String. However, the "value" must be of proper form to be converted into the Datatype as defined in the Vocabulary Attribute's Datatype.

The JSON Datatypes that can be used as a "value" are:

- String
- Boolean
- Double
- Int
- Long

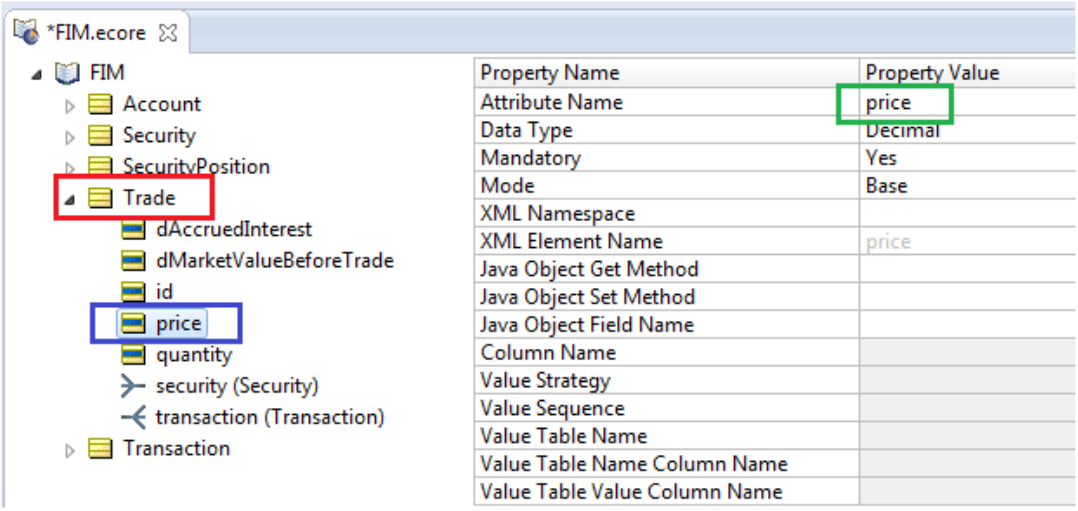
For a Date "value", use a String to represent the Date, which will be converted into a proper Date object for rule processing.

If the "value" cannot be properly converted into the Vocabulary Attribute's Datatype, a `CcServerExecutionException` will be thrown informing the user that the CcServer failed to convert the JSON "values".

Example of Attribute name-value pairs:

There is one Attribute, "`price`", with a corresponding value. Based on the `__metadata : #type`, these Attribute values are looked up under the Vocabulary Trade Entity.

```
{
    "Objects": [{
        "price": "950.000000",
       "__metadata": {
            "#id": "Trade_id_1",
            "#type": "Trade"
        }
    }],
}
```



Association name-value pairs

Each JSON Entity can have any number of Association name-value pairings. The Association names inside the JSON Entity correspond to an Vocabulary Entity Association Role Name, defined in the Vocabulary for that JSON Entity type. Like the Attribute, as described above, Association names inside the JSON Entity are used to look up the corresponding Vocabulary Association for that Vocabulary Entity type. If JSON Entity Association names don't match with any Vocabulary Entity Association Role Name, then the JSON Entity Association is ignored, and won't be used by the rules.

The "value" associate with "name" can be either a `JSONObject` or a `JSONArray` (of other JSON Objects). However, it is possible that if the original "value" was a `JSONObject`, a `JSONArray` may be in the output. If there is a rule that does a += operator on the Association, the `JSONObject` will be converted into a `JSONArray` so that multiple `JSONObjects` can be associated with that "name".

Also, the Associated `JSONObject`, the "value", can be a Referenced Associated Object, which points to another `JSONObject` in the payload. In this scenario, a "ref_id" is use to point to the intended Entity. As described above, the `#type` value is not needed when a Referenced Associated Object is used because the "type" can be inferred by the Rules Engine.

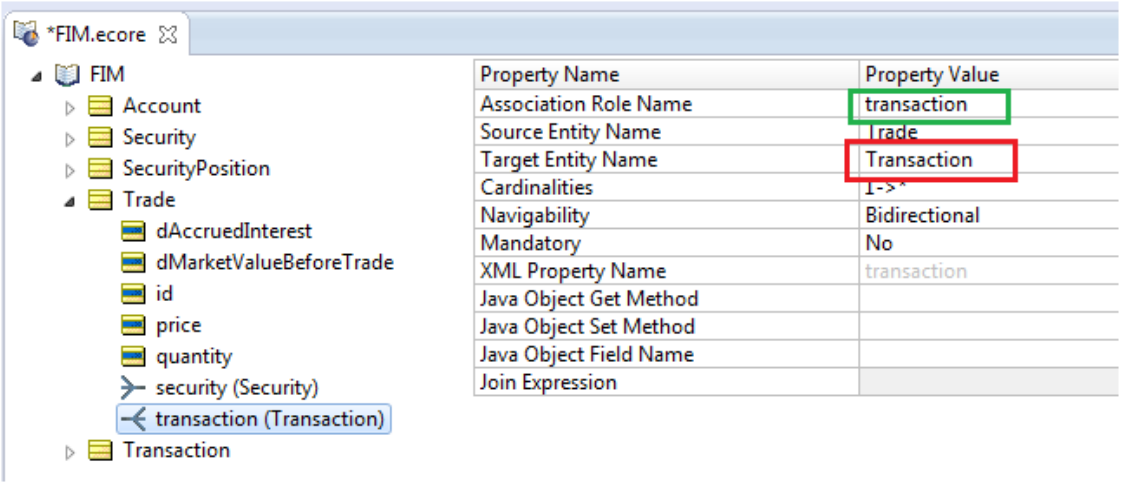
Example of Embedded Association name-value pairs:

There is one Association, "transaction", with a corresponding JSON Object as a value. This is an Imbedded Association, which is an Entity under another Entity. The Transaction Entity, as defined by its `__metadata : #type = Transaction` is associated with Trade through a Role Name of "transaction"

```
{
    "Objects": [{
        "dMarketValueBeforeTrade": "10216333.000000",
        "price": "950.000000",
        “transaction": [
            {
                "__metadata": {
                    "#ref_id": "Transaction_id_1",
                }
            }],
        },
        "__metadata": {
            "#id": "Trade_id_1",
            "#type": "Trade"
        }
    },
    {
        "maxPctHiYield": "35.000000",
        "__metadata": {
           "#id": "Transaction_id_1",
           "#type": "Transaction"
        }
     }],
]
```

Example of Referenced Association name-value pairs:

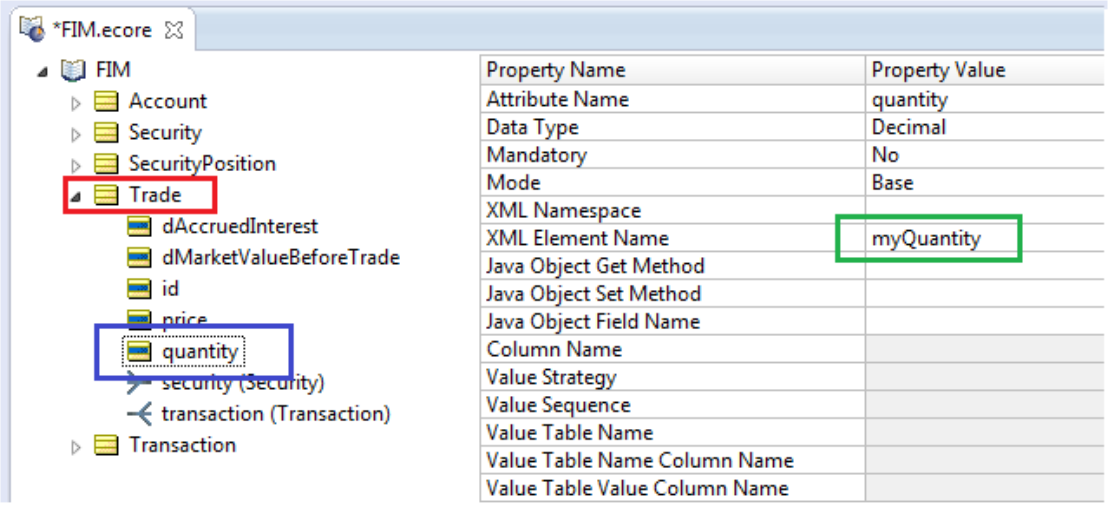
```
{
    "Objects": [{
        "dMarketValueBeforeTrade": "10216333.000000",
        "price": "950.000000",
        “transaction": [
            {
                "maxPctHiYield": "35.000000",
                "__metadata": {
                    "#id": "Transaction_id_1",
                    "#type": "Transaction"
                }
            }],
        },
        "__metadata": {
            "#id": "Trade_id_1",
            "#type": "Trade"
        }
    }],
]
```



XML Element Name overrides for Attributes and Association names

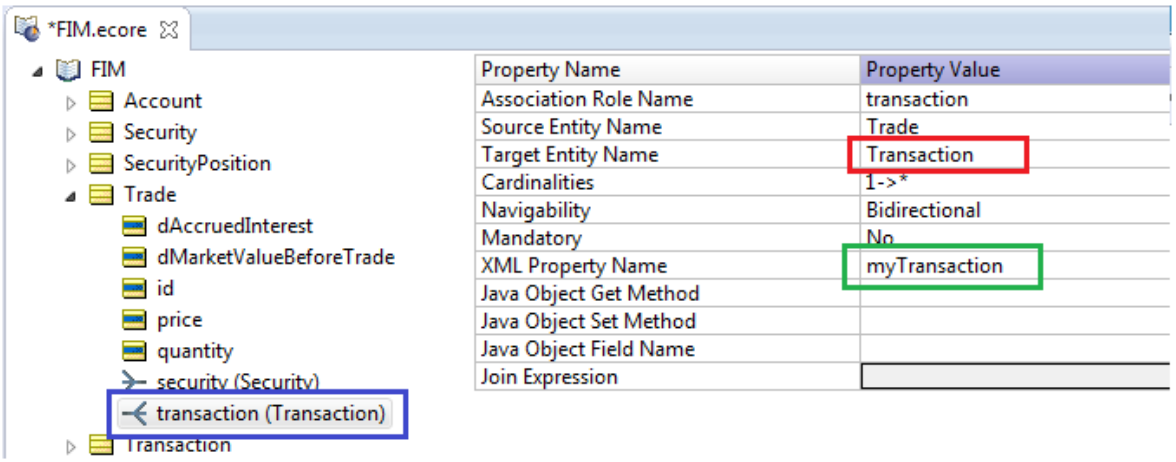
JSON Entity Attribute names are first matched against XML Name overrides, which are defined in the Vocabulary Attribute. If no XML Element Name is defined, then JSON Entity Attribute names are matched directly against the Vocabulary Attribute name.

```
{
    "Objects": [{
        "dMarketValueBeforeTrade": "10216333.000000",
        "price": "950.000000",
        "myQuantity": "100.000000",
      "__metadata": {
            "#id": "Trade_id_1",
            "#type": "Trade"
        }
    }],
}
```



Much like the Attribute's XML Element Name override, Associations also have an XML Element Override.

```
{
    "Objects": [{
        "price": "950.000000",
        “myTransaction": [
            {
                "maxPctHiYield": "35.000000",
                "__metadata": {
                    "#id": "Transaction_id_1",
                    "#type": "Transaction"
                }
            }],
        },
        "__metadata": {
            "#id": "Trade_id_1",
            "#type": "Trade"
        }
    }],
]
```



Sample JSON request and response messages

You can set the Decision Service being called into the URL, or include in the message header.

Passing the Decision Service as a URL parameter

A `GET` request lets you add URL encoding of parameters so that you can pass a Decision Service and its version in URL parameters instead of the message header. . For example, the following request would get the properties for a Decision Service `ProcessOrder` version 1.16, with the `GET` parameters shown in bold:

```
http://<serverHost>:8850/axis/corticon/decisionService/getProperties
        ?name=ProcessOrder&majorVersion=1&minorVersion=16
```

Evaluation of a request will look first for the URL parameter, and if not found, it will look for it as an HTTP header.

Setting the Decision Service information in the HTTP(S) header

The Decision Service payload is enclosed in the request message's body in the JSON object form. The JSON object structure is specified in a separate document.

You must specify the Decision Service name in the HTTP header field `dsname`, as shown in this example for `ProcessOrder`:

```
content-type:application/json
dsName:ProcessOrder
content-length:5479
host:localhost:8850
connection:Keep-Alive
user-agent:Apache-HttpClient/4.3.4 (java 1.5)
accept-encoding:gzip,deflate
```

Because neither a version or effective date is provided, the highest major version's highest minor version applies.

```
content-type:application/json
dsName:ProcessOrder
dsMajorVersion:1
...
```

With only the major version provided, the highest minor version of that major version applies.

```
content-type:application/json
dsName:ProcessOrder
dsMajorVersion:1
dsMinorVersion:10
...
```

When both major and minor version are provided, that version of the Decision Service handles the request.

```
content-type:application/json
dsName:ProcessOrder
dsEffectiveTimestamp:12/11/2015
...
```

When the header provides effective timestamp **instead of major or major/minor version**, the request is handled by the Decision Service that is in effect within the range of the given date.

How the CorticonExecuteRest REST API reads the JSON request header

Once the header is complete, the request connects to the host, and submits the JSON request to the designated Decision Service.

The CorticonExecuteRest REST API looks inside the header for the version numbers and effective date information.

Code inside CorticonExecuteRest REST API:

```
String lstrDecisionServiceName               =
request.getHeader(ICcServer.REST_HEADER_DECISION_SERVICE_NAME);
String lstrDecisionServiceMajorVersion       =
request.getHeader(ICcServer.REST_HEADER_DECISION_SERVICE_MAJOR_VERSION);
String lstrDecisionServiceMinorVersion       =
request.getHeader(ICcServer.REST_HEADER_DECISION_SERVICE_MINOR_VERSION);
String lstrDecisionServiceEffectiveTimestamp =
request.getHeader(ICcServer.REST_HEADER_DECISION_SERVICE_EFFECTIVE_TIMESTAMP);
```

Constant values inside ICcServer:

```
public static final String REST_HEADER_DECISION_SERVICE_NAME = "dsName";
public static final String REST_HEADER_DECISION_SERVICE_MAJOR_VERSION = "dsMajorVersion";
public static final String REST_HEADER_DECISION_SERVICE_MINOR_VERSION = "dsMinorVersion";
```

```
public static final String REST_HEADER_DECISION_SERVICE_EFFECTIVE_TIMESTAMP =
"dsEffectiveTimestamp";
```

It is recommended that you use the ICcServer Constant values rather than the literal values inside your application code.

Basic structure of JSON output

The original JSON Input is updated dynamically when rules fire. However, at the end of rule processing, the Rule Messages from the rules, are added to the JSON output and returned to the user. On the original JSON Object, a new Attribute named "`Messages`" is added. The "`value`" could be null or a JSON Object. If the value is a null, that means that no Rule Messages were created by rule processing. If the value is a JSON Object, then there will be an Attribute "`name`" of "`Message`" with a "`value`" of `JSONArray`. Each JSON Object in the JSON Array represents one Rule Message that was created during rule processing. Inside each JSON Object, there will be an "`entityReference`", "`text`", "`severity`", and also a "`__metadata`". The first three map to the Rule Message in the rules, and the "`__metadata`" contains the `#type` to ensure a good type casting.

```
{
 "Objects": [<JSONArray>
 ],
 "__metadataRoot": {
  <execution property>: "<value",
  .
  .
 },
 "Messages": {
  "Message": [
   {
    "entityReference": "<#id of referenced Entity>",
    "text": "<Rule Messages Text>",
    "severity": "<Severity>",
    "__metadata":
    {
     "#type": "#RuleMessage"}
    },
    "version": "<version of the Decision Service>"
  },
 }
}
```

Example of Messages in the output returned to user:

```
{
 "Messages": {
  "Message": [
  {
   "entityReference": "Trade_id_1",
   "text": "[AccountConstraint,5] A restricted account [ Sears ] can't be involved in a
 trade.",
    "severity": "Warning",
    "__metadata": {"#type": "#RuleMessage"}
  },
  {
    "entityReference": "Trade_id_1",
    "text": "[AccountConstraint,4] No account [ Airbus ] involved in a trade can exceed
 its maximum
            percentage [ 70.000000 ] for High Yield securities [ 86.156842

].",
    "severity": "Warning",
    "__metadata": {"#type": "#RuleMessage"}
  },
  {
```

```
    "entityReference": "Trade_id_1",
    "text": "[AccountConstraint,4] No account [ Sears ] involved in a trade can exceed
its maximum
            percentage [ 65.000000 ] for High Yield securities [ 79.980241
].",
    "severity": "Warning",
    "__metadata": {"#type": "#RuleMessage"}
   },
   "__metadata": {"#type": "#RuleMessages"},
   "version": "0.0"
  },
  "Objects": [{
    "dMarketValueBeforeTrade": 20432666,
    "price": "950.000000",
    "transaction": [
      {
       "dPositionHiGrade": 0,
       "dPositionHiYield": 269397.538944,
       "dAccruedInterest": 2249.666294,
.
.
.
.
}
```


Sample JSON Input and Output

The following code presents the input and output of the `TradeAllocation` sample's `AllocateTrade` Tester.

JSON Input:


```
{
 "Objects": [{
  "dMarketValueBeforeTrade": "10216333.000000",
  "price": "950.000000",
  "transaction": [
  {
   "account": [{
    "maxPctHiYield": "35.000000",
    "dPositionHiYield": "330000.000000",
    "dPositionHiGrade": "819167.000000",
    "securityPosition": [
    {
     "quantity": "5000",
     "dMarketValue": "819167.000000",
     "security": [{
      "symbol": "PMBND",
      "yield": "6.000000",
      "daysInHolding": "23",
      "dMarketValue": "164.000000",
      "dProfile": "HI-GRD",
      "dAnnualInterestAmt": "60.000000",
      "price": "160.000000",
      "issuer": "Phillip Morris",
      "sin": "Y",
      "dAccruedInterest": "4.000000",
      "faceValue": "1000.000000",
      "rating": "A",
      "__metadata": {
      "#id": "Security_id_1",
      "#type": "Security"
      }
     }],
     "__metadata": {
      "#id": "SecurityPosition_id_1",
      "#type": "SecurityPosition"
     }
```

```
    },
    {
     "quantity": "3000",
     "dMarketValue": "330000.000000",
     "security": [{
      "symbol": "3MBND",
      "yield": "12.000000",
      "daysInHolding": "40",
      "dMarketValue": "110.000000",
      "dProfile": "HI-YLD",
      "dAnnualInterestAmt": "90.000000",
      "price": "100.000000",
      "issuer": "3M",
      "sin": "N",
      "dAccruedInterest": "10.000000",
      "faceValue": "1000.000000",
      "rating": "B",
      "__metadata": {
      "#id": "Security_id_3",
      "#type": "Security"
      }
     }],
     "__metadata": {
      "#id": "SecurityPosition_id_2",
      "#type": "SecurityPosition"
     }
    }
    ],
    "warnMargin": "3.000000",
    "name": "Boeing",
    "restricted": "false",
    "maxPctHiGrade": "75.000000",
    "number": "1640",
    "dMarketValue": "1149167.000000",
    "__metadata": {
    "#id": "Account_id_1",
    "#type": "Account"
    }
   }],
   "security": [{"__metadata": {"#ref_id": "Security_id_2"}}],
   "__metadata": {
    "#id": "Transaction_id_1",
    "#type": "Transaction"
   }
  },
  {
   "account": [{
    "maxPctHiYield": "65.000000",
    "dPositionHiYield": "2495556.000000",
    "dPositionHiGrade": "819167.000000",
    "securityPosition": [
    {
     "quantity": "5000",
     "dMarketValue": "819167.000000",
     "security": [{"__metadata": {"#ref_id": "Security_id_1"}}],
     "__metadata": {
      "#id": "SecurityPosition_id_3",
      "#type": "SecurityPosition"
     }
    },
    {
     "quantity": "2000",
     "dMarketValue": "295556.000000",
     "security": [{"__metadata": {"#ref_id": "Security_id_2"}}],
     "__metadata": {
      "#id": "SecurityPosition_id_4",
      "#type": "SecurityPosition"
     }
    },
```

```
    {
     "quantity": "20000",
     "dMarketValue": "2200000.000000",
     "security": [{"__metadata": {"#ref_id": "Security_id_3"}}],
     "__metadata": {
      "#id": "SecurityPosition_id_5",
      "#type": "SecurityPosition"
     }
    }
    ],
    "warnMargin": "3.000000",
    "name": "Sears",
    "restricted": "true",
    "maxPctHiGrade": "45.000000",
    "number": "1920",
    "dMarketValue": "3314722.000000",
    "__metadata": {
    "#id": "Account_id_2",
    "#type": "Account"
    }
   }],
   "security": [{"__metadata": {"#ref_id": "Security_id_2"}}],
   "__metadata": {
    "#id": "Transaction_id_2",
    "#type": "Transaction"
   }
  },
  {
   "account": [{
    "maxPctHiYield": "70.000000",
    "dPositionHiYield": "4769444.000000",
    "dPositionHiGrade": "983000.000000",
    "securityPosition": [
    {
     "quantity": "6000",
     "dMarketValue": "983000.000000",
     "security": [{"__metadata": {"#ref_id": "Security_id_1"}}],
     "__metadata": {
      "#id": "SecurityPosition_id_6",
      "#type": "SecurityPosition"
     }
    },
    {
     "quantity": "2500",
     "dMarketValue": "369444.000000",
     "security": [{"__metadata": {"#ref_id": "Security_id_2"}}],
     "__metadata": {
      "#id": "SecurityPosition_id_7",
      "#type": "SecurityPosition"
     }
    },
    {
     "quantity": "40000",
     "dMarketValue": "4400000.000000",
     "security": [{"__metadata": {"#ref_id": "Security_id_3"}}],
     "__metadata": {
      "#id": "SecurityPosition_id_8",
      "#type": "SecurityPosition"
     }
    }
    ],
    "warnMargin": "2.000000",
    "name": "Airbus",
    "restricted": "false",
    "maxPctHiGrade": "35.000000",
    "number": "2750",
    "dMarketValue": "5752444.000000",
    "__metadata": {
    "#id": "Account_id_3",
```

```
    "#type": "Account"
    }
   }],
   "security": [{"__metadata": {"#ref_id": "Security_id_2"}}],
   "__metadata": {
    "#id": "Transaction_id_3",
    "#type": "Transaction"
   }
  }
  ],
  "dAccruedInterest": "38889.000000",
  "quantity": "5000.000000",
  "security": {
  "symbol": "BGBND",
  "yield": "11.000000",
  "daysInHolding": "40",
  "dMarketValue": "148.000000",
  "dProfile": "HI-YLD",
  "dAnnualInterestAmt": "80.000000",
  "price": "140.000000",
  "issuer": "Boeing",
  "sin": "N",
  "dAccruedInterest": "8.000000",
  "faceValue": "1000.000000",
  "rating": "BBB",
  "__metadata": {
   "#id": "Security_id_2",
   "#type": "Security"
  }
  },
  "__metadata": {
  "#id": "Trade_id_1",
  "#type": "Trade"
  }
 }],
 "__metadataRoot": {
  "#restrictInfoRuleMessages": "true",
  "#restrictViolationRuleMessages": "true",
  "#restrictWarningRuleMessages": "false"
 }
}
JSON TRANSLATION = 78
```


JSON Output:


```
{
  "Messages": {
    "Message": [
      {
       "entityReference": "Trade_id_1",
       "text": "[AccountConstraint,5] A restricted account [ Sears ] can't be involved
in a trade.",
       "severity": "Warning",
       "__metadata": {"#type": "#RuleMessage"}
      },
      {
       "entityReference": "Trade_id_1",
       "text": "[AccountConstraint,4] No account [ Airbus ] involved in a trade can
exceed
              its maximum percentage [ 70.000000 ] for High Yield securities [ 86.156842
 ].",
       "severity": "Warning",
       "__metadata": {"#type": "#RuleMessage"}
      },
      {
       "entityReference": "Trade_id_1",
      "text": "[AccountConstraint,4] No account [ Sears ] involved in a trade can exceed
```

```
              its maximum percentage [ 65.000000 ] for High Yield securities [ 79.980241
 ].",
       "severity": "Warning",
       "__metadata": {"#type": "#RuleMessage"}
      },
      {
       "entityReference": "Trade_id_1",
       "text": "[AccountConstraint,4] No account [ Boeing ] involved in a trade can
exceed
              its maximum percentage [ 35.000000 ] for High Yield securities [ 42.253808
 ].",
       "severity": "Warning",
       "__metadata": {"#type": "#RuleMessage"}
      }
    ],
    "__metadata": {"#type": "#RuleMessages"},
    "version": "0.0"
  },
  "Objects": [{
    "dMarketValueBeforeTrade": 20432666,
    "price": "950.000000",
    "transaction": [
      {
       "dPositionHiGrade": 0,
       "dPositionHiYield": 269397.538944,
       "dAccruedInterest": 2249.666294,
       "dActualQuantity": 281.208287,
       "account": [{
         "dPctHiYield": 42.253808,
         "dPositionHiYield": "330000.000000",
         "dNewPositionHiGrade": 819167,
         "maxPctHiGrade": "75.000000",
         "restricted": "false",
         "dMarketValue": "1149167.000000",
         "number": "1640",
         "dPositionHiGrade": "819167.000000",
         "maxPctHiYield": "35.000000",
         "dNewPositionHiYield": 599397.538944,
         "name": "Boeing",
         "warnMargin": "3.000000",
         "securityPosition": [
          {
            "quantity": "5000",
            "dMarketValue": "819167.000000",
            "security": [{
            "symbol": "PMBND",
            "yield": "6.000000",
            "daysInHolding": "23",
            "dMarketValue": "164.000000",
            "dProfile": "HI-GRD",
            "dAnnualInterestAmt": "60.000000",
            "price": "160.000000",
            "issuer": "Phillip Morris",
            "sin": "Y",
            "dAccruedInterest": "4.000000",
            "faceValue": "1000.000000",
            "rating": "A",
            "__metadata":
             {
               "#id": "Security_id_1",
               "#type": "Security"
              }
            }],
            "__metadata":{
               "#id": "SecurityPosition_id_1",
               "#type": "SecurityPosition"
             }
          },
          {
```

```
        "quantity": "3000",
        "dMarketValue": "330000.000000",
        "security": [{
            "symbol": "3MBND",
            "yield": "12.000000",
            "daysInHolding": "40",
            "dMarketValue": "110.000000",
            "dProfile": "HI-YLD",
            "dAnnualInterestAmt": "90.000000",
            "price": "100.000000",
            "issuer": "3M",
            "sin": "N",
            "dAccruedInterest": "10.000000",
            "faceValue": "1000.000000",
            "rating": "B",
            "__metadata":{
              "#id": "Security_id_3",
              "#type": "Security"
             }
          }],
          "__metadata":{
            "#id": "SecurityPosition_id_2",
            "#type": "SecurityPosition"
           }
         }
      ],
      "__metadata":{
         "#id": "Account_id_1",
         "#type": "Account"
      },
      "dNewMarketValue": 1418564.538944,
      "dPctHiGrade": 57.746192
    }],
    "dMarketValue": 269397.538944,
    "security": [{"__metadata": {"#ref_id": "Security_id_2"}}],
    "dQuantity": 281.208287,
    "__metadata":{
      "#id": "Transaction_id_1",
      "#type": "Transaction"
    },
    "dPrice": 950
  },
  {
    "dPositionHiGrade": 0,
    "dPositionHiYield": 777065.429329,
    "dAccruedInterest": 6489.064129,
    "dActualQuantity": 811.133016,
    "account": [{
      "dPctHiYield": 79.980241,
      "dPositionHiYield": "2495556.000000",
      "dNewPositionHiGrade": 819167,
      "maxPctHiGrade": "45.000000",
      "restricted": "true",
      "dMarketValue": "3314722.000000",
      "number": "1920",
      "dPositionHiGrade": "819167.000000",
      "maxPctHiYield": "65.000000",
      "dNewPositionHiYield": 3272621.429329,
      "name": "Sears",
      "warnMargin": "3.000000",
      "securityPosition": [{
         "quantity": "5000",
         "dMarketValue": "819167.000000",
         "security": [{"__metadata": {"#ref_id": "Security_id_1"}}],
         "__metadata": {
            "#id":  "SecurityPosition_id_3",
            "#type": "SecurityPosition"
           }
       },
```

```
      {
        "quantity": "2000",
        "dMarketValue": "295556.000000",
        "security": [{"__metadata": {"#ref_id": "Security_id_2"}}],
        "__metadata": {
          "#id": "SecurityPosition_id_4",
          "#type": "SecurityPosition"
        }
      },
      {
        "quantity": "20000",
        "dMarketValue": "2200000.000000",
        "security": [{"__metadata": {"#ref_id": "Security_id_3"}}],
        "__metadata": {
          "#id": "SecurityPosition_id_5",
          "#type": "SecurityPosition"
        }
      }
    ],
    "__metadata":{
        "#id": "Account_id_2",
        "#type": "Account"
    },
    "dNewMarketValue": 4091787.429329,
    "dPctHiGrade": 20.019784
  }],
  "dMarketValue": 777065.429329,
  "security": [{"__metadata": {"#ref_id": "Security_id_2"}}],
  "dQuantity": 811.133016,
  "__metadata": {
     "#id": "Transaction_id_2",
     "#type": "Transaction"
  },
  "dPrice": 950
},
{
  "dPositionHiGrade": 0,
  "dPositionHiYield": 1348537.031727,
  "dAccruedInterest": 11261.269577,
  "dActualQuantity": 1407.658697,
  "account": [{
    "dPctHiYield": 86.156842,
    "dPositionHiYield": "4769444.000000",
    "dNewPositionHiGrade": 983000,
    "maxPctHiGrade": "35.000000",
    "restricted": "false",
    "dMarketValue": "5752444.000000",
    "number": "2750",
    "dPositionHiGrade": "983000.000000",
    "maxPctHiYield": "70.000000",
    "dNewPositionHiYield": 6117981.031727,
    "name": "Airbus",
    "warnMargin": "2.000000",
    "securityPosition": [{
        "quantity": "6000",
        "dMarketValue": "983000.000000",
        "security": [{"__metadata": {"#ref_id": "Security_id_1"}}],
        "__metadata": {
          "#id": "SecurityPosition_id_6",
          "#type": "SecurityPosition"
         }
      },
      {
        "quantity": "2500",
        "dMarketValue": "369444.000000",
        "security": [{"__metadata": {"#ref_id": "Security_id_2"}}],
        "__metadata": {
          "#id": "SecurityPosition_id_7",
          "#type": "SecurityPosition"
```

```
                }
              },
             {
               "quantity": "40000",
               "dMarketValue": "4400000.000000",
               "security": [{"__metadata": {"#ref_id": "Security_id_3"}}],
               "__metadata": {
                  "#id": "SecurityPosition_id_8",
                  "#type": "SecurityPosition"
                  }
              }
            ],
            "__metadata": {
               "#id": "Account_id_3",
               "#type": "Account"
              },
            "dNewMarketValue": 7100981.031727,
            "dPctHiGrade": 13.843158
          }],
          "dMarketValue": 1348537.031727,
          "security": [{"__metadata": {"#ref_id": "Security_id_2"}}],
          "dQuantity": 1407.658697,
          "__metadata": {
             "#id": "Transaction_id_3",
             "#type": "Transaction"
          },
          "dPrice": 950
         }
      ],
      "dAccruedInterest": 40000,
      "quantity": "5000.000000",
      "security": {
        "symbol": "BGBND",
        "yield": "11.000000",
        "daysInHolding": "40",
        "dMarketValue": "148.000000",
        "dProfile": "HI-YLD",
        "dAnnualInterestAmt": "80.000000",
        "price": "140.000000",
        "issuer": "Boeing",
        "sin": "N",
        "dAccruedInterest": "8.000000",
        "faceValue": "1000.000000",
        "rating": "BBB",
        "__metadata": {
         "#id": "Security_id_2",
         "#type": "Security"
        }
      },
      "__metadata": {
        "#id": "Trade_id_1",
        "#type": "Trade"
      }
    }],
    "__metadataRoot": {
      "#restrictInfoRuleMessages": "true",
      "#restrictViolationRuleMessages": "true",
      "#restrictWarningRuleMessages": "false"
    }
  }
```


Testing a JSON request

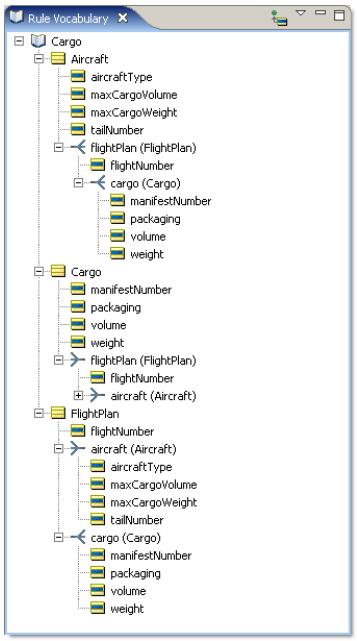
Your installation provides a sample JSON request as well as an API test to run the request. For an example of running a Corticon JSON-formatted request on Corticon Java server, see Path 4: Using JSON/RESTful client to consume a Decision Service on Java Server, and on Corticon Server for .NET, see Path 4: Using JSON/RESTful client to consume a Decision Service on .NET server.

XML requests and responses

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.



Sample XML 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">
```

Optional execution properties can be set in the request to override default values on the server. The available execution properties, set here to other than their default value, are as follows:

```
  <ExecutionProperties>
    <ExecutionProperty
       name="PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_INFO"
       value="true" />
    <ExecutionProperty
       name="PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_WARNING"
       value="true" />
    <ExecutionProperty
       name="PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_VIOLATION"
       value="true" />
```

```
    <ExecutionProperty
       name="PROPERTY_EXECUTION_RESTRICT_RESPONSE_TO_RULEMESSAGES_ONLY"
       value="true" />
    <ExecutionProperty
       name="PROPERTY_EXECUTION_LOCALE"
       value="fr-FR" />
    <ExecutionProperty
       name="PROPERTY_EXECUTION_TIMEZONE"
       value="America/Chicago" />
  </ExecutionProperties>
```

Notice the unique id 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 XML 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 kilograms 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>
```

16

Implementing Rule Execution Recording in a database

Corticon Decision Service execution is stateless, where all state is maintained in the message payloads. A single incoming message payload seeds the engine's working memory after which rules processing commences. Once rules processing is complete, the final state of the engine's working memory is returned as a response. The response message payload includes two discrete sets of data: data payload and posted messages that can each include data values as well internal information about the sequence of rules firing. While some data might have been stored in a database through an Enterprise Data Connection, the complete message payload is flushed from memory.

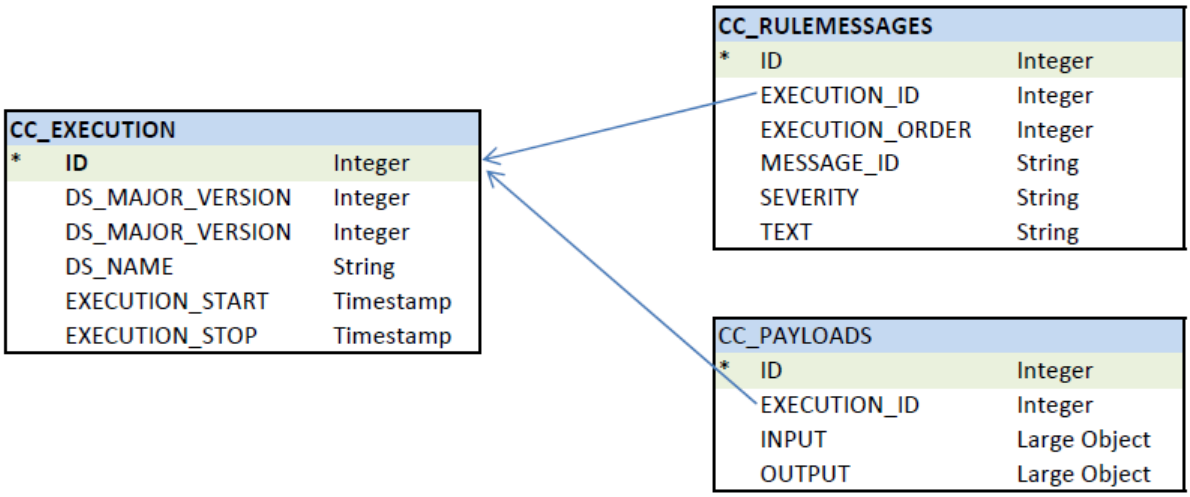
There are several reasons to want to retain payloads and rule messages:

- Auditors might be assigned to determine how certain results were derived from a rule processing. When the exact request and its response payload can be accessed, the auditors can precisely reconstruct a decision.
- Rules always evolve. Developers of rules can replay a 'real' request to see how rule changes impact responses as well as performance.
- Rule modelers can check rule coverage in a large set of 'real' rule messages to see if rules considered obsolete are really not getting any use.
- Rule modelers can also track down common sequences of rule activities to reveal race conditions or re-entrant rules.

Corticon's Rule Execution Recording feature records all input payloads sent to a Decision Service, all the rule messages produced processing the payloads, and the returned payloads. This provides documentation of the activity of a Decision Service. When setup and activated at the server level, each Decision Service deployed on the server must opt in to use the server's rule execution recording mechanism.

Overview of schema for storing execution payloads and rule messages

The general pattern of the schema for the necessary tables in a database you create are as follows:



Within the schema, each Decision Service version execution is timestamped with its time interval, and provided a unique primary key. Payloads and rule messages are in separate tables, each with a unique primary as well as the foreign key to link it to its execution identifier.

Consider:

- The database you use for rule execution recording could be, and typically should be, distinct from the enterprise database of record. Rule execution recording can be used independent of Corticon EDC.
- Multiple Corticon Servers could connect to and record execution in the same database.
- Where rule tracing has been implemented, rule messages will be prepended with the metadata of the rule that fired. (Enabled by uncommenting the `brms.properties` line `#com.corticon.reactor.rulestatement.metadata=false` and then setting it to `true`)
- The process is asynchronous "fire-and-forget" from Corticon Server. Data that accrues from this feature in your database is entirely your responsibility. You must provide adequate performance, storage, backup, rollover, and reporting mechanisms. Correspondingly, Corticon Servers and Studio does not read from the execution recording database at any time.
- This feature is not supported on Studio's embedded server. When a remote server is used as the Studio's test subject, that server could be performing execution recording.

Creating the database schema for Rule Execution Recording

The first step is to setup the schema in your preferred database. Corticon Studio provides a wizard that lets you:

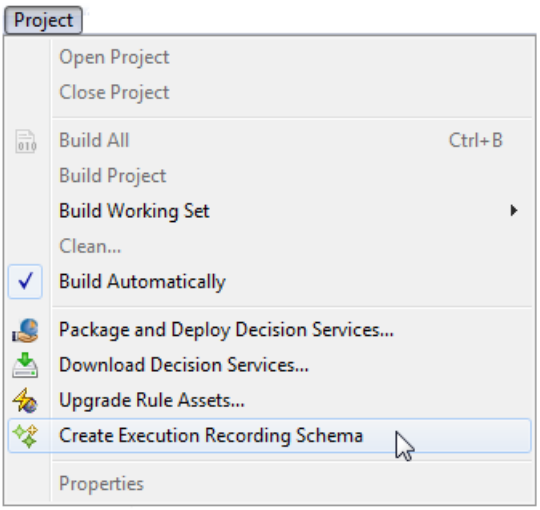
- Define and test the connection to the target database
- Create the default schema for the database brand in the target database
- Define the properties and encrypted credentials that a server will add to its `brms.properties` file to connect to that database

---

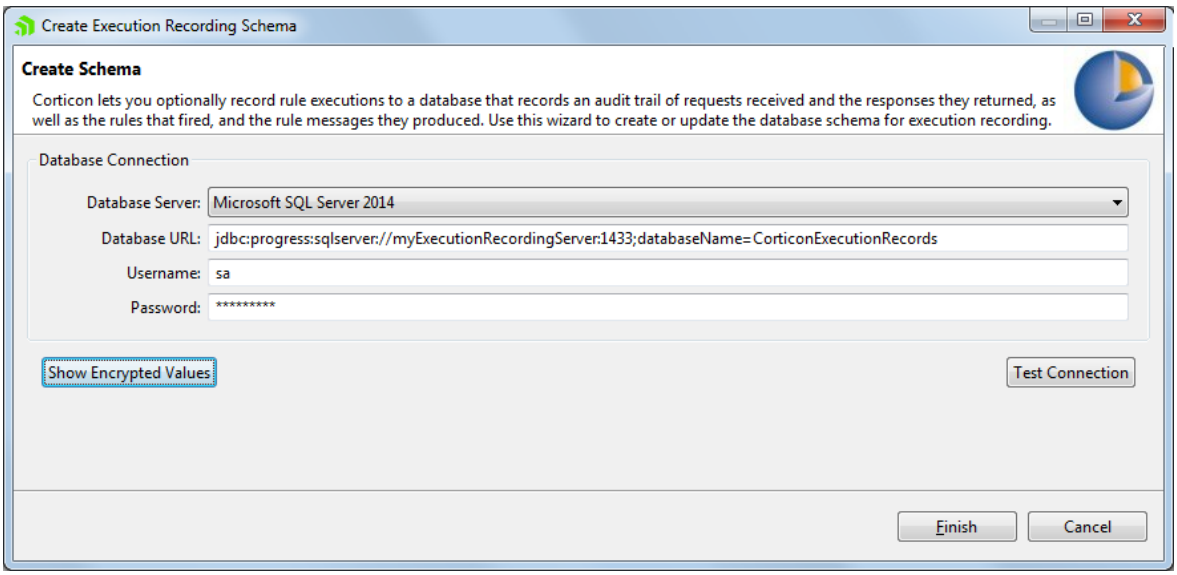
Note: Your database administrator might prefer to create the schema by using (or adapting) the SQL script samples supplied in a Corticon Server installation at `[CORTICON_HOME]/Server/src/sql`. The database connection parameters can also be created entirely in a server's `brms.properties` file, although the credentials would have to be in plain text as the decryption algorithm requires that the credentials (either or both username and password) are encrypted by the algorithm applied in Studio.

To test and generate the connection parameters

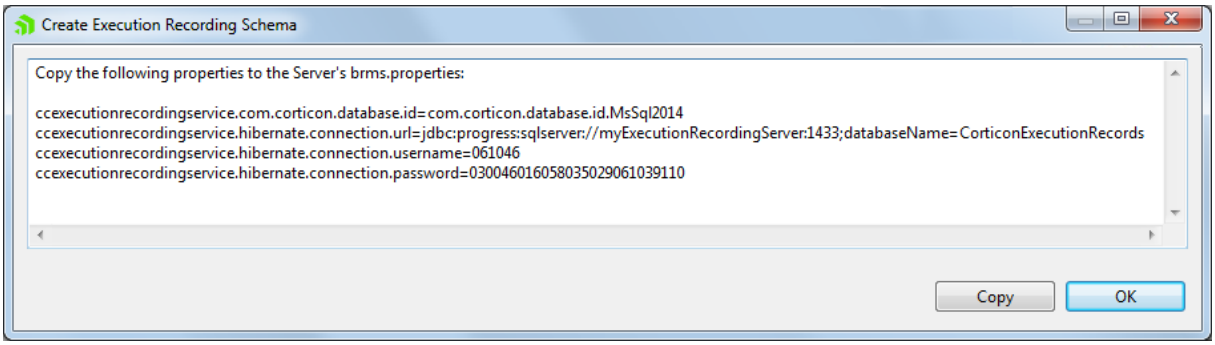
1. In the target database's administrative tool, locate or create the named database instance for recording executions.
2. In Studio, choose the menu command **Project > Create Execution Recording Schema**, as shown:



3. In the **Create Execution Recording Schema** dialog, choose the **Database Server** brand and version brand from the dropdown list. Then edit the **Database URL** that is displayed to change `<server>:nnnn` to your server *host:port*, and `<database_name>` to the name you created. Then enter appropriate username and password credentials for that database.



4. Click **Test Connect** to ensure that the information achieves connection.
5. Click **Show Encrypted Values** to get the connection properties and values that you will move to each participating server.



6. Click **Copy** to put the connection information on the clipboard, click **OK**. Paste the information into a text file that will be transferred to Server locations for inclusion in their `brms.properties`.

Note: This completes the testing of the connection information and creation of encrypted credentials. You can continue to actually create the schema using the default scripts for the specified database.

---

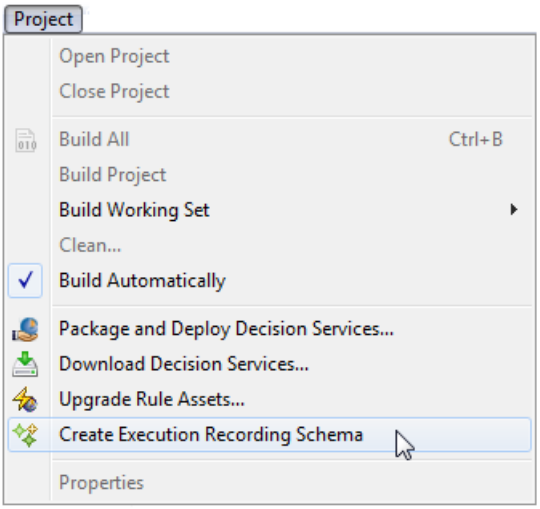
7. Click **Cancel**.

To create the schema from Studio

1. Edit Studio's `brms.properties` to add the line:

```
com.corticon.server.execution.recording.enabled=true
```

2. Restart Studio, and then choose the menu command **Project > Create Execution Recording Schema**, as shown:



3. Enter and test the connection information as described in the previous procedure.
4. Click **Test Connect** to ensure that the information achieves connection.

---

5. **Note:** If the schema exists, its tables and existing data will be deleted, and then the tables recreated.

To create the schema, click **Finish**.

6. It is a good idea to shut off the ability to recreate the schema once it has been created. To do that, edit Studio's `brms.properties` to change the line you added to:

```
com.corticon.server.execution.recording.enabled=false
```

Configuring Corticon Servers to store rule messages that they execute

On each server you must set a few properties in the **brms.properties** file to enable recording of rule executions:

1. Enable the Server to instantiate Rule Execution Recording during startup by setting this property to `true`:

   ```
   com.corticon.server.execution.recording.enabled=true
   ```

2. Once enabled, both payloads and rule messages will be emitted. You can shut off either one by changing its property setting to `false`:

   ```
   com.corticon.server.execution.recording.process.payloads=false
   com.corticon.server.execution.recording.process.rulemessages=false
   ```

3. Paste the text of the connection information then delete (or comment out) the first line.

```
### EXAMPLE:   Copy the following properties to the Server's brms.properties:

ccexecutionrecordingservice.com.corticon.database.id=com.corticon.database.id.MsSql2014
ccexecutionrecordingservice.hibernate.connection.url   \
             =jdbc:progress:sqlserver://myExecutionRecordingServer:1433; \
              databaseName=CorticonExecutionRecords
ccexecutionrecordingservice.hibernate.connection.username=061046
ccexecutionrecordingservice.hibernate.connection.password=030046016058035029061039110
```

Note: Either or both the username and password could be entered as plain text instead of encrypted values.

Turning message recording on for individual Decision Services

With the schema set up in the target database, the intent to record enabled on the server, and the database connection tested, all that needs to be done is to set the properties on each Decision Service that will participate in this feature. By default, no Decision Service will use this feature. There are two ways to enable a Decision Service to enable execution recording:

- **Execution properties in a manually edited CDD file** - Add the following line to the `options` in a `decisionservice` section of the CDD file: `<option name="PROPERTY_EXECUTION_RECORDING_SERVICE_ENABLED" value="true"`
- **CcServer API methods** - Use `setDecisionServicePropertyValue` for the `DecisionServiceName` to set the static property `PROPERTY_EXECUTION_RECORDING_SERVICE_ENABLED` and the property value `true`.

Once established on a Decision Service, you can turn off execution recording by changing the setting to `false`.

A

Service contract examples

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.

For details, see the following topics:

- Examples of XSD and WSDLS available in the Deployment Console
- Extended service contracts
- Extended datatypes

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.
```

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:8850/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:8850/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" />
<xsd:element name="FlightPlan"
type="tns:FlightPlanType" minOccurs="0" 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:8850/axis/services/Corticon" />
      </port>
  </service>
  </definitions>
```

1.1 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

1.2 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" />
```

1.3 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. All 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: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>
```

1.4 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.

1.5 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>
```

1.6 VocabularyAttributeNameTypes

Every attribute in a Corticon Vocabulary is 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.

2.1 Header

This section of the XSD is identical to the FLAT version, described in 1.1 Header on page 235.

2.2 CorticonRequestType and CorticonResponseType

This section of the XSD is identical to the FLAT version, described in 1.2 CorticonRequestType and CorticonResponseType on page 235.

2.3 WorkDocumentsType

One line in this section differs from the FLAT version (described in 1.3 WorkDocumentsType on page 236):

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

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

2.4 MessagesType

This section of the XSD is identical to the FLAT version, described in 1.4 MessagesType on page 237.

2.5 VocabularyAttributeNameTypes

This section of the XSD is the same as the FLAT version, described 1.6 VocabularyAttributeNameTypes on page 239.

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

When **Decision Service** is selected in input option **1** of Deployment Console: Service Contract Specifications, the XML Messaging Style input option **4** becomes inactive ("grayed out"). This occurs because the XML Messaging Style option at the Decision Service level, (input property **13** of Deployment Console: Decision Service Deployment Properties) becomes the governing setting.

3.1 Header

This section of the XSD is identical to the Vocabulary-level FLAT version, described in 1.1 Header on page 235.

3.2 CorticonRequestType and CorticonResponseType

This section of the XSD is identical to the Vocabulary-level FLAT version, described in 1.2 CorticonRequestType and CorticonResponseType on page 235, 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"`.

3.3 WorkDocumentsType

This section of the XSD is identical to the Vocabulary-level FLAT version, described in 2.3 WorkDocumentsType on page 239.

3.4 MessagesType

This section of the XSD is identical to the Vocabulary-level FLAT version, described in 1.4 MessagesType on page 237.

3.5 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 input option **1** of Deployment Console: Service Contract Specifications, the XML Messaging Style input option becomes inactive ("grayed out"). This occurs because the XML Messaging Style option at the Decision Service level becomes the governing setting.

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:8850/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:8850/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 minOccurs=""0"">
      <xsd:choice maxOccurs="unbounded">
      <xsd:element name="Cargo" type="tns:CargoType" />
      </xsd:choice>
```

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:8850/axis/services/Corticon" />
    </port>
  </service>
</definitions>
```

4.1 Header

This section of the XSD is identical to the Vocabulary-level FLAT version, described in 1.1 Header on page 235.

4.2 CorticonRequestType and CorticonResponseType

This section of the XSD is identical to the Decision-Service-level FLAT version, described in 1.2 CorticonRequestType and CorticonResponseType on page 235.

4.3 WorkDocumentsType

This section of the XSD is identical to the Vocabulary-level HIER version, described in 1.3 WorkDocumentsType on page 236.

4.4 MessagesType

This section of the XSD is identical to the Vocabulary-level FLAT version, described in 1.4 MessagesType on page 237.

4.5 VocabularyEntityNameType and VocabularyAttributeNameTypes

The *structure* of this section of the XSD is identical to the Vocabulary-level HIER version (described 2.5 VocabularyAttributeNameTypes on page 239). 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

5.1 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

5.2 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>
```

5.3 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>
```

5.4 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>
```

5.5 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>
```

5.6 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:8850/axis/services/Corticon" />
      </port>
    </service>
 </definitions>
```

6 Vocabulary-level WSDL, HIER XML messaging style

6.1 SOAP Envelope

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in 5.1 SOAP Envelope on page 243.

6.2 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>
```

6.3 Messages

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in 5.3 Messages on page 244.

6.4 PortType

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in 5.4 PortType on page 244.

6.5 Binding

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in 5.5 Binding on page 244.

6.6 Service

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in 5.6 Service on page 244.

7 Decision-service-level WSDL, FLAT XML messaging style

7.1 SOAP Envelope

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in 5.1 SOAP Envelope on page 243.

7.2 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>
```

7.3 Messages

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in 5.3 Messages on page 244.

7.4 PortType

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in 5.4 PortType on page 244, with the exception of the following line:

```
<portType name="DecisionServiceNameSoap">
```

7.5 Binding

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in 5.5 Binding on page 244, with the exception of the following line:

```
<binding name="DecisionServiceNameSoap" type="tns: DecisionServiceNameSoap">
```

7.6 Service

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in 5.6 Service on page 244, 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

8.1 SOAP Envelope

This section of the WSDL is identical to the Vocabulary-level FLAT version, described in 5.1 SOAP Envelope on page 243.

8.2 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>
```

8.3 Messages

This section of the WSDL is identical to the Decision Service-level FLAT version, described in 5.3 Messages on page 244.

8.4 PortType

This section of the WSDL is identical to the Decision Service-level FLAT version, described in 5.4 PortType on page 244.

8.5 Binding

This section of the WSDL is identical to the Decision Service-level FLAT version, described in 5.5 Binding on page 244.

8.6 Service

This section of the WSDL is identical to the Decision Service-level FLAT version, described in 5.6 Service on page 244.

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 your brms.properties override file. See Configuring Corticon properties and settings on page 267 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>
```

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>
```

B

Corticon API reference

Corticon APIs are exposed interfaces that enable use of Corticon in applications.

For details, see the following topics:

- Java API
- REST Management API

Java API

The Corticon APIs are documented in JavaDoc format, and are installed with Corticon Studio and Corticon Server for Java. The essential management API is described in Corticon Server API JavaDocs, as linked here from the Progress remote documentaton site.

REST Management API

Overview

The Corticon REST Management API provides several REST methods for management of Corticon Server and Decision Service deployment .

Common Request/Response types

Base64 Encoded Files

All JSON embedded files will be encoded into strings using the Base64 Content-Transfer-Encoding as described in RFC 2045 The String will have the following properties:

- The String will NOT be URL safe encoded.
- The string will be on a single line -- in other words, no line separators

The implementation of this particular encoding uses the Java SE DatatypeConvert item to convert to and from Base64 The decoding of the file is done (more or less) using this code (where `base64String` is the string containing the Base64 encoding):

```
import javax.xml.bind.DatatypeConverter;
...
String base64String = ...;
...
byte[] base64ByteArray = DatatypeConverter.parseBase64Binary(base64String);
InputStream base64FileInputStream = new ByteArrayInputStream(base64ByteArray);
```

The encoding of the file should be done using the requirements specified above. Check that the library you use performs proper encoding and decoding the string. The following example shows how to properly encode an array of bytes into a Base64 string using the JavaSE DatatypeConverter (where `byteArrayOfBinaryFile` is the byte array used to encode using Base64):

```
import javax.xml.bind.DatatypeConverter;
...
byte[] byteArrayOfBinaryFile = ...;
...
String base64String = DatatypeConverter.printBase64Binary(byteArrayOfBinaryFile);

//Note: This version will chunk the Base64 string into 76 character lines, most
implementations do not have problems with this but some will.
```

Windowed Metrics

Some of the metrics returned by various calls to the API contain windowed metrics. These metrics will be placed in an array of objects, each object will represent a particular section of time. The sections of time for each are differentiated by a "timestamp" field in the object, the value of which is a UTC count of milliseconds. If for some reason a metric is not available for a particular window, its field will not be included in the window object.

Metric Types

Both windowed and non-windowed metrics have specific types associated with them, as follows:

- `milliseconds` - Stored in JSON as a positive integer value, the value represents a countable number of milliseconds.
- `count` - Stored in JSON as a positive integer value, the value represents some countable value in the set of integers.

- `byte` - Stored in JSON as a positive integer value, the value represents a discrete number of bytes. This value is often used to show quantities of used or unused memory.
- `string` - Stored in JSON as a string, the value represents human readable text.

Error handling

Error handling is done through HTTP error codes and appropriate error objects.

Error Objects

A failed API call will place an object in the entity similar to this:

```
{
    "error"
    {   "type" :  The java type of the exception that was thrown to generate this
error,
        "parentError" : The nested error object that was the cause of thisexception
                            (This field is included only when this error has a nested
error),
        "message" : The message from this error code,
       "stackTrace" : This array contains lines of a stacktrace, each will correspond

                       to a stackframe or an exception label if there are nested
exceptions
        [...]
    }
}
```

Common error behavior

All REST urls defined have a common error behavior. An error response returned by the server consists of two parts:

1. A HTTP status code other than 200.
2. A JSON Object in the response payload containing the error property.

Common HTTP status codes

The server can return following codes:

- `200 OK` The request was processed successfully.
- `400 Bad request` An incorrect request.
- `500 Internal Server Error` The server encountered an error while processing the request.

Summary of REST methods for management of Corticon Servers and Decision Services

Method	Type	Syntax and function
API ListDecisionServices	`HTTP GET`	`<base>/decisionService/list` Returns a list of Decision Services deployed on the server. The resulting payload will be enclosed in the response's body in JSON object form.
API Deploy Decision Service	`HTTP POST`	`<base>/decisionService/deploy` Attempts to add a Decision Service to the server. Request objects will be sent as the content.

Method	Type	Syntax and function
API Undeploy Decision Service	`HTTP POST`	`<base>/decisionService/undeploy` Attempts to remove the Decision Service from the server and delete the EDS file associated with it. The request will take HTTP headers that provides the Decision Service name, and, optionally the Major and Minor version Number.
API Get Decision Service Properties	`HTTP GET`	`<base>/decisionService/getProperties` Returns the properties pertaining to the Decision Service. The request gets the properties for the Decision Service passed in its header. The request will take HTTP headers that provide the Decision Service name, and, optionally the Major and Minor version Number.
API Set Decision Service Properties	`HTTP POST`	`<base>/decisionService/setProperties` Attempts to modify the properties of a specified Decision Service.
API Ping Server	`HTTP GET`	`<base>/server/ping` Returns an object containing the current uptime of the server, which confirms that the server is reachable and running.
API Retrieve Metrics	`HTTP POST`	`<base>/server/metrics` Retrieves metrics for the server. The request can pass in a Decision Service (or a list of Decision Services as an array), and a timestamp showing when the windowed metrics will begin. If the JSON object does not contain any Decision Service, metrics are returned for all the Decision Services in the server along with the server metrics.
API Get Server Info	`HTTP GET`	`<base>/server/info` Returns information about the server. The returned object will contain information about both the server and the Decision Services deployed on the server.
API Get Server Log	`HTTP GET`	`<base>/server/log` Returns the server log entries after the specified timestamp.
API Get Server Properties	`HTTP GET`	`<base>/server/getProperties` Returns the properties pertaining to the server. The returned object will be a list of key/value pairs consisting of key: *`propertyName value: value`* with information about the property.

Method	Type	Syntax and function
API Set Server Properties	`HTTP POST`	`<base>/server/setProperties` Sets properties on the server using a JSON object. The object will contain a key/value format system: *`propertyName value: value`* intended to be set for this property.
API Set Server License	`HTTP POST`	`<base>/server/setLicense` Sets the license file for the server. This can be used to add a license in the event the current one is not usable.

API ListDecisionServices

```
<base>/decisionService/list
```

HTTP GET

Returns a list of Decision Services deployed on the server. The resulting payload is enclosed in the response's body in JSON object form. The JSON object structure is as follows:

```
{
    "decisionServices" :
    [
        {
        "name" : <Decision Service Name>,
        "majorVersion" : <Decision Service Major Version Number>,
        "minorVersion" : <Decision Service Minor Version Number>,
        },
        ...
    ]
}
```

An example of a response is:

```
{
    "decisionServices" :
    [
        {
            "name" : "OrderProcessing",
            "majorVersion" : "1",
            "minorVersion" : "10",
        },
        {
            "name" : "OrderProcessing",
            "majorVersion" : "1",
            "minorVersion" : "11",
        },
        {
            "name" : "Cargo",
            "majorVersion" : "1",
            "minorVersion" : "0",
        }
    ]
}
```

In the case where no Decision Services are deployed on the server, the payload contains an empty JSON array and the response code is set to 204:

```
{
    "decisionServices" : []
}
```

Success status codes:

- 200 OK
- 204 No Content (the list is empty)

Error status codes:

- 500 Internal Server Error

API Deploy Decision Service

```
<base>/decisionService/deploy
```

HTTP POST

Attempts to add a Decision Service to the server. Request objects are sent as the content and are formatted as follows:

Note: Some items use files encoded in Base64. See the section "Base64 Encoded Files" in the overview topic, REST Management API on page 252, for more information.

```
{
    "edsFile" : <Base64 Encoded binary of the eds file>,
    "edsFileName" : <The name of the eds file that is being uploaded, OPTIONAL>,
    "serviceName" : <The name of the Decision Service that is being sent to the server>,
    "minSize" : <The minimum server pool size DEPRECATED>,
    "maxSize" : <The maximum server pool size>,
    "msgStyle" : <The XML message style to be used for this Decision Service>,
    "dbAccessMode" : <The database access mode, OPTIONAL>,
    "dbReturnMode" : <The database access Entities Return mode, OPTIONAL>,
    "dbPropFile" : <Base64 Encoded binary of the database properties file, OPTIONAL>,
    "dbPropFileName" : <The name of the eds properties file, OPTIONAL>
}
```

Responses are formatted as follows:

```
{
    "decisionService" : <This is present if the request completed successfully>
    {
        "name" : <Name of the Decision Service that was just successfully deployed>,
        "majorVersion" : <The major version number of the Decision Service deployed>,
        "minorVersion" : <The minor version number of the Decision Service deployed>
    },
    "error" : <This object will only be present if the response was not "OK">
}
```

Success status codes:

- 200 OK

Error status codes:

- 400 Bad Request
- 500 Internal Server Error

API Undeploy Decision Service

```
<base>/decisionService/undeploy
```

HTTP POST

Attempts to remove the specified Decision Service from the server, and to delete the EDS file associated with it. The request takes HTTP headers that provide the Decision Service name, and, optionally, the Major and Minor version numbers. The format for the request headers is as shown:

```
name : <The name of the Decision Service we are trying to remove>,
majorVersion : <The major version of the Decision Service we are trying to remove.
               This field is optional but required if minorVersion is used>,
minorVersion : <The minor version of the Decision Service we are trying to remove.
               This field is optional>
```

Responses are formatted as shown:

```
{
    "error" : <This object will only be here if the response is not "OK">
}
```

Success status codes:

- 200 OK

Error status codes:

- 400 Bad Request
- 500 Internal Server Error

API Get Decision Service Properties

```
<base>/decisionService/getProperties
```

HTTP GET

Returns the properties pertaining to the Decision Service. The request gets the properties for the Decision Service passed in its header. The request will take HTTP headers that provide the Decision Service name, and, optionally the Major and Minor version Number. The headers are set as follows:

```
Name: <The name of the Decision Service, REQUIRED>
majorVersion: <The major version number of the Decision Service, OPTIONAL>
minorVersion: <The minor version number of the Decision Service, OPTIONAL>
```

The returned object is a list of key/value pairs consisting of key: `propertyName` value: `JSON object` with information about the property.

```
{
<PropertyName> :
<value>
},
...
}
```

An example of response is as shown:

```
{
    "effectiveDateStart":"",
    "dbAccessMode":"",
    "restrictInfoRuleMessages":"",
```

```
        "majorVersion":1,
        "restrictWarningRuleMessages":"",
        "dbPropFilePath":"",
        "ruleflowUri":"",
        "minorVersion":10,
        "msgStyle":"",
        "runningInBatch":false,
        "edsUri":"C:/Program Files/Eclipse/
                    eclipse-platform-4.3.1-win32-x86_64/eclipse/
                    CcServerSandbox/DoNotDelete/DecisionServices/
                    U0_1418246336581.225329/Order_v1_10.eds",
        "autoReload":true,
        "dbReturnMode":"ALL",
        "loadedFromCdd":false,
        "deploymentTimestamp":"01/15/15 3:17:57 PM",
        "maxPoolSize":10,
        "minPoolSize":1,
        "ruleflowTimestamp":"12/10/14 7:00:00 PM",
        "cddPath":"",
        "effectiveDateStop":"",
        "restrictViolationRuleMessages":"",
        "edsTimestamp":"12/31/14 4:18:56 PM"
    }
```

Success status codes:

- 200 OK

Error status codes:

- 400 Bad Request
- 500 Internal Server Error

API Set Decision Service Properties

```
<base>/decisionService/setProperties
```

HTTP POST

Attempts to modify the properties of a specified Decision Service. The request headers must have information about which Decision Service's properties to get. The Decision Service name, major version number, and minor version number are all required. The headers are set as follows:

```
   Name: <The name of the Decision Service
         whose properties you want to set, REQUIRED>
majorVersion: <The major version number of the Decision Service
              whose properties you want to set, REQUIRED>
minorVersion: <The minor version number of the Decision Service
              whose properties you want to set, REQUIRED>
```

The object must contain a key/value format system of the form key: `property name` value: `value` intended to be set for this property, as shown:

```
{
     <Property name> : <Property's intended value>,
     ...
}
```

The response contains a JSON object similar to the one in API Get Decision Service Properties on page 257. There is a list of key/value pairs representing the properties that were attempted to be set, in the form key: `property name` value: `object` representing success/failure of setting the property value, as shown:

```
{
    "errors" : [ <A list of the property field names that have errors>
    ...     ]
    <Property name> : {
        "status" : "OK", <The status field's existence signifies that the requested
                          property change succeeded; if there was an error, this field
                          is not included in the object>
        "error" : {...} <This field will only exist if the property did not get
                         successfully set for any reason; this will be a standard
                         error object>
    },
    ...
}
```

For example, if the following values were provided in a request:

```
{
    "restrictInfoRuleMessages" : true,
    "deploymentTimestamp" : "02/01/15 6:02:55 PM"
}
```

The following excerpted response would be returned with an HTTP status code of INTERNAL SERVER ERROR(500):

```
{
    "errors" ; [ "deploymentTimestamp" ],
    "restrictInfoRuleMessages":{
        "status":"OK",
    },
    "deploymentTimestamp":{
        "error":{
            "message":"Property name: \"deploymentTimestamp\" is ReadOnly",
            "type":
            "com.corticon.eclipse.rest.delegates.CcRestDelegatePropertyRequestErrorException",

            "stackTrace":[
                "com.corticon.eclipse.rest.delegates.RestDelegate
                      .setDecisionServicePropertyValue(RestDelegate.java:1436)",
                "com.corticon.eclipse.rest.delegates.RestDelegate
                      .SetDecisionServiceProperties(RestDelegate.java:943)",
                "com.corticon.eclipse.rest.CorticonSetDecisionServiceProperties

.postCorticonSetDecisionServiceProperties(CorticonSetDecisionServiceProperties.java:39)",
                "sun.reflect.NativeMethodAccessorImpl
                      .invoke0(Native Method)",
                "sun.reflect.NativeMethodAccessorImpl
                      .invoke(NativeMethodAccessorImpl.java:57)",
                "sun.reflect.DelegatingMethodAccessorImpl
                      .invoke(DelegatingMethodAccessorImpl.java:43)",
                "java.lang.reflect.Method
                      .invoke(Method.java:601)",                ...
                ...
              "java.util.concurrent.ThreadPoolExecutor$Worker.
                                  run(ThreadPoolExecutor.java:603)",
              "java.lang.Thread.run(Thread.java:722)"
            ]
        }
    }
}
```

Success status codes:

- 200 OK

Error status codes:

- 400 Bad Request
- 500 Internal Server Error

API Ping Server

```
<base>/server/info
```

HTTP GET

Returns an object containing the current uptime of the server. This call indicates whether the server is reachable and running. An example of a response is as shown:

```
{
    "systemTime" : <The current uptime of the server, in milliseconds. >}
}
```

Success status codes:

- 200 OK

Error status codes:

- 500 Internal Server Error

API Retrieve Metrics

```
<base>/server/metrics
```

HTTP POST

Retrieves metrics for the server. The request can pass in a Decision Service (or a list of Decision Services as an array), and a timestamp showing when the windowed metrics will begin. If the JSON object does not contain any Decision Service, metrics are returned for all the Decision Services in the server along with the server metrics. See Windowed Metrics for more information. The object is formatted as follows:

```
{
 "timestamp" : <timestamp for metrics in milliseconds since Unix Epoch in GMT timezone>,
 "decisionServices" : <Decision Services that you want the metrics for,
            if  this list is not provided then metrics for all Decision Services
            will be provided>
 [
    {
   "name" : <Name of Decision Service we want metrics for>,
   "majorVersion" : <Major version of Decision Service we want metrics for>,
   "minorVersion" : <Minor Version of Decision Service we want metrics for>
  },
 ...
 ]
}
```

The responses are formatted as shown:

```
{
 "decisionServices" :
 [
  {
   "name" : <Name of the Decision Service these metrics belong to>,
   "majorVersion" : <Major Version of the Decision Service these metrics belong to>,
   "minorVersion" : <Minor Version of the Decision Service these metrics belong to>,
   "totalExecutionTime" : <The total execution time, in milliseconds,
                           for this Decision Service since it was added to the server >,
   "totalNumberOfExecutions" : <Total number of executions for this Decision Service
                               since it was added to the server>,
   "window" :
   [
```

```
      {
       "timestamp" : <The timestamp of this metrics window, in milliseconds
                      since the UNIX Epoch in the GMT timezone>,
       "totalIntervalExecutionTime" : <The total execution time ,in milliseconds,
                                       for this metrics window >,
       "totalIntervalNumberOfExecutions" : <The total number of executions
                                            for this metrics window>
      },
      ...
     ]
    },
    ...
   ],
   "serverMetrics" :
   {
    "totalExecutionTime" : <UTC timer format showing the total execution time for all
                            Decision Services deployed on this server since it was created>,
    "totalNumberOfExecutions" : <Total number of executions for all
                                 Decision Services deployed on this server since it was created>,

    "window" :
    [
     {
      "timestamp" : <The timestamp of this metrics window in milliseconds
                     since the UNIX Epoch in the GMT timezone>,
      "memoryUsage" : <Memory usage for this window in bytes>,
      "totalIntervalExecutionTime" : <The total execution time, in milliseconds,
                                      for this metrics window >,
      "totalIntervalNumberOfExecutions" : <The total number of executions
                                           for this metrics window>
     },
     ...
    ]
   },
   "error" : <This object will only be added if an error was encountered
              while processing the request>
   {...}
  }
```

In the event of an error, the API attempts to return a partial answer, and adds an error object to the returned object. The error object (shown above) is added to the response object, and the HTTP response code is set accordingly.

Success status codes:

- 200 OK
- 206 Partial Content (where non-failing errors were encountered while processing the request)

Error status codes:

- 400 Bad Request
- 500 Internal Server Error

API Get Server Log

```
<base>/server/log
```

HTTP GET

Returns the server log entries after the specified timestamp. The request will be a HTTP get request with the parameters passed in the header.

The response object is formatted as follows:

```
{
    {
    "logEntries": [ <Array of log entry objects>
        {
            "timestamp": <Log entry timestamp in milliseconds>,
            "loglevel": <Log entry log level>,
            "logger": <Name of the logger that this entry was logged with>,
            "marker": <Log marker associated with this log entry>,
            "message": <Message that was logged in this entry>,
            "throwable": <OPTIONAL, throwable object associated with this (if applicable)>
        },
        ...
    ]
}
```

If the server encounters an error unrelated to the property value, then a standard error response is provided.

Success status code: 200 OK

Error status codes: 500 Internal Server Error

API Get Server Info

```
<base>/server/info
```

HTTP GET

Returns information about the server. The returned object contains information about both the server and the Decision Services deployed on that server. Both the server and Decision Services have a window and a general section. Both sections provide information on their metrics. Both the window and general fields are an array of objects, and each object consists of a type and a name -- these correspond to the type and name of a metrics value returned in the Metrics REST API call. For more information about the type field, refer to the Metric Types section. The objects representing the metrics are shown here in the order they would show in the metrics call. The resulting payload is enclosed in the response's body in JSON object form. The JSON object structure is as follows:

```
{
"metricTypes": {
   "decisionService": {
      "window": [
      {
    "name": <Name of the general info field>,
    "type": <Field's type as a description, (see Metric Types)>,
    "dataType" : <Field's storage type, what type it should be stored as>,
    "aggregateTypes" : [ <Array of aggregations that can be done on this datatype,
                         this can be AVERAGE, MIN, MAX, or TOTAL>
       },
       ...
    ],
      "general": [
      {
    "name": <Name of the general info field>,
    "type": <Field's type as a description, (see Metric Types)>,
    "dataType" : <Field's storage type, what type it should be stored as>,
    "aggregateTypes" : [ <Array of aggregations that can be done on this datatype,
                         this can be AVERAGE, MIN, MAX, or TOTAL>
       },
       ...
    ]
   },
   "server": {
      "window": [
      {
    "name": <Name of the general info field>,
    "type": <Field's type as a description, (see Metric Types)>,
```

```
    "dataType" : <Field's storage type, what type it should be stored as>,
    "aggregateTypes" : [ <Array of aggregations that can be done on this datatype,
                         this can be AVERAGE, MIN, MAX, or TOTAL>
       },
       ...
    ],
      "general": [
      {
    "name": <Name of the general info field>,
    "type": <Field's type as a description, (see Metric Types)>,
    "dataType" : <Field's storage type, what type it should be stored as>,
    "aggregateTypes" : [ <Array of aggregations that can be done on this datatype,
                         this can be AVERAGE, MIN, MAX, or TOTAL>
       },
       ...
     ]
   }
 },
"windowSize": <The size, in milliseconds, of each interval window>
}
```

An example of a response is:

```
{
    "metricTypes": {
        "decisionService": {
            "window": [
                {
                    "name": "totalIntervalExecutionTime",
                    "type": "milliseconds"
                },
                {
                    "name": "totalIntervalNumberOfExecutions",
                    "type": "count"
                }
            ],
            "general": [
                {
                    "name": "totalExecutionTime",
                    "type": "milliseconds"
                },
                {
                    "name": "totalNumberOfExecutions",
                    "type": "count"
                }
            ]
        },
        "server": {
            "window": [
                {
                    "name": "memoryUsage",
                    "type": "bytes"
                },
                {
                    "name": "totalIntervalExecutionTime",
                    "type": "milliseconds"
                },
                {
                    "name": "totalIntervalNumberOfExecutions",
                    "type": "count"
                }
            ],
            "general": [
                {
                    "name": "totalExecutionTime",
                    "type": "milliseconds"
                },
                {
                    "name": "totalNumberOfExecutions",
                    "type": "count"
```

```
                    }
                ]
            }
        },
        "windowSize": 10000
    }
```

Success status codes:

- 200 OK

Error status codes:

- 500 Internal Server Error

API Get Server Properties

```
<base>/server/getProperties
```

HTTP GET

Returns the properties that pertain to the server. The returned object is a list of key/value pairs consisting of: `"PropertyName":"PropertyValue"` where value is the current value of the property.

```
{
    <PropertyName> : <Property Value>
}
```

An example of a response is:

```
{
    "buildNumber":"Development",
    "serverExecutionTimesIntervalTime":"10000",
    "fullVersionNumber":"Version: 5.6.0.0",
    "serverDiagnosticWaitTime":"",
    "serviceReleaseNumber":"0.0",
    "versionNumber":"5.6"
}
```

Success status codes:

- 200 OK

Error status codes:

- 500 Internal Server Error

API Set Server Properties

```
<base>/server/setProperties
```

HTTP POST

Sets properties on the server using a JSON object. The object contains a key/value format system of the form key: `property name` value: `value` intended to be set for this property.

```
{
    <Property name> : <Property's intended value>,
    ...
}
```

An example of a request is:

```
{
    "serverExecutionTimesIntervalTime" : 10000
}
```

The response contains a JSON object similar to what returns in `getProperties`. There is a list of key/value pairs representing the properties that were attempted to be set. The pairs are key: `property name` value: `object` representing success/failure of setting the property value.

```
{
    "errors" : [ <A list of the property field names that have errors>
     ...
     ]
    <Property name> : {
        "error" : {...} <This field will only exist if the property did not get
successfully set for any reason, this will be a standard error object as defined in the
 section "error objects".>
    },
    ...
}
```

For example, if the following object was sent:

```
{
    "serverExecutionTimesIntervalTime" : 10000,
    "versionNumber" : 1.1
}
```

You would get the following response woth an HTTP status code of BAD REQUEST (400):

```
{
    "errors" : [ "versionNumber" ],
    "serverExecutionTimesIntervalTime":{},
    "versionNumber":{
        "error":{
            "message":"Property: \"versionNumber\" is read only, value not applied",

"type":"com.corticon.eclipse.rest.delegates.CcRestDelegatePropertyRequestErrorException",
            "stackTrace":[
                "com.corticon.eclipse.rest.delegates.RestDelegate.
                                setServerPropertyValue(RestDelegate.java:1539)",
                "com.corticon.eclipse.rest.delegates.RestDelegate.
                                setServerProperties(RestDelegate.java:1071)",
                ...
                ...
                "java.util.concurrent.ThreadPoolExecutor$Worker.
                                run(ThreadPoolExecutor.java:603)",
                "java.lang.Thread.run(Thread.java:722)"
            ]
        }
    }
}
```

Success status code: 200 OK

Error status codes: 400 Bad Request

API Set Server License

```
<base>/server/setLicense
```

HTTP POST

Sets the license file for the server. This can be used to add a license in the event the current one is not usable. The format for the request JSON object is as follows:

```
{
    "licenseFile" : <The license file encoded into a base64 string>,
    "licenseFileName" : <Optional, The name of the license file>
}
```

In the event of an error a standard error object will be included in the response. If the license file was valid and set successfully an empty object is returned:

```
{}
```

Success status code: 200 OK

Error status codes:

- 500 Internal Server Error
- 400 Bad Request

C

Configuring Corticon properties and settings

Corticon installs groups of properties that specify the property names and default values of most user-configurable behaviors in Corticon Studio and Corticon Servers. When you launch Studio or Server, the groups of default property *name=value* pairs are loaded in the following order:

Property Groups	Properties
Common properties	Modifies the behavior of elements common to both the Corticon Studio and Corticon Server .
Studio properties	Controls behaviors of specific Corticon Studio functions.
Server properties	Controls behaviors of specific Corticon Server functions.
Deployment properties	Controls behaviors of the Corticon Deployment Console functions.
`CcOem.properties`	Reserved for licensed OEM customers to override vendor-specific properties.
`CcDebug.properties`	Reserved for internal use.

These properties pages are not intended for user access. Instead, the file `brms.properties`, installed by every product at the root of `[CORTICON_WORK_DIR]`, enables you to add your override settings to be applied after the default settings have been loaded.

Note: A running instance of Corticon Server can modify certain properties through API method calls, as discussed in the Administrative API section. While settings in `brms.properties` persist across Corticon Server sessions, changes applied through APIs only remain in effect for that Corticon Server session. When Corticon Server starts a new session, it will look to the default settings and the override properties file.

Properties used by Corticon Studio

Corticon Studio uses mostly the Common and Studio properties. See those topics for information on the possible settings and values that you might want to add to your override properties file. For more information on the Studio file and common overrides, see *"Applying logging and override properties to Corticon Studio and its built-in Server" in the Rule Modeling Guide*.

Properties used by Corticon Servers

Corticon Servers use mostly the Common and Server properties. See those topics for information on the possible settings and values that you might want to add to your override properties file for deployed servers.

Properties used by Deployment Console

The Deployment Console, installed with each of the Corticon Servers, uses mostly the Common and Deployment properties. See those topics for information on the possible settings and values that you might want to add to your override properties file for deployment consoles.

Note: Preferred technique for overriding default properties - In earlier releases, updates were made to the default properties in the `CcConfig.jar`. That practice is now discouraged because it prevents reversion to initial values when you are trying to resolve problems. Use the `brms.properties` file installed at the work directory root, or -- for Studio -- the location specified in Eclipse preferences. For backward compatibility, all previous locations of property settings are checked, and will continue to be supported; however, if you use those classic techniques, the `brms.properties` at the work directory root will trump all others.

For details, see the following topics:

- Using the override file, brms.properties
- Setting override properties in the brms.properties file
- Common properties
- Corticon Studio properties
- Corticon Server properties
- Corticon Deployment Console properties

Using the override file, brms.properties

The override file, `brms.properties`, lets you specify properties you want to modify and your preferred value without the mechanics of maintaining a file in a JAR, as well as ensuring that you can revert to the original behavior by just removing, commenting out, or clearing your custom properties file.

Note: When you set certain configuration properties in the Web Console, or in corresponding API methods, they are persisted to `ServerState.xml` so that they take effect each time Corticon Server is started. These settings apply AFTER your override properties file is loaded. It is recommended that you choose this override file for settings as you approach production so that you have just this last-loaded file of consistent, user-modifiable settings. The settings where this applies are the ones in the Web Console topics *"Configure Rules Server"* in the *Deploying Web Service with Java guide*, and the results of searching for `ICcServer.` instances that indicate they are overrides in the *"Configuring Corticon properties and settings" appendix of the Integration and Deployment Guide.*

For the changes to take effect, restart Corticon Studio and Servers after changing override properties. The complete set of properties are described in detail in the following topics including the default value that is set.

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.

Setting override properties in the brms.properties file

The file `brms.properties` installed at the work directory root lists properties that Studio and Server users routinely want to modify. The installed file lists each of these properties with a set of comments and then shows the commented default name=value pair.

To specify a preferred value, edit the file, remove the `#` from the beginning of a property's line, and then add your preferred value after the equals sign. For example, to change interval of diagnostic readings from five minutes to two minutes, locate the line:

```
#com.corticon.server.DiagnosticWaitTime=300000
```

and then change it to

```
com.corticon.server.DiagnosticWaitTime=120000
```

When you save the edited file, and the restart the Server, your changes are in effect.

The content of the installed `brms.properties` file is as follows:

```
##########################################################################################################
#  Log settings
#
#  logpath       - The directory where logs are written. Default value is
%CORTICON_WORK_DIR%/logs.
#                  Note: Use forward slashes as path separator.  Example: c:/Users/me/logs
#  loglevel can be:
#     OFF    - Turn off all logging
#     ERROR  - Log only errors
#     WARN   - Log all errors and warnings
#     INFO   - Log all info, warnings and errors (Default value)
#     DEBUG  - Log all debug infomation and all messages applicable to INFO level
#     TRACE  - Equivalent to DEBUG with some tracing logs
#     ALL    - Highest level of detail
#  logDailyRollover      - Specifies whether to rollover the logs on a daily basis. Default
value is true.
#  logRolloverMaxHistory - Specifies the number of rollover logs to keep. Default value is
5.
#  com.corticon.server.execution.logPerDS - This property enables the sifting of the logs
into execution log files specific
```

```
#                                          to each Decision Service. Default value is false.
#  logFiltersAccept  - When the log level is set to INFO or higher, this property lists
accepted logging items.
#                       Possible filter values:
DIAGNOSTIC,RULETRACE,TIMING,INVOCATION,VIOLATION,INTERNAL,SYSTEM
#                  Default accepted filter value list: DIAGNOSTIC,SYSTEM
#
#  Note: The loglevel and logpath can be changed using following methods, which will override
 this setting.
#  - ICcServer.setLogLevel(String)
#  - ICcServer.setLogPath(String)
##################################################################################################
#logpath=%CORTICON_WORK_DIR%/logs
#loglevel=INFO
#logDailyRollover=true
#logRolloverMaxHistory=5
#com.corticon.server.execution.logPerDS=false
#logFiltersAccept=DIAGNOSTIC,SYSTEM
##################################################################################################
#  Hibernate EDC log settings
#  See hibernate documentation for more info
##################################################################################################
#org.hibernate.SQL=DEBUG
#org.hibernate.cache=DEBUG
##################################################################################################
#  Option that will restrict certain types of Rule Messages from being posted to the
#  output of an execution.  There are 3 different properties to allow the user to
#  select exactly what is returned to them from the execution.
#
#  Default is false (for all three properties)
##################################################################################################
#com.corticon.server.restrict.rulemessages.info=false
#com.corticon.server.restrict.rulemessages.warning=false
#com.corticon.server.restrict.rulemessages.violation=false

##################################################################################################
#  Option to relax the enforcement of Custom Data Type Constraints
#  If set to true a CDT violation will post a warning message and execution will continue
#  If set to false a CDT violation will cause an exception to be thrown halting execution
#
#  Default is false
##################################################################################################
#com.corticon.vocabulary.cdt.relaxEnforcement=false

##################################################################################################
#  Option to prepend rule metadata to the business rule statement text
#  If set to true, the Rulesheet (if part of a Ruleflow) and rule ID will be prepended
#  to all business rule statements at deployment/compile time.
#  If set to false, no change is made to the rule statements
#
#  Default is false
##################################################################################################
#com.corticon.reactor.rulestatement.metadata=false


##################################################################################################
# CORTICON SERVER SPECIFIC PROPERTIES
##################################################################################################
#  Determines whether the Dynamic Update Monitor Service should be started automatically
#  when the Server is initialized.
#
#  Note: The maintenance service can be shutdown and restarted using, which will override
#  this setting:
#  - ICcServer.stopDynamicaUpdateMonitoringService()
#  - ICcServer.startDynamicUpdateMonitoringService()
#
#  Default is true (Start the Update Monitor Service)
##################################################################################################
#com.corticon.ccserver.dynamicupdatemonitor.autoactivate=true

##################################################################################################
```

```
#  Intervals of time (ms) at which the Server checks for:
#    1.  Changes in any cdd loaded to the Server by a loadFromCdd or loadFromCddDir call
#    2.  Changes in any cdd file including new cdds within the directory of cdds from a
#        loadFromCddDir call
#    3.  Changes in any of the Decision Services (.eds files) loaded to the server.
#        This is done via a timestamp check.
#
#    If any changes as described above are detected, the Server's state is dynamically
#    updated to reflect the changes.  The "maintenance thread" that checks for these
#    changes at the specified intervals can be shutdown and restarted using:
#    - ICcServer.stopDynamicaUpdateMonitoringService()
#    - ICcServer.startDynamicUpdateMonitoringService()
#
#  Default is 30 secs (30,000 ms)
##################################################################################################
#com.corticon.ccserver.dynamicUpdateMonitoringService.serviceIntervals=30000

##################################################################################################
#  Option to automatically start and configure the server diagnostics thread
#  when an ICcServer is created in the CcServerFactory
#
#  Default is true
##################################################################################################
#com.corticon.server.startDiagnosticThread=true

##################################################################################################
#  Wait time of the Server Diagnostic Monitor
#  The Diagnostic Service will post a log message every x milliseconds
#  Default is 5 minutes (30,000 ms)
##################################################################################################
#com.corticon.server.DiagnosticWaitTime=30000

##################################################################################################
#  This is the configuration for using the Execution Recording Service to write execution
information to a
#  database.
#
#  com.corticon.server.execution.recording.enabled
#  If enabled, the CcServer will instantiate the Execution Recording Service during startup.
#
#  Default value is false
#
#  com.corticon.server.execution.recording.process.payloads
#  If the Execution Recording Service is turned on, the user can specify whether the Input
and Output payloads
#  should be written to the CC_PAYLOADS database table.
#
#  Default value is true
#
#  com.corticon.server.execution.recording.process.rulemessages
#  If the Execution Recording Service is turned on, the user can specify whether the
CcRuleMessage
#  should be written to the CC_RULEMESSAGES database table.
#
#  Default value is true
##################################################################################################
#com.corticon.server.execution.recording.enabled=false
#com.corticon.server.execution.recording.process.payloads=true
#com.corticon.server.execution.recording.process.rulemessages=true

##################################################################################################
#  These properties are used to setup the default Execution Recording Service.
#
#  ccexecutionrecordingservice.com.corticon.database.id
#  The Database Id of the Database that the service will use
#
#  Database Id                                      | Database Name
#  ==================================================================================
#  com.corticon.database.id.Oracle12c               | Oracle Database 12c
#  com.corticon.database.id.Oracle10g               | Oracle Database 10g
#  com.corticon.database.id.Oracle                  | Oracle Database 11g
```

```
#  com.corticon.database.id.DB210.5              | IBM DB2 10.5
#  com.corticon.database.id.DB2                  | IBM DB2 9.5
#  com.corticon.database.id.MsSql                | Microsoft SQL Server 2008
#  com.corticon.database.id.MsSql2012            | Microsoft SQL Server 2012
#  com.corticon.database.id.MsSql2014            | Microsoft SQL Server 2014
#  com.corticon.database.id.MySQL                | MySQL 5.6 Database
#  com.corticon.database.id.PostgreSQL           | PostgreSQL 9.4 Database
#  com.corticon.database.id.OE11.5               | Progress OpenEdge 11.5
#  com.corticon.database.id.OE11.4               | Progress OpenEdge 11.4
#  com.corticon.database.id.OE11.3              | Progress OpenEdge 11.3
#  com.corticon.database.id.OE10.2              | Progress OpenEdge 10.2
#
#  ccexecutionrecordingservice.hibernate.connection.username
#  The username to connect to the Database.
#
#  ccexecutionrecordingservice.hibernate.connection.password
#  The password to connect to the Database
#
#  ccexecutionrecordingservice.hibernate.connection.url
#  The URL to connect to the Database
##################################################################################################
#ccexecutionrecordingservice.com.corticon.database.id=
#ccexecutionrecordingservice.hibernate.connection.username=
#ccexecutionrecordingservice.hibernate.connection.password=
#ccexecutionrecordingservice.hibernate.connection.url=
```

You can add lines for other properties and values that you want to override. See the following sections for details. For example, to express a preference for decimal values displayed and rounded to two places instead of the six places preset for this property, add the following line to the `brms.properties` file -- it does not matter where in the file as long as it is on a separate line:

```
decimalscale=2
```

Common properties

The following properties are used by both Corticon Studio and Corticon Server.

Note: Common properties are stored as a set of defaults in the `CcCommon.properties` file that is packaged in the `CcConfig.jar`. Each property's notes, options, and default value are listed in this section. You should always set override values in `brms.properties` file located at your work directory root --or, in Studio, the preferred location specified in **Preferences**.

Important - Some properties in the `CcCommon.properties` file that were used to control logging -- `logverbosity, com.corticon.logging.thirdparty.logger.class` -- are no longer used. See the topic *"Changing logging configuration" in the Using Corticon Server logs section of Integration and Deployment Guide* for more information.

Log settings:

- `logpath` - The directory where logs are written. Default value is `%CORTICON_WORK_DIR%/logs`. Note: Use forward slashes as path separator, as in `c:/Users/me/logs`
- `loglevel` - The depth of detail in standard logging. Value can be one of:
  - `OFF` - Turn off all logging
  - `ERROR` - Log only errors

  - `WARN` - Log all errors and warnings
  - `INFO` - Log all info, warnings and errors (Default value). Include all entry types in the `logFiltersAccept` list.
  - `DEBUG` - Log all debug infomation and all messages applicable to `INFO` level. Includes all entry types in the `logFiltersAccept` list.
  - `TRACE` - Equivalent to DEBUG with some tracing logs. Includes all entry types in the `logFiltersAccept` list.
  - `ALL` - Maximum detail. Includes all entry types in the `logFiltersAccept` list.

- `logDailyRollover` - Specifies whether to rollover the logs on a daily basis. Default value is `true`.
- `logRolloverMaxHistory` - Specifies the number of rollover logs to keep. Default value is `5`.
- `logFiltersAccept` - When the log level is set to INFO or higher, this property allows logging of items that are listed. The filter values that can be in the comma-separated list are:
  - `RULETRACE` - Records performance statistics on rules
  - `DIAGNOSTIC` - Records service performance diagnostics at a defined interval (default is 30 seconds).
  - `TIMING` - Records timing events.
  - `INVOCATION` - Records invocation events.
  - `VIOLATION` - Records exceptions.
  - `INTERNAL` - Records internal debug events
  - `SYSTEM` - Records low-level errors and fatal events.

  The default `logFiltersAccept` setting is: DIAGNOSTIC,SYSTEM

The `loglevel` and `logpath` can be changed using following methods, which will override this setting.

- `ICcServer.setLogLevel(String)`
- `ICcServer.setLogPath(String)`

```
logpath=%CORTICON_WORK_DIR%/logs
loglevel=INFO
logDailyRollover=true
logRolloverMaxHistory=5
logFiltersAccept=DIAGNOSTIC,SYSTEM
```

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).

```
decimalscale= 6
```

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`.

```
enableNewOrModified= false
```

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).

The lenientDateTimeFormat sub-property does the following:

- When `false`, forces all dateTime values to Zulu format which is the XML standard
- When `true`, allow any dateTime format supported by Java to be used in the payload

Default for ensureComplianceWithServiceContract is true (sort)

Default for lenientDateTimeFormat is false

```
com.corticon.ccserver.ensureComplianceWithServiceContract=true
com.corticon.ccserver.ensureComplianceWithServiceContract.lenientDateTimeFormat=false
```

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.jdom.translation.textmode= NORMALIZE
```

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: ixia_locid="138">

- 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.load= true
```

Control of cross-asset validation behavior. The default setting, `false`, causes cross-asset validation to occur immediately whenever any change is made. Consider an example where a Vocabulary Editor and three associated Rulesheet Editors are open simultaneously. If this setting is false, a Vocabulary update will cause the Rulesheets to revalidate themselves in real time. This dynamic validation provides instant feedback but carries a performance cost.

The alternative setting, `true`, causes cross-asset validation to be deferred until the associated editor is activated. In the prior example, a Vocabulary update will trigger only Vocabulary validation rules. Rulesheet Editors will not automatically revalidate themselves until they are activated. This setting can improve performance at the expense of immediate feedback.

Default value is `false`.

```
com.corticon.resource.validate.on.activation=false
```

Control of vocabulary validation behavior on delete. The default value true ensures that a vocabulary will be in a properly identified state after a delete operator is conducted. This may cause performance issues with very large vocabularies. Default value is `true`.

```
com.corticon.vocabulary.validate.on.delete=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.localization.setsupportedlocales.swap= 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.vocabulary.cache= true
```

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.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
```

Used in XML translation. Based on this setting, extra processing ensures that the incoming document's Enities, Attributes, and Associations match the namespaces defined in the Vocabulary. If the Vocabulary Entity or Attribute does not have an explicitly set namespace value, the Element's namespace must be the same as the namespace for the WorkDocuments Element. If the namespaces don't match, the Entity, Attribute, or Association will not be read into memory during execution. Also, if new Enitities, Attributes, or Associations are added to the XML because of rules, then explicitly set Vocabulary value will be used, otherwise the WorkDocument's namespace will be used. Default value is `false`.

```
com.corticon.xml.namespace.ignore=false
```

Specifies whether the XSD and WSDL generators adds the usage attribute on the CorticonRequest and CorticonResponse definition. The "usage" is deprecated, and no longer used. However, to be backwards compatible with customers that have already generated proxies from older Schemas or WSDLs, the user now has the option to add the usage to the generated .xsd or .wsdl. Default value is false.

```
com.corticon.servicecontracts.append.usagelabel=false
```

Date/time formats in CcCommon.properties

Corticon Studio's `DateTime` datatype uses both date and time data. The `Date` datatype handles only date information, and the `Time` datatype handles only time information.

The Corticon XML Translator will maintain the consistency of `DateTime`, `Date`, and `Time` values from input to output documents as long as the masks that are used are contained in the lists.

Note: Date/time formats are Common properties that are stored as a set of defaults in the `CcCommon.properties` file that is packaged in the `CcConfig.jar`. Each property's notes, options, and default value are listed in this section. You should always set override values in `brms.properties` file located at your work directory root --or, in Studio, the preferred location specified in **Preferences**.

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
```

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.crml.OclDate.defaultDateForTimeValues=1970-01-01
com.corticon.crml.OclDate.defaultDateFormatForTimeValues= yyyy-MM-dd
```

When `com.corticon.crml.OclDate.locale=true`, it will override the default datetime mask and use the locale mask as the date style type defined by `com.corticon.crml.OclDate.datetype` and the time style type defined by `com.corticon.crml.OclDate.typetype`.

```
com.corticon.crml.OclDate.locale=false
com.corticon.crml.OclDate.datetype=3
com.corticon.crml.OclDate.timetype=2
```

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.permissive =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).

```
com.corticon.crml.OclDate.maskliterals =true
```

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}

Symbol	Meaning	Presentation	Patterns
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}
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

Note: Property settings you list in your `brms.properties` do not *append* to an existing list, they *replace* the default values. 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.

Corticon Studio properties

The following properties are used by Corticon Studio.

Note: Studio properties are stored as a set of defaults in the `CcStudio.properties` file that is packaged in the `CcConfig.jar`. Each property's notes, options, and default value are listed in this section. You should always set override values in `brms.properties` file located at your work directory root -- or, in Studio, the preferred location specified in **Preferences**.

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.undoredo.stack.size=3
```

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.insertrowstoend=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`.

```
com.corticon.designer.corticon.insertcolumnstoend=10
```

Determines the Namespace prefix to use when exporting Tester Data to XML/SOAP documents. There is no default value.

```
com.corticon.tester.namespace.prefix=
```

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.encoding.standard=UTF-8
```

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.

```
com.corticon.designer.valuesets.compressLargeSetsUsingNot=false
```

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`.

```
xmi.import.ecore.readonlyflag.default=true
```

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.eclipse.platform.io.useuuids=false
```

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.tester.ccserver.maxdecisionservices=15
```

Specifies whether the logic used to localize rule expressions will be invoked (true) or not (false). Default value is `true`.

```
com.corticon.localize.expressions=true
```

Specifies whether test tree node association text will display domain and target entity type information. Default value is `true`.

```
com.corticon.tester.associations.includedomainandtype=true
```

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.studio.migration.datesubtype.default=Full DateTime
```

Specifies the number of visible rows in a Drop Down Combo Box. Default value is `10`.

```
com.corticon.eclipse.ui.dropdowns.visiblerows=10
```

Sets the Studio Test's XML messaging style:

- `Hier` (hierarchical)
- `Flat`
- `Autodetect`

Default value is `Hier`.

```
com.corticon.designer.tested.xmlmessagingstyle=Hier
```

These are the messaging styles selectable in the Deployment Descriptor file, described in Functions of the Deployment Console tool on page 48.

Set the font type and size used by the Graphic Visualizer. Default values are `arial.ttc` and `10`, respectively.

```
com.corticon.crml.CrmlGraphVisualizer.fontname=msgothic.ttc
com.corticon.crml.CrmlGraphVisualizer.fontsize=10
```

Specifies whether columns added via the Completeness Checker will be automatically sized based on the data in the columns. Default value is `true`.

```
com.corticon.eclipse.ui.completeness.check.autosize=true
```

Specifies how the Rule Messages will be displayed in the Tester after execution. based on the data in the columns. Options are ExecutionOrder, Severity, and Entity. Default value is `ExecutionOrder`.

```
com.corticon.tester.result.messages.sorting=ExecutionOrder
```

Specifies the data format the Tester Input Tree will be converted to and sent for execution. Possible values are `XML` and `JSON`. Default value is `XML`.

```
com.corticon.tester.ccserver.execute.format=XML
```

Specifies whether String attribute values should be trimmed in the Tester Expected tree. When set to `false`, suppresses trimming of leading and trailing whitespaces..

Default value is `true`.

```
com.corticon.tester.trimstringvalues=true
```

Corticon Server properties

The following properties are used by Corticon Servers.

Note: Server properties are stored as a set of defaults in the `CcServer.properties` file that is packaged in the `CcConfig.jar`. Each property's notes, options, and default value are listed in this section. You should always set override values in the `brms.properties` file located at your work directory root -- or, in Studio, the preferred location specified in **Preferences**.

Important - The logging property `com.corticon.server.execution.logperthread` is no longer used. See the topic *"Changing logging configuration" in the Using Corticon Server logs section of Integration and Deployment Guide* for more information.

Enables sifting of the logs into execution log files specific to each Decision Service. Default value is `false`.

```
com.corticon.server.execution.logPerDS=false
```

Settings that restrict each of the three types of Rule Messages (info, warning, and violation) from being posted to the output of an execution.

Note: When logs are generated for individual Decision Service versions, these properties are set as Execution Properties on each Decision Service version through the API .

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
```

Setting that restricts the CorticonResponse to contain only RuleMessages. Default value is `false`.

```
com.corticon.server.restrict.response.rulemessages=false
```

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. Use forward slashes as path separator. Example: `C:/Users/{username}/Progress/CorticonWork/SER/CcServerSandbox`. Default value is `%CORTICON_WORK_DIR%/%CORTICON_SETTING%/CcServerSandbox`

```
com.corticon.ccserver.sandboxDir=%CORTICON_WORK_DIR%/%CORTICON_SETTING%/CcServerSandbox
```

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 by checking the timestamp.

If any changes are detected, Corticon Server's state is dynamically updated to reflect the changes.

The maintenance service can be shutdown and restarted using:

- `ICcServer.stopDynamicaUpdateMonitoringService()`
- `ICcServer.startDynamicUpdateMonitoringService()`

Default for serviceIntervals is 30 secs (`30000` ms)

```
com.corticon.ccserver.serviceIntervals=30000
```

Determines whether the Dynamic Update Monitor Service should be started automatically when Corticon Server is initialized.

The maintenance service can be shutdown and restarted using:

- `ICcServer.stopDynamicaUpdateMonitoringService()`
- `ICcServer.startDynamicUpdateMonitoringService()`

Default is `true` (Starts the Update Monitor Service automatically).

```
com.corticon.ccserver.dynamicupdatemonitor.autoactivate=true
```

Determines whether to display Corticon Server messages to `System.out`, reflecting the activities in the maintenance thread. This is primarily a debugging property to be used under instructions from Progress technical support. Default is `false` (no messages).

```
com.corticon.ccserver.servermessages=false
```

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.maxloops=100
```

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.rulebuilder.exception=raise
```

Specifies the location of the JRE that will be used by the Corticon Server to compile the Ruleflows 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.ccserver.compiler.javahome.location=
```

Specifies 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.cdolistener.collectionmapping=java.util.Vector
com.corticon.cdolistener.listmapping=java.util.ArrayList
com.corticon.cdolistener.setmapping=java.util.HashSet
```

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.bos=true
```

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 `true`.

```
com.corticon.server.compile.classpath.include.alljarsunderccserver=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.engine.CheckForAssociationDuplicates=true
```

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 `false`.

```
com.corticon.reactor.rulebuilder.UseLoopContainerStrategy=false
```

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.BuildWaitTime=300000
```

Properties related to Decision Service/Version level monitoring.

The performance monitoring service can also be shutdown and restarted using the following methods, which will override this setting.

- `ICcServer.stopServerPerformanceMonitoringService()`
- `ICcServer.startServerPerformanceMonitoringService()`

`com.corticon.server.monitoring.decisionservice.record.times` specifies whether the Server will auto-start recording time measurements.

Default value is `true`

```
com.corticon.server.monitoring.decisionservice.record.times=true
```

`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.times.total=100
```

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.

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 `true`.

```
com.corticon.server.monitoring.decisionservice.record.data=true
```

`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.registration.delimiter=;
```

`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.registration.delimiter=,
```

`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.record.data.bucket.results.delimiter=:
```

```
com.corticon.server.monitoring.decisionservice.trackingattribute.number
    =ds name;ds major version number,ds minor version number;tracking
attribute;attribute type;bucket definitions
```

where:

- *ds name* is the name of the Decision Service to be monitored
- *ds major version number* is the Major Version number of the Decision Service to be monitored
- *ds minor version number* is the Minor Version number of the Decision Service to be monitored
- *tracking attribute* is the fully qualified path to the attribute as defined in Vocabulary
- *bucket definitions* is the definitions of each bucket in which <tracking attribute> will be evaluated. This is an options field. If null, the Server will keep track of all unique values. Bucket definitions can be distinct values or range values. Range values only apply to <attribute type> Date, Decimal, and Integer.

These values are delineated using values from `com.corticon.server.monitoring.decisionservice.base.registration.delimiter`.

Bucket definitions are delineated using values from `com.corticon.server.monitoring.decisionservice.bucket.registration.delimiter`

For example:

```
com.corticon.server.monitoring.decisionservice.trackingattribute.1
      =AllocateTrade;1;1;Trade.transaction.dPrice;Decimal
```

and

```
com.corticon.server.monitoring.decisionservice.trackingattribute.1
      =AllocateTrade;1;1;Trade.transaction.dPrice;Decimal;<100,[100..200), >= 200
```

Properties that control monitoring execution times of Decision Service/Versions over defined interval periods.

The time interval monitoring service can be shutdown and restarted using the following API methods, which will override this setting.

- `ICcServer.stopServerExecutionTimesIntervalService()`
- `ICcServer.startServerExecutionTimesIntervalService()`

`com.corticon.server.monitoring.decisionservice.interval.record.times` specifies whether Corticon Server will auto-start the recording of time interval measurements. Default value is `true`

```
com.corticon.server.monitoring.decisionservice.interval.record.times=true
```

`com.corticon.server.monitoring.decisionservice.interval.record.sleep` indicates the number of millisconds that the interval results will be recorded. Default value is `10000` (10 seconds)

```
com.corticon.server.monitoring.decisionservice.interval.record.sleep=10000
```

`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.monitoring.decisionservice.interval.record.total=50
```

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.server.soap.collection.results.delimiter=;
```

Specify if and from where `.cdd` files get auto-loaded into Corticon Server when it starts up.

This property can be 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.

```
com.corticon.ccserver.autoloaddir.enable=true
com.corticon.ccserver.autoloaddir=%CORTICON_WORK_DIR%/cdd
```

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.appendservertimes=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.ccserver.cloneAssociationHashSets=false
```

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.load=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.server.serverstate.persistchanges=true
```

By default, attributes are checked for null values to prevent invalid operator calls. 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.

```
com.corticon.reactor.rulebuilder.DisableNullCheckingOnExtensions=false
```

Determines whether all href references will be removed from the CorticonResponse in a post process step. This will only occur if the CorticonRequest contains the messageType attribute that tells the CcServer to execute in HIER mode. Default is `true`

```
com.corticon.server.execution.post.removehrefs.hier=false
```

Used by the Maintenance Thread to clean up temporary files inside the CcServerSandbox. Default is 10 minutes (`600000` ms)

```
com.corticon.server.tempfile.cleanup.interval=600000
```

Used by the Ruleset Compiler while under .NET. This property will allow the user to use a standard java.exe program to call into the ANT process to compile or to use ikvm.exe with the help of IKVM's OpenJDK. Default is `ikvm`

```
com.corticon.server.compile.dotnet.application=ikvm
```

Compile option: Add the Rule Assets to the compiled EDS file. By having the Rule Assets inside the EDS file, new versions of the deployed Decision Service can be made on the Corticon Server. The Rule Assets will be encrypted. Including the Rule Assets in the EDS file will increase the EDS file significantly. Default is `false`

```
com.corticon.server.compile.add.ruleassets=false
```

Compile option: Add the Rule Asset's Report to the compiled EDS file. By having the Report inside the EDS file, any user can get the report for a deployed Decision Service through an in-process or a SOAP call to the Corticon Server. Including the Report in the EDS file will increase the EDS file significantly. Default is `true`

```
com.corticon.server.compile.add.report=true
```

Compile option: Add the Rule Asset's WSDL to the compiled EDS file. By having the WSDL inside the EDS file, any user can get the WSDL for a deployed Decision Service through an in-process or a SOAP call to the Corticon Server. Including the WSDL in the EDS file will increase the EDS file significantly. Default is `true`

```
com.corticon.server.compile.add.wsdl=true
```

Compile option: This property will allow the customer to configure the memory settings that are used to compile the Rule Assets into an EDS file. Default is `-Xms256m -Xmx512m`

```
com.corticon.ccserver.compile.memorysettings=-Xms256m -Xmx512m
```

Option that will restrict certain types of Rule Messages from being posted to the output of an execution. There are 3 different properties to allow the user to select exactly what is returned to them from the execution. Default is `false` (for all three properties)

```
com.corticon.server.restrict.rulemessages.info=false
com.corticon.server.restrict.rulemessages.warning=false
com.corticon.server.restrict.rulemessages.violation=false
```

Options that allow the user to define how many Rule Messages will be returned from the execution of a Decision Service. This helps to prevent users from accidentally deploying a Decision Service with diagnostic Rule Messages posted when each Rule is fired.

The property `com.corticon.server.execution.xml.rulemessages.messagesinblock` defines how many messages will be returned in the output. Default is `5000`

```
com.corticon.server.execution.xml.rulemessages.messagesinblock=5000
```

The property `com.corticon.server.execution.xml.rulemessages.blocknumber` defines which block of messages will be returned in the response. This allows the user to specify the 2nd, 3rd, or nth number of 1000 messages to be returned. Default is `1` (the first block)

```
com.corticon.server.execution.xml.rulemessages.blocknumber=1
```

Option to relax the enforcement of Custom Data Type Constraints. When set to `true`, a CDT violation will post a warning message and execution will continue. When set to `false`, a CDT violation will cause an exception to be thrown halting execution. Default is `false`

```
com.corticon.vocabulary.cdt.relaxEnforcement=false
```

Option to prepend rule metadata to the business rule statement text. When set to `true`, the rulesheet (if part of a ruleflow) and rule ID will be prepended to all business rule statements at deployment/compile time # When set to `false`, no change is made to the rule statements. Default is `false`

```
com.corticon.reactor.rulestatement.metadata=false
```

Option to ignore the xsi:type related to Entities/Associations in an XML Payload. Default is `false`

```
com.corticon.xml.xsi.type.ignore=false
```

These properties relate to the server monitor thread and server performance diagnostics.

Option to automatically start and configure the server diagnostic thread when an ICcServer is created in the CcServerFactory

```
com.corticon.server.startDiagnosticThread=true
```

Option to enable server diagnostics (requires that the monitor thread has been started.) Default is `true`

```
com.corticon.server.EnableServerDiagnostics=true
```

Wait time (Interval) in milliseconds of the Server Diagnostic Monitor. Default is `30000` - 30 seconds.

```
com.corticon.server.DiagnosticWaitTime=300000
```

The timeout set on an execution thread waiting for an available Reactor from the Decision Service pool. When the thread's wait time exceeds this property's value, then a `CcServerTimeoutException` is thrown for that thread. Default value is `180000` milliseconds -- 3 minutes.

```
com.corticon.server.serverpool.timeout=180000
```

Option to append the version number of the Ruleflow to the compiled `.eds` file. Default value is `true`.

```
com.corticon.server.compile.eds.appendversion=true
```

Option to not add the Entity Name as an `xsi:type` value for those new Entities that are created through Rules using the `.new` operator. This only applies to XML payloads. Default value is `true` -- Entity Name will be added as an `xsi:type`.

```
com.corticon.server.xml.newentities.addtype=true
```

When a record is retrieved from the database during an EDC execution, standard behavior is to synchronize the retrieved record to what was passed in the payload. If the payload value is null, this property will determine if that null value is set into the retrieved record (`true`) or if the null value is ignored (`false`). By default, the null value in the payload will be ignored, and the value from the database will be honored. Default value is `false`.

```
com.corticon.server.execution.sync.nulls=false
```

This property value is the time that an Execution Thread's timeout value. The Execution Thread needs to complete its operation within the time out period, otherwise a `CcServerTimeoutException` will be thrown. The value is in milliseconds. Default value is `180000` (180000ms = 3 minutes)

```
com.corticon.server.execution.queue.timeout=180000
```

Specifies whether the XSD and WSDL generators append the word "Type" at the end of each complexType in the related XSD or WSDL file. This was the standard in earlier versions of the generators. Default value is `false`.

```
com.corticon.servicecontracts.append.typelabel=false
```

Controls all Service Call-Out (SCO) `newEntity` calls. When set to `true`, all entities are added to the work document. When set to `false`, new entities are prevented from being added to the work document, providing a performance boost when the entities are needed for rule process but are not needed in the result document of the rule service. Default value is `true`.

```
com.corticon.services.registerNewSCOEntities=true
```

Controls all new operator calls in rule actions. When set to true, all entities created by rules are added to the work document. When set to `false`, new entities are prevented from being added to the work document, providing a performance boost when the entities are needed for rule process but are not needed in the result document of the rule service. Default value is `true`.

```
com.corticon.reactor.engine.registerNewEntities=true
```

Controls whether null values in the new initializers will create a new entity. When set to `true`, null values in the new initializers will NOT create a new entity. When set to `false`, null values in the new initializers WILL create a new entity. Default value is `false`.

```
com.corticon.reactor.rulebuilder.ForceAllNullCheckingForNew=false
```

Corticon Deployment Console properties

The following properties are used by Corticon's Deployment Console.

Note: Deployment Console properties are stored as a set of defaults in the `CcDeployment.properties` file that is packaged in the `CcConfig.jar`. Each property's notes, options, and default value are listed in this section. You should always set override values in `brms.properties` file located at your work directory root.

For support of BPEL in WSDL generation. Adds a partnerlink section to the generated WSDL to make it BPEL compliant. Default is `false`.

```
com.corticon.deployment.supportBPELinWSDLgeneration=false
```

Adds the default namespace declaration to WSDL generation.

```
com.corticon.xml.addDefaultNamespace=true
```

Adds the default namespace declaration to XSD generation.

```
com.corticon.schemagenerator.addDefaultNamespace=true
```

URLs for the Web Service enabled instances of the Corticon Server. The values appear in the 'SOAP Server URL' field in the Deployment Console window. They are also used by the Test utility as the locations of the available Remote # Servers.

Defaults are `http://localhost:8850/axis/services/Corticon` (Default HTTP port used by the bundled Progress Application 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.soapbindingurl_1=http://localhost:8850/axis/services/Corticon
#com.corticon.deployment.soapbindingurl_2=http://localhost:9080/axis
#com.corticon.deployment.soapbindingurl_3=http://localhost:7001/axis
```

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 `useChoice` 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.deployment.schema.useChoice=true
```

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.servicecontracts.ensureComplianceWithDotNET_WCF=false
```

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`.

```
com.corticon.ccdeployment.sandboxDir=%CORTICON_WORK_DIR%/%CORTICON_SETTING%/CcDeploymentSandbox
```

(Note that this is not the same property as the `sandboxDir` used in Server properties.)

Tells 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 `false`.

```
com.corticon.deployment.ensureUniqueTargetNamespace=false
```

D

Supported RDBMS brands and features for Corticon EDC

Corticon EDC can connect a Vocabulary to an instance of a supported data source. All data sources support **Import/Clear Database Metadata** and **Validate Mappings**. Some Corticon EDC features are not supported in certain supported data sources. Data manipulations and data source startup functions that might be required to ensure error-free interaction between Corticon EDC and a data source are noted.

Note: The mapping of database columns to a Corticon Vocabulary through SQL might experience problems when database columns have hyphens, spaces or other special characters (even though some databases and SQL parsers allow them). The generally accepted valid values are all alphanumeric characters and the underscore character. It is a plus to use all-lowercase names to avoid platform case inconsistencies. For more information on Corticon's accepted names, see the topic *"Vocabulary node naming restrictions" in the Quick Reference Guide*.

The following table summarizes the features:

Data Source	Import Vocabulary	Create Schema	DB Rows: Read/Update	DB Rows: Read-only	Import Metadata	Import Enumerations	Extend to database
DataDirect Cloud: Rollbase	-	-	Yes	-	Yes	Yes	Yes
DataDirect Cloud: Salesforce	-	-	Yes	-	Yes	Yes	Yes
IBM DB2	-	Yes	Yes	Yes	Yes	Yes	Yes
Microsoft SQL Server	-	Yes	Yes	Yes	Yes	Yes	Yes

Data Source	Import Vocabulary	Create Schema	DB Rows: Read/Update	DB Rows: Read-only	Import Metadata	Import Enumerations	Extend to database
MySQL	-	Yes	Yes	Yes	Yes	Yes	Yes
Oracle Database	-	Yes	Yes	Yes	Yes	Yes	Yes
Postgres	-	Yes	Yes	Yes	Yes	Yes	Yes
Progress OpenEdge	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Progress OpenAccess	Yes	-	Yes	Yes	Yes	Yes	Yes

Note: The feature of importing database metadata will infer associations when the information (foreign keys) is available in the data source's metadata.

For the current list of supported data sources and versions, access the web location Progress Corticon 5.6 - Supported Platforms Matrix.

For details, see the following topics:

- DataDirect Cloud: Rollbase
- DataDirect Cloud: Salesforce
- IBM DB2
- Microsoft SQL Server
- MySQL
- Postgres
- Oracle Database
- Progress OpenEdge
- Progress OpenAccess

DataDirect Cloud: Rollbase

Vocabulary: Database Connection definition

- **Database URL** - DataDirect Cloud Server. Must include `TransactionMode=ignore;` Default port: 443
- **Username and Password** - DataDirect Cloud credentials. Rollbase credentials are specified in DataDirect Cloud.
- **Catalog filter** - Yes.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - No.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - No.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- Many-to-many associations are not supported.

DataDirect Cloud: Salesforce

Vocabulary: Database Connection definition

- **Database URL** - DataDirect Cloud Server. Must include `TransactionMode=ignore;` Default port: 443
- **Username and Password** - DataDirect Cloud credentials. Salesforce credentials are specified in DataDirect Cloud.
- **Catalog filter** - Yes.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - No.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - No.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- While Corticon rules allow a Boolean to be `true`, `false`, or `null` (the absence of any value), SalesForce only accepts `true` or `false`. You must trap and handle `null` Booleans in Corticon before passing them to SalesForce.

IBM DB2

Vocabulary: Database Connection definition

- **Database URL** - Default port: 50000
- **Username and Password** - DB2 credentials.
- **Catalog filter** - Yes.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - Yes.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- None.

Microsoft SQL Server

Vocabulary: Database Connection definition

- **Database URL** - Default port: 1433
- **Username and Password** - SQL Server credentials.
- **Catalog filter** - Yes.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - Yes.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- None.

MySQL

Vocabulary: Database Connection definition

- **Database URL** - Default port: 3306
- **Username and Password** - Refer to MySQL documentation or your database administrator.
- **Catalog filter** - No.
- **Schema filter** - No.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - Yes.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- While MySQL presents databases as catalogs, the Database Connection definition cannot apply a filter to those catalogs.

  .

Postgres

Vocabulary: Database Connection definition

- **Database URL** - Default port: 5432
- **Username and Password** - Postgres credentials.
- **Catalog filter** - Yes.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - Yes.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- None.

Oracle Database

Vocabulary: Database Connection definition

- **Database URL** - Default port: 1521
- **Username and Password** - Oracle DB credentials.
- **Catalog filter** - No.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - Yes.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- None.

Progress OpenEdge

Vocabulary: Database Connection definition

- **Database URL** - Default port: 5566
- **Username and Password** - OpenEdge credentials.
- **Catalog filter** - Yes.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - Yes. Only for Vocabulary elements that were not created through a BRVD import.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- **Importing a file to create a Vocabulary** - Import requires a `.brvd` file created in OpenEdge (see Progress OpenEdge documentation for details.) The import function into Corticon is described in *"Importing an OpenEdge Business Rules Vocabulary Definition (BRVD) file" in the Rule Modeling Guide.*
- **Startup of OE server** - It is recommended that you start the OpenEdge database server with the following parameters within the `Proenv` window, shown here with values used in a test environment:

  ```
  proserve db_name -n 65 -Mn 20 -Mpb 4 -Ma 20 -Mi 3 -S port_number
  ```

  where:

  - *`db_name`* is the database name
  - *`port_number`* is the port number
  - Other OpenEdge parameters as described in OpenEdge Database Server parameters.

Progress OpenAccess

Vocabulary: Database Connection definition

- **Database URL** - Default port: 19991. Must include `ServerDataSource=OA_OpenEdgeAppServer`
- **Username and Password** - OpenAccess credentials.
- **Catalog filter** - Yes.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - No.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- **Importing a file to create a Vocabulary** - Import requires a `.brvd` file created in OpenEdge (see Progress OpenEdge documentation for details.) The import function into Corticon is described in *"Importing an OpenEdge Business Rules Vocabulary Definition (BRVD) file" in the Rule Modeling Guide.*
- See the OpenEdge documentation for Progress OpenAccess for constraints and additional information.

E

Access to Corticon knowledge resources

Complete online documentation for the current release

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

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

Corticon guides (PDF):

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

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

Corticon JavaDoc API reference (HTML):

- Corticon Server API
- CorticonExtensions API

See also:

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