A large, stylized graphic of the letter 'P' composed of multiple parallel green lines, creating a 3D effect. It is positioned in the upper right quadrant of the page.

Corticon Server:

Integration and Deployment Guide

Copyright

© 2018 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.

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Deliver More Than Expected, Icenium, Kendo UI, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, 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. Analytics360, AppServer, BusinessEdge, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustAssembly, JustDecompile, JustMock, Kinvey, NativeScript Sidekick, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Sitefinity, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient 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.

Updated: 2018/09/27

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, XML, or JSON payloads.....	14
Chapter 2: Types of Corticon Servers.....	15
Chapter 3: Preparing Studio files for deployment.....	17
Mapping the Vocabulary.....	17
XML/JSON mapping.....	18
Java object mapping.....	21
Verifying Java object mapping.....	24
Inheritance and Java object messaging.....	24
Entity mapping.....	25
Attribute mapping.....	27
Association mapping.....	30
Java generics.....	31
Java enumerations.....	31
Listeners.....	34
Properties that get baked into Decision Services.....	34
Chapter 4: Packaging and deploying Decision Services	37
Properties that impact Decision Service compilation.....	38
Deployment related files	39
Rule asset (ECORE, ERS, ERF) files	39
Test asset (ERT) files	39
Corticon Deployment Descriptor (CDD) files.....	39
Decision Service (EDS) files.....	40
Schema (XSD, WSDL) files.....	40
Using Studio to compile and deploy Decision Services	40
Compiling and deploying to a Corticon Server	43
Compiling and saving to Studio disk for later deployment.....	45
Deploying Decision Services into Web Console Applications from Studio.....	46
Using Web Console to deploy Decision Services.....	49
Using Deployment Descriptors to deploy Decision Services.....	49

Structure of a Deployment Descriptor (.cdd) file.....	49
Setting properties in a CDD file.....	50
Setting deployment properties in a CDD file through APIs.....	51
Example of a complete CDD file.....	53
Using the Server Deployment Console	54
Setting the autoloaddir property.....	57
Using command line utilities to compile Decision Services.....	57
Syntax of the compile and test commands.....	58
Creating a build process in Ant.....	65
Using Server API to compile and deploy Decision Services.....	67
Creating custom context URLs on a web server.....	70
 Chapter 5: Integrating Corticon Decision Services.....	 73
Components of a call to Corticon Server.....	73
The Decision Service Name.....	74
The Data.....	74
Service contracts: Describing the call.....	74
Setting Service Contract properties.....	75
XML workDocument.....	76
Java business objects.....	76
Creating XML service contracts with Corticon Deployment Console.....	76
Types of XML service contracts.....	80
XML Schema (XSD).....	80
Web Services Description Language (WSDL).....	80
Annotated Examples of XSD and WSDLs Available in the Deployment Console.....	80
Passing null values in messages.....	81
Invoking a Decision Service from a different Decision Service.....	82
 Chapter 6: Secure servers with authentication and encryption.....	 87
Implementing deployment security	87
Setting up authentication for secure server access.....	88
Setting up encryption between servers and clients.....	97
 Chapter 7: Invoking Corticon Server.....	 99
Methods of calling Corticon Server.....	99
SOAP call.....	100
Java call.....	100
REST call.....	101
Request/Response mode.....	102
Administrative APIs.....	102
 Chapter 8: Inside Corticon Server.....	 105

Setting Server startup to auto load CDD files.....	105
The basic path.....	106
About Working Memory.....	106
Ruleflow compilation into an EDS file.....	107
Batch processing.....	107
Multi-threading, concurrency reactors, and server pools.....	107
State.....	110
Reactor state.....	110
Corticon Server state.....	110
Dynamic discovery of new or changed Decision Services.....	111
Replicas and load balancing.....	112
Exception handling.....	112
Chapter 9: Decision Service versioning and effective dating.....	113
Deploying Decision Services with identical Decision Service names.....	114
Invoking a Decision Service by version number.....	115
Creating samples of versioned Ruleflows.....	115
Specifying a version in a SOAP request message.....	119
Specifying version in a Java API call.....	121
Default behavior with no target version.....	122
Invoking a Decision Service by date.....	122
Modifying the sample Rulesheets and Ruleflows.....	123
Specifying Decision Service effective timestamp in a SOAP request message.....	124
Specifying effective timestamp in a Java API call.....	126
Specifying both major version and effective timestamp.....	127
Default behavior with no timestamp.....	127
Summary of major version and effective timestamp behavior.....	127
Chapter 10: Using Corticon Server logs.....	129
How users typically use logs.....	129
Changing logging configuration	130
Configuring log content.....	130
Configuring log files.....	131
Troubleshooting Corticon Server problems.....	132
Chapter 11: Performance and tuning guide.....	139
Rulesheet performance and tuning.....	139
Server performance and tuning.....	140
Setting Server build properties.....	140
Setting Server execution properties.....	141
Optimizing pool settings for performance.....	144
Single machine configuration.....	145
Cluster configuration.....	146

Capacity planning.....	147
The Java clock.....	147
Diagnosing runtime performance of server and Decision Services	148

Chapter 12: Enabling Server handling of locales, languages, and timezones.....155

Handling requests and replies across locales.....	156
Examples of cross-locale processing.....	157
Example of cross-locale literal dates.....	160
Example of requests that cross timezones.....	165

Chapter 13: Implementing Rule Execution Recording in a database...167

Chapter 14: Request and response examples.....173

JSON/RESTful request and response messages.....	173
About creating a JSON request message for a Decision Service.....	174
Sample JSON request and response messages.....	180
XML requests and responses.....	188
Sample XML CorticonRequest content.....	188
Sample XML CorticonResponse content.....	189

Chapter 15: Sample client applications.....191

Appendix A: Service contract examples.....193

Examples of XSD and WSDLs available in the Deployment Console.....	194
Vocabulary-level XML schema, FLAT XML messaging style.....	194
Vocabulary-level XML schema, HIER XML messaging style.....	201
Decision-service-level XML schema, FLAT XML messaging style.....	201
Decision-service-level XML schema, HIER XML messaging style.....	202
Vocabulary-level WSDL, FLAT XML messaging style.....	204
Vocabulary-level WSDL, HIER XML messaging style.....	206
Decision-service-level WSDL, FLAT XML messaging style.....	207
Decision-service-level WSDL, HIER XML messaging style.....	207
Examples of service contract reports generated from a Ruleflow.....	208
Guidelines for REST/JSON Service Contracts.....	208
Extended service contracts.....	209
Extended datatypes.....	210

Appendix B: Corticon API reference.....213

Java API.....	213
---------------	-----

REST Management API.....	213
Common REST request/response Types.....	214
Error handling in the REST Management API	215
Using the REST API Swagger documentation.....	215
Accessing the Vocabulary metadata of a Decision Service.....	217
Summary of REST methods for management of Decision Services	220
 Appendix C: Setting Web Console server properties.....	233
 Appendix D: Configuring Corticon properties and settings.....	235
 Appendix E: Corticon 5.7.2 Online Tutorials and Documentation.....	239

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”.	<ul style="list-style-type: none"> 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.	<ul style="list-style-type: none"> 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.	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> • SOAP: RPC or Document-style 	<ul style="list-style-type: none"> • XML String (RPC-style) • XML Document (Document-style)
Java Session EJB	<ul style="list-style-type: none"> • Corticon Server API via JMS • Corticon Server API via RMI 	<ul style="list-style-type: none"> • XML String or JDOM • collection or map of Java Business Objects
Java Classes	<ul style="list-style-type: none"> • in-process Java methods from the Corticon Server API 	<ul style="list-style-type: none"> • 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, XML, or JSON 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 [Web Services guide](#).

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.

Preparing Studio files for deployment

For details, see the following topics:

- [Mapping the Vocabulary](#)
- [XML/JSON mapping](#)
- [Java object mapping](#)
- [Entity mapping](#)
- [Attribute mapping](#)
- [Association mapping](#)
- [Java generics](#)
- [Java enumerations](#)
- [Listeners](#)
- [Properties that get baked into Decision Services](#)

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/JSON 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 or JSON document – then your Vocabulary may need to be configured to match the naming convention of the elements in your XML/JSON payload.

Displaying XML/JSON Mapping

On the Vocabulary menu, choose **Add Document Mapping > Add XML/JSON Mapping**.

Entity Mapping

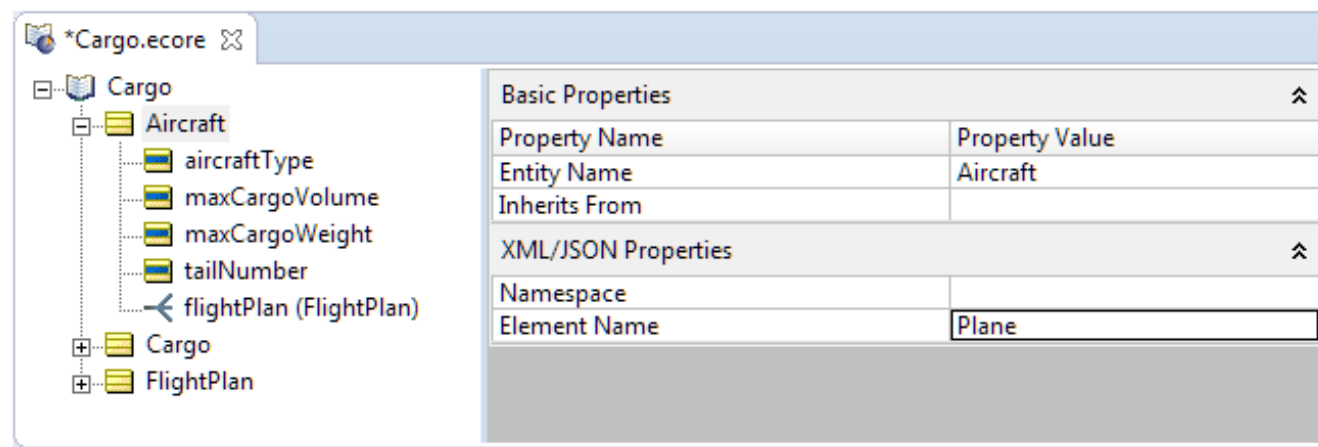
When XML/JSON mapping has been added to the Vocabulary, its Entity properties are displayed.

Table 3: XML/JSON Mapping Entity Properties

Property	Value
XML Namespace	Specifies the full namespace of XML Element Name when there is no exact match.
XML Element Name	Specifies the XML Element Name when there is no exact match.

Vocabulary entities correspond to XML complex elements (complexType). 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 **Element Name** property, as shown:.

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 **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

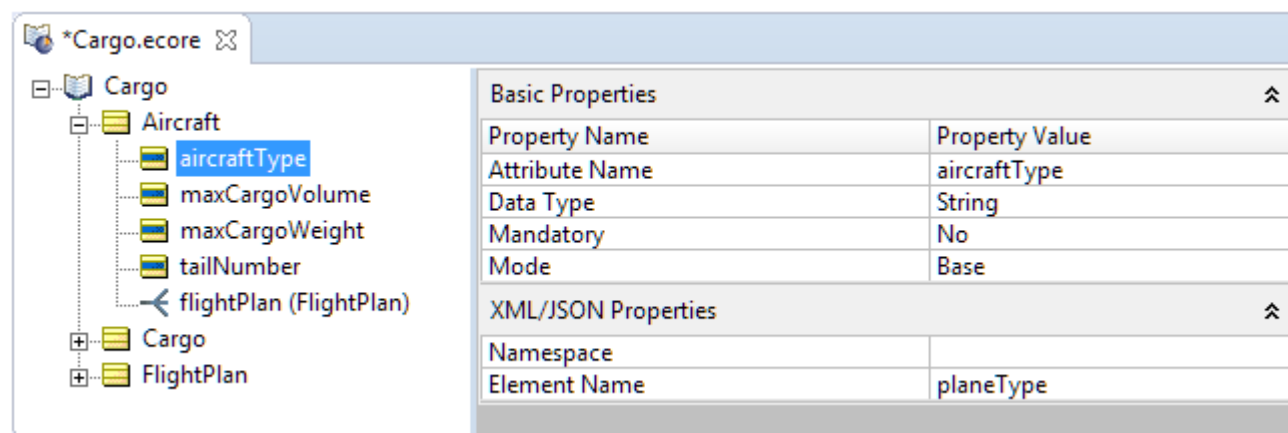
When XML/JSON mapping has been added to the Vocabulary, its Attribute properties are displayed.

Table 4: XML/JSON Mapping Attribute Properties

Property	Value
XML Namespace	Specifies the full namespace of XML Element Name when there is no exact match.
XML Element Name	Specifies the XML Element Name when there is no exact match.

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 **Element Name** property, as shown in the following figure.

Figure 2: Mapping a Vocabulary Attribute to an XML SimpleType



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

Association Mapping

When the Vocabulary has added XML/JSON mapping, you set their properties on the properties page of the Association.

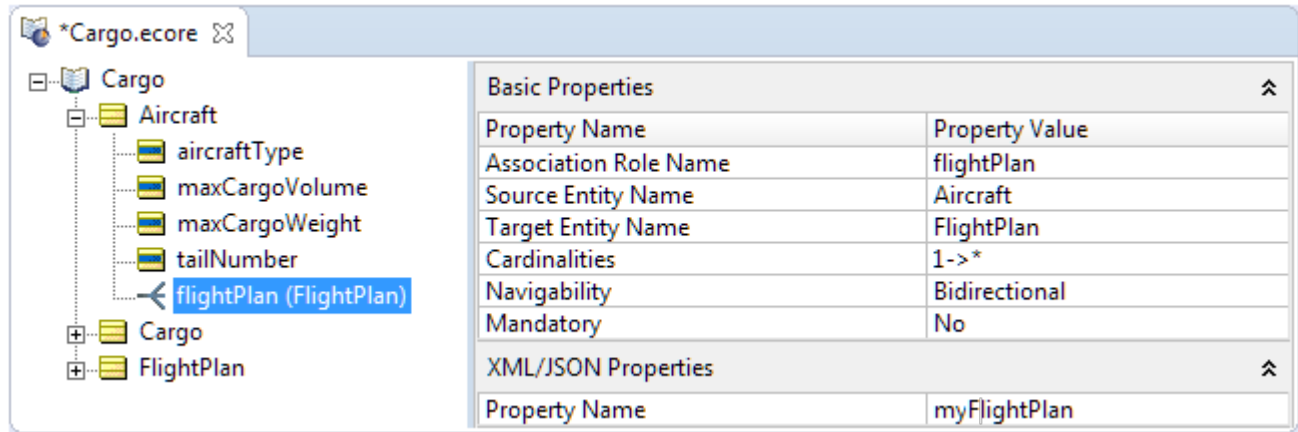
>

Table 5: XML Mapping Association Properties

Property	Value
XML Property Name	Specifies the XML Element Name when there is no exact match to the Vocabulary association name.

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 **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" />
</xsd:schema>
```

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" />
    </xsd:schema>
  </types>
</definitions>
```

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 property

`com.corticon.deployment.ensureUniqueTargetNamespace` in a server's `brms.properties` file to tell the XSD and WSDL Generators to create unique Target Namespaces inside the output document.

When 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>`

When 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`

The default value is `false`. If changed, a restart of the Server and the Deployment Console is required.

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" />
</xsd:schema>
```

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">
      <xsd:element name="CorticonRequest" type="tns:CorticonRequestType" />
      <xsd:element name="CorticonResponse" type="tns:CorticonResponseType" />
    </xsd:schema>
  </types>
  <message name="CorticonRequest" xmlns="http://schemas.xmlsoap.org/wsdl/" />
  <message name="CorticonResponse" xmlns="http://schemas.xmlsoap.org/wsdl/" />
  <port name="Corticon" binding="tns:CorticonBinding" />
  <binding name="CorticonBinding" type="tns:CorticonRequestType" />
  <binding name="CorticonBinding" type="tns:CorticonResponseType" />
</definitions>
```

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.

Note: For information about mapped Java objects usage on a .NET server, see 'Using .NET Business Objects as payload for Decision Services' in the *.NET Server Guide*

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.

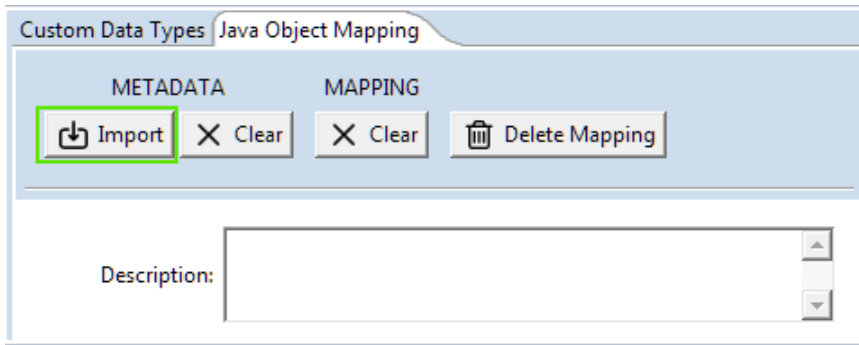
To import package metadata:

1. Open your Vocabulary in Corticon Studio's **Vocabulary Edit** mode.
2. On the Vocabulary menu, choose **Add Document Mapping > Add Java Object Mapping**.

A new tab is added to the top level of the Vocabulary.

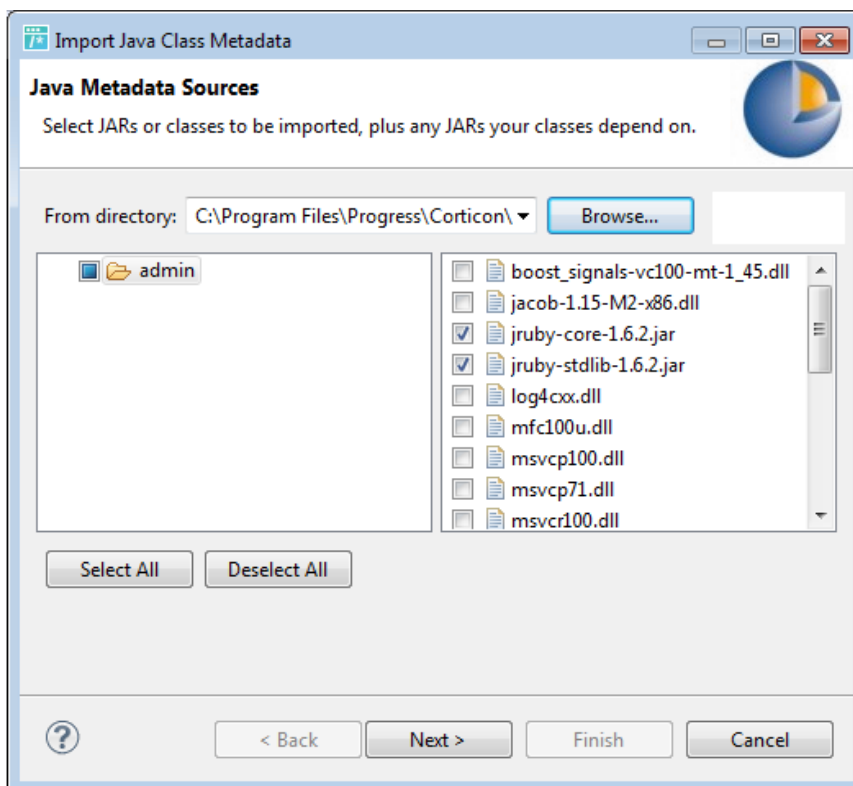
3. On the **Java Object Mapping** tab, click **Import**

Figure 8: Importing Java Class Metadata for Mapping



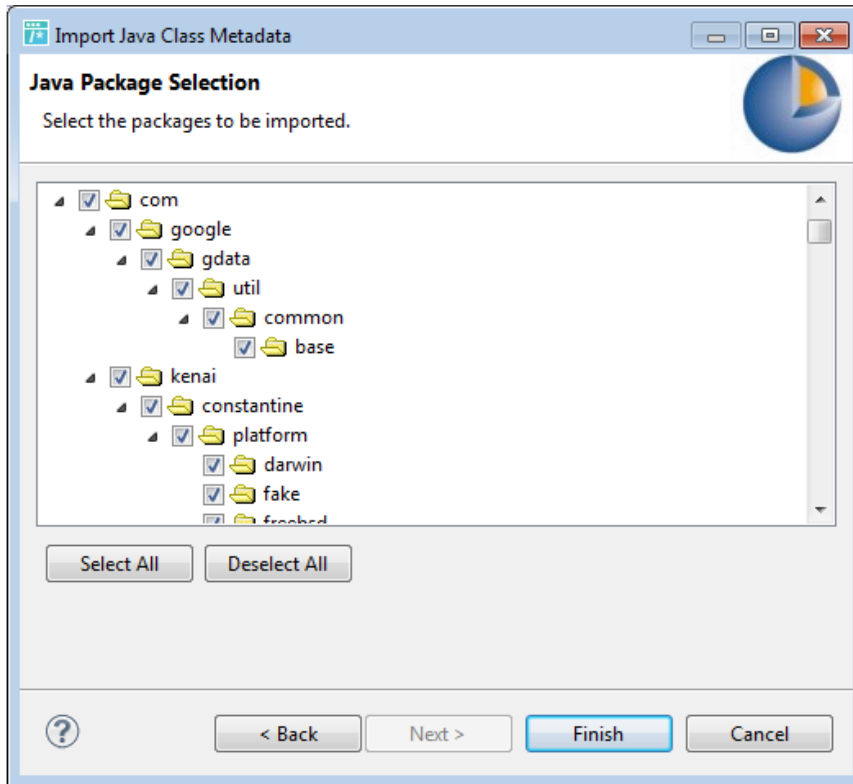
4. Use the **Browse** button to select the location of your Java Business Objects. They should be compiled class files or Java archives (.jar files).

Figure 9: Browsing to your Java Class files



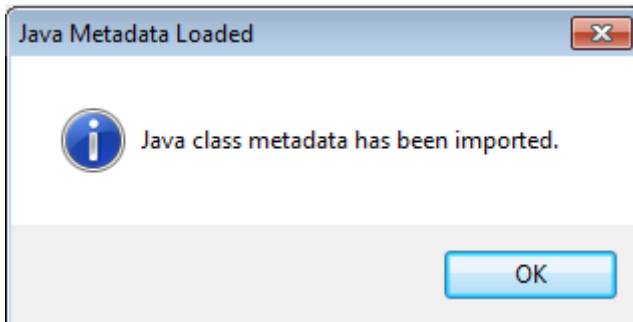
5. Select the package containing the Java business objects as shown:

Figure 10: Importing Java Class Metadata for Mapping



6. 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.

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 12: Problem window showing list of current mapping problems

Properties Problems Error Log Properties			
0 errors, 5 warnings, 0 infos			
Description	Res...	Path	Location
Warnings (5 items)			
No Java object mapping for association cargo could be found in imported metadata.	Cargo.ecore	Samples/Rul...	FlightPlan.cargo
No Java object mapping for attribute packaging could be found in imported metadata.	Cargo.ecore	Samples/Rul...	Cargo.packaging

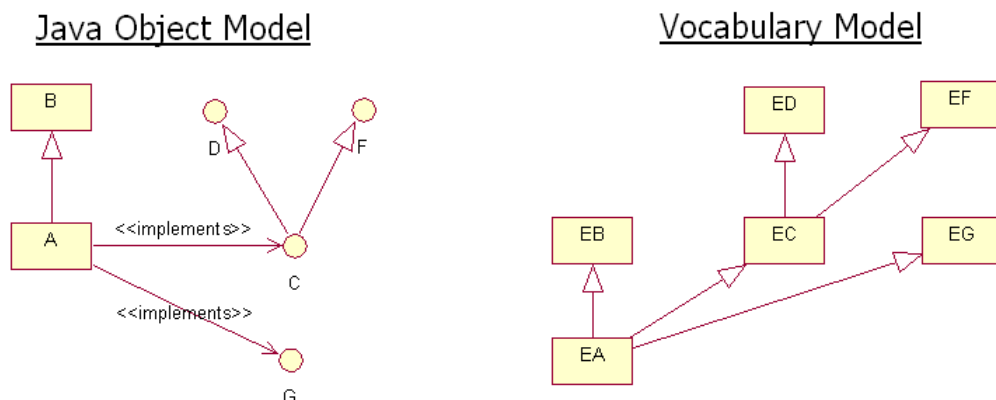
Note: The **Problems** view typically opens in the lower section of the Corticon Studio window -- if you do not see it, choose **Window > Show View > Problems**.

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

Inheritance and Java object messaging

Each Entity in a Vocabulary can be mapped to a Java Class or Java Interface. Java Classes may have one ancestor. Java Interfaces may have multiple ancestors. A Java Class may implement one or more Interfaces. Say a Java Class A inherits from Java Class B and implements Java Interfaces C & G. Say Java Interface C has as its ancestors Java Interfaces D & F. Say these Classes and Interfaces are mapped to Entities EA, EB, EC, ED, EF & EG in the Vocabulary. The relationships amongst the Java Classes shall be reflected in the Vocabulary using multiple inheritance. Entity EA shall have as its ancestors Entities EB, EC & EG. Entity EC shall have as its ancestors entities ED & EF as shown below:

Figure 13: How the Vocabulary Incorporates Inheritance from a Java Object Model



When a collection of Java objects are passed into the engine through the JOM API, the Java translator determines how to map them to the internal Entities using the following algorithm:

Naming conventions used in the graphic above:

- DS = Decision Service
- JO = Java Object in input collection
- JC = Java Class for the JO and any of its direct or indirect ancestors
- JI = Java Interfaces implemented directly or indirectly by JO
- E = A Vocabulary Entity with no descendents found in DS context
- AE = An Ancestor Entity (one with descendents) found in DS context
- CDO = In memory Java Data Object created by Corticon for use in rule execution

For each E:

- If there is a JO whose JC or JI is mapped to E then
 - Instantiate a CDO for E and link to JO
 - Put CDO in E bucket
- Traverse E's inheritance hierarchy one level up
 - For each AE discovered in current level:
 - Put CDO in AE bucket
- If E has another level of inheritance hierarchy, repeat last step

This design effectively attempts to instantiate the minimum number of CDOs possible and morphs them into playing multiple Entity roles. Ideally, no duplicate copies of input data exists in the engine's working memory thus avoid data synchronization issues.

Entity mapping

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

Figure 14: 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     // Public 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() {}

```

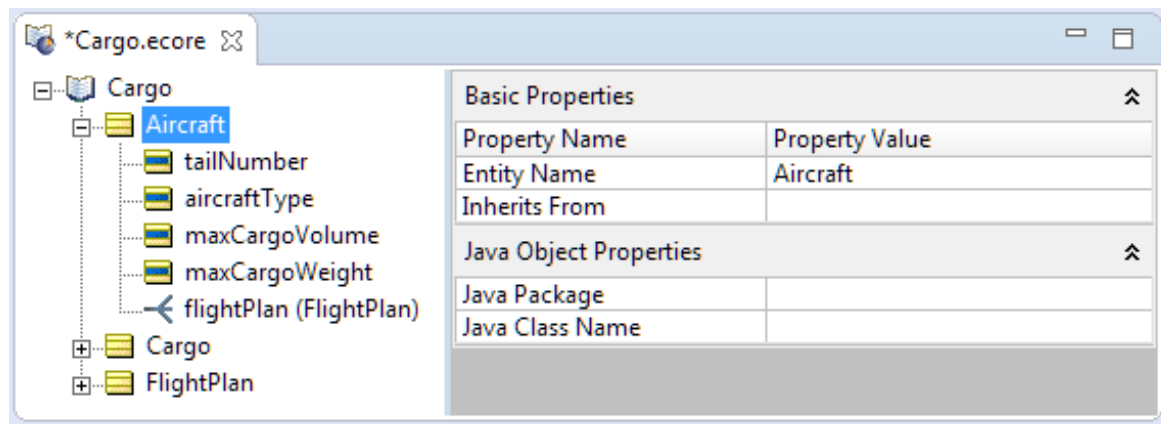
We can see in line 6 of this figure that this class is not actually named `Aircraft` – it is named `MyAircraft`. The automatic mapper attempts to locate a class by the same name as each entity. In the case of `Aircraft`, it looks for a class named `Aircraft`. Not finding one, it leaves the field empty,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

When Java Object mapping has been added to the Vocabulary, its Entity properties are displayed.

Table 6: Java Object Mapping Entity Properties

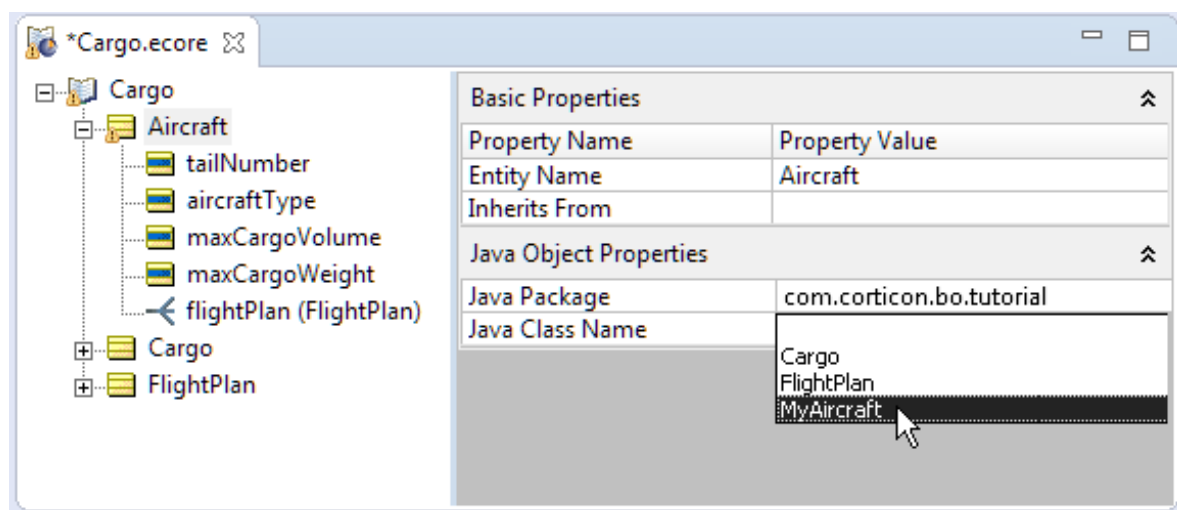
Property	Value
Java Package	Specifies the package to be used for Java class metadata mapping.
Java Class Name	Maps the entity to the specified class when no class exists with the entity name.

Figure 15: 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 16: 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 17: 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

```

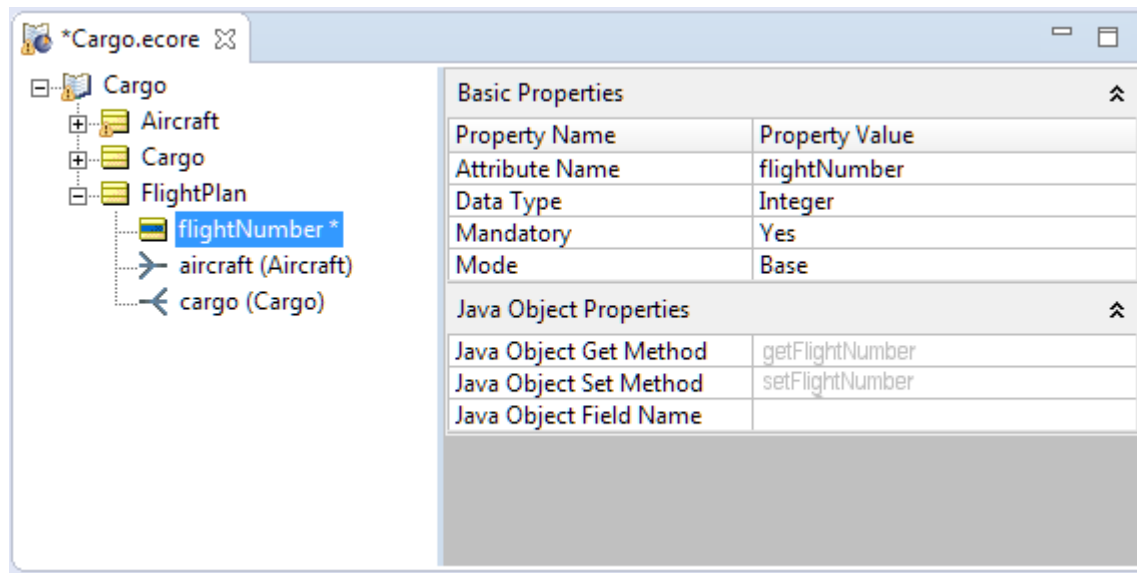
When Java Object mapping has been added to the Vocabulary, its Attribute properties are displayed.

Table 7: Java Object Mapping Attribute Properties

Java Object Get Method	Manually specifies the GET method of a class property that does not conform to naming conventions of the auto-mapper.
Java Object Set Method	Manually specifies the SET method of a class property that does not conform to naming conventions of the auto-mapper.
Java Object Field Name	Manually specifies a public instance variable name.

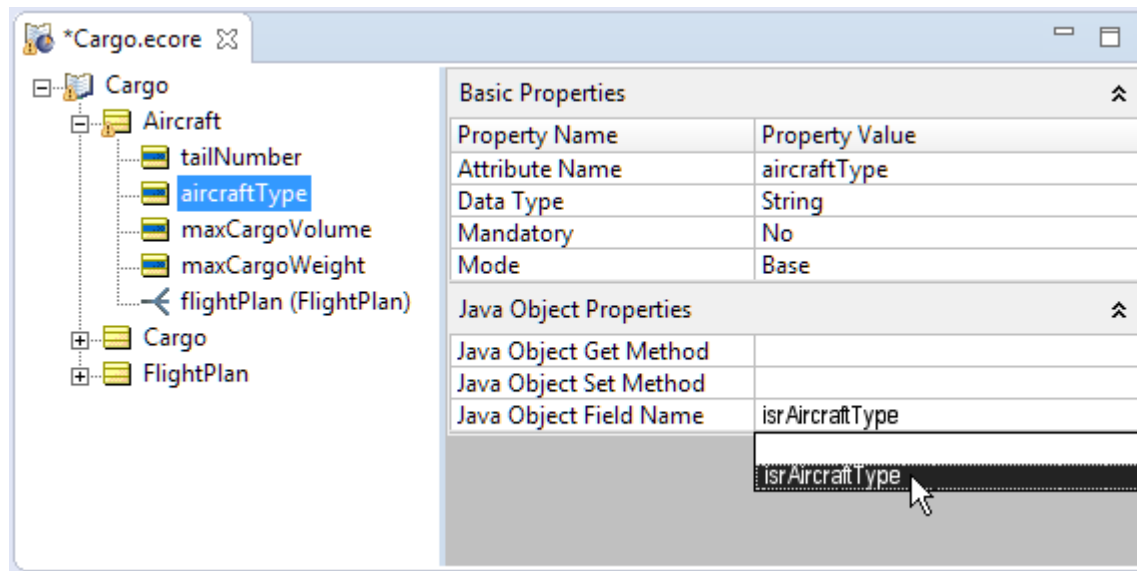
In the case of attribute `tailNumber`, the mapper finds get and set methods that conform to this naming convention, so the method names are inserted into the fields in gray type, as shown in the following figure.

Figure 18: Auto-Mapped Attribute Method Names



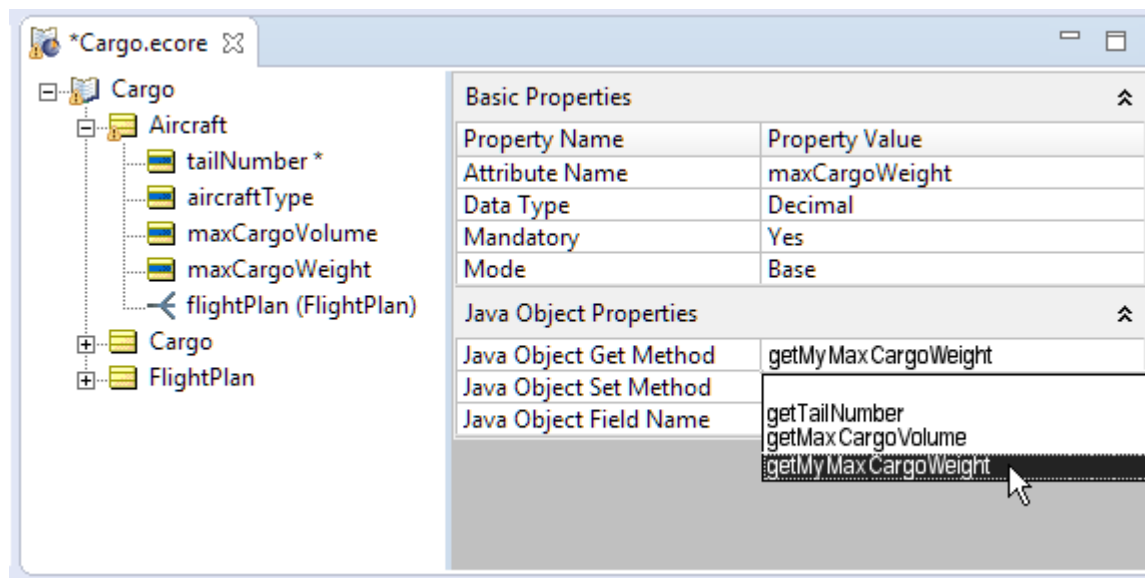
In those cases where the mapper cannot locate the corresponding methods, you will need to select them manually. Notice in the `MyAircraft` class shown in [First Portion of MyAircraft Class](#), no get and set methods exist for `istrAircraftType` since it is a public instance variable. Therefore, we need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

Figure 19: 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.

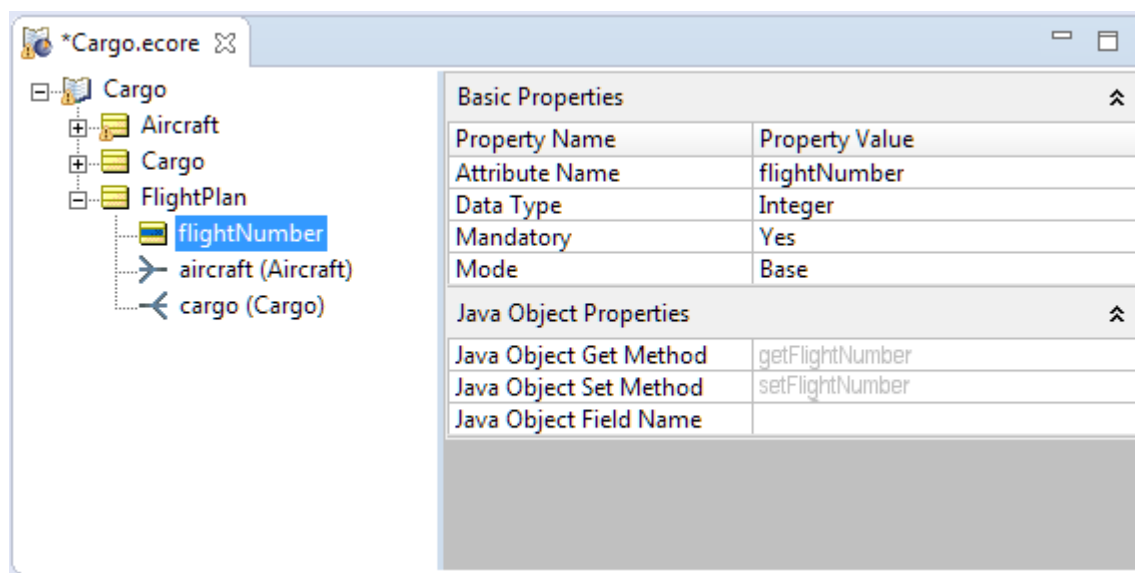
Figure 20: Manually Mapped Property Get and Set Method Names



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 21: 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

When the Vocabulary has added Java Object mapping, you set their properties on the properties page of the Association.

Table 8: Java Object Mapping Association Properties

Property	Value
Java Object Get Method	Manually specifies the GET method of a class property that does not conform to naming conventions of the auto-mapper.
Java Object Set Method	Manually specifies the SET method of a class property that does not conform to naming conventions of the auto-mapper.
Java Object Field Name	Manually specifies a public instance variable name.

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 **Java Object Set Method** drop-downs, as shown in [Manually Mapped Association Get and Set Method Names](#), below.

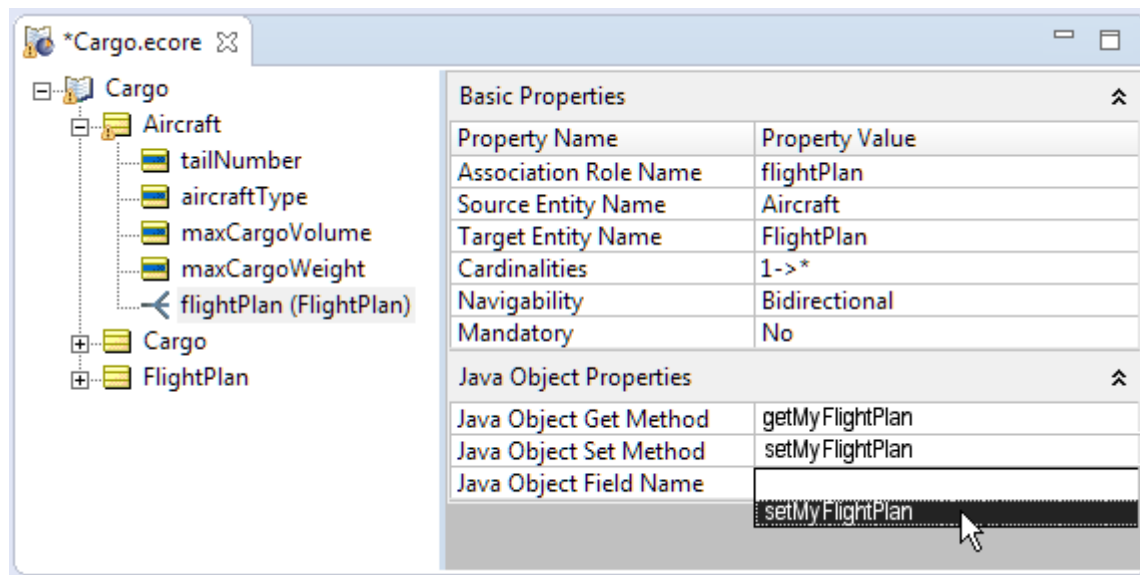
Figure 22: Third Portion of MyAircraft Class

```

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

```

Figure 23: 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 24: Vocabulary CDT Labels must match Business Object Enumeration Labels (Types)

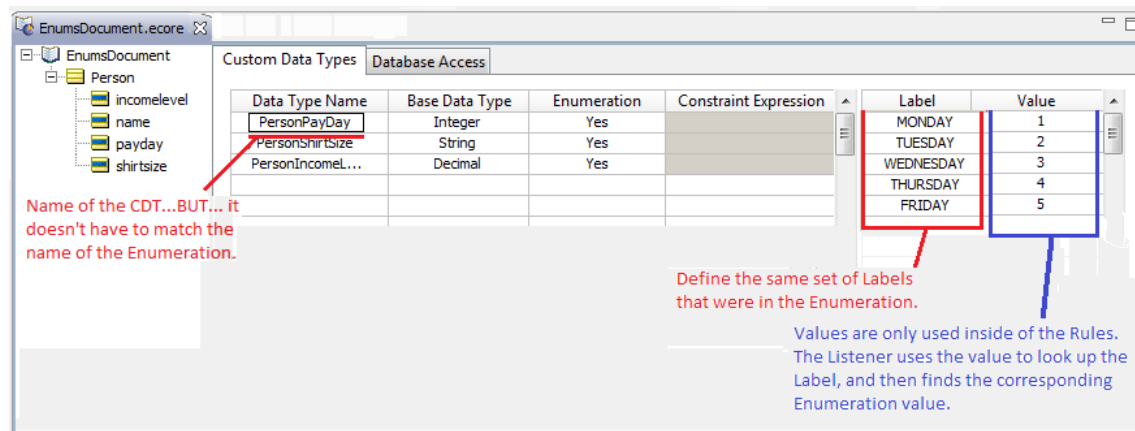
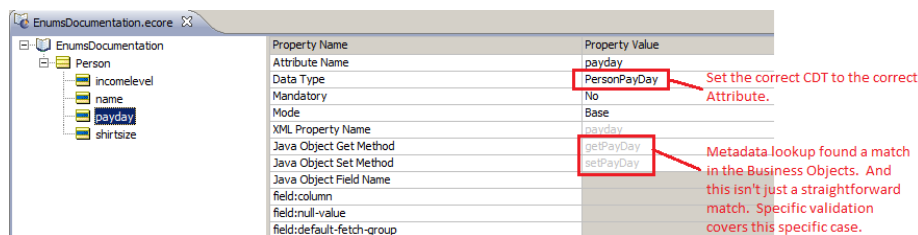


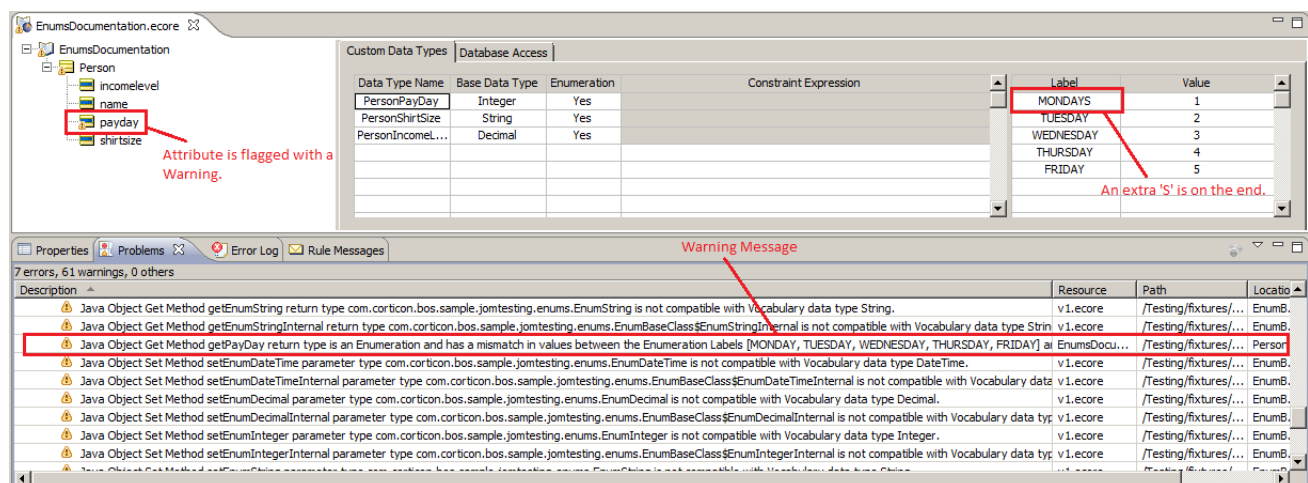
Figure 25: Vocabulary Mapper found correct Metadata based on matching enumeration labels



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 26: 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].

Listeners

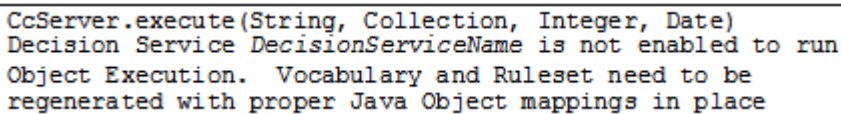
During runtime, when an attribute's value is updated by rules, the update is communicated back to the business object by way of "Listener" classes. Listener classes are compiled at deployment time when Corticon Server detects a Ruleflow using Java Object Messaging. Once compiled, these Listener classes 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 27: Server Error Message When Listeners Not Present

A screenshot of a server error message displayed in a text box. The message is in a monospaced font and contains the following text: `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.

```
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
```

Properties that get baked into Decision Services

The following properties are settings you can apply to your Corticon Server installation by adding the properties and appropriate values as lines in its `brms.properties` file, and then restarting Server. By *baked in*, the setting is incorporated into the compiled Decision Service, and as such cannot be changed by resetting the value on a server where the Decision Service runs.

By default, null attributes generate a warning when on the right hand side of an assignment expression. This value will prevent warning messages from being generated when an attribute's value is null. This is useful for when an extended operator is being used to generate a value and it is possible for some parameters to be null

```
com.corticon.reactor.rulebuilder.DisableWarningOnNullAttribute=false
```

By default, attributes are checked against a null value to prevent further rule evaluation. This value can disable the null checks on attribute parameters used in an extension call out thereby allowing null values to be passed into an extended operator call

```
com.corticon.reactor.rulebuilder.DisableNullCheckingOnExtensions=false
```

Specifies whether the rule engine uses Loop Container Strategy. Loop Container Strategy will create a Rule container object for rules that form a loop, just as when loops are enabled, so that when sequential rules are executed they are executed as if they are in a loop, but without looping.

Unspecified value is true.

```
com.corticon.reactor.rulebuilder.UseLoopContainerStrategy=false
```

For information about related properties

See also:

- [Properties that get baked into Decision Services](#) on page 34
- [Setting Server execution properties](#) on page 141
- [Setting Server build properties](#) on page 140
- [Setting Service Contract properties](#) on page 75

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:

- [Properties that impact Decision Service compilation](#)
- [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](#)
- [Creating custom context URLs on a web server](#)

Properties that impact Decision Service compilation

The following properties are settings you can apply to your Corticon Server installation by adding the properties and appropriate values as lines in its `brms.properties` file, and then restarting Server. These properties apply only when compiling assets into a Decision Service.

Compile option: This property lets you configure memory settings for compiling the Rule Assets into an EDS file.

Default value is `-Xms256m -Xmx512m`

```
com.corticon.ccserver.compile.memorysettings=-Xms256m -Xmx512m
```

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 value 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 value is `true`

```
com.corticon.server.compile.add.wsdl=true
```

For information about related properties

See also:

- [Properties that impact Decision Service compilation](#) on page 38
- [Setting Server execution properties](#) on page 141
- [Setting Server build properties](#) on page 140
- [Setting Service Contract properties](#) on page 75

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, attributes, 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.

Corticon Deployment Descriptor (CDD) files

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

Important: CDD deployment of Ruleflows - CDD Deployment of `.erf` files is not supported on all application servers. The recommended practice is to deploy precompiled `.eds` files when using CDD deployment.

When Corticon 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 57 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 54 - The graphical Deployment Console is included in both Corticon Server installations.
- [Using Deployment Descriptors to deploy Decision Services](#) on page 49 - 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 uses custom Extensions or Service Call-Outs (SCOs), be sure to add their classes to the project as described in *"Using extensions when creating Decision Services" in the Guide to Creating Corticon Extensions*.

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 74 discusses these concepts in greater depth, and the topics in [Service contract examples](#) on page 193 show each of the various types.

This section includes [Generating XSD and WSDL schema files](#) on page 60 as part of the command line utilities, while [Creating XML service contracts with Corticon Deployment Console](#) on page 76 discusses its usage as part of the section "Integrating Corticon Decision Services."

Using Studio to compile and deploy Decision Services

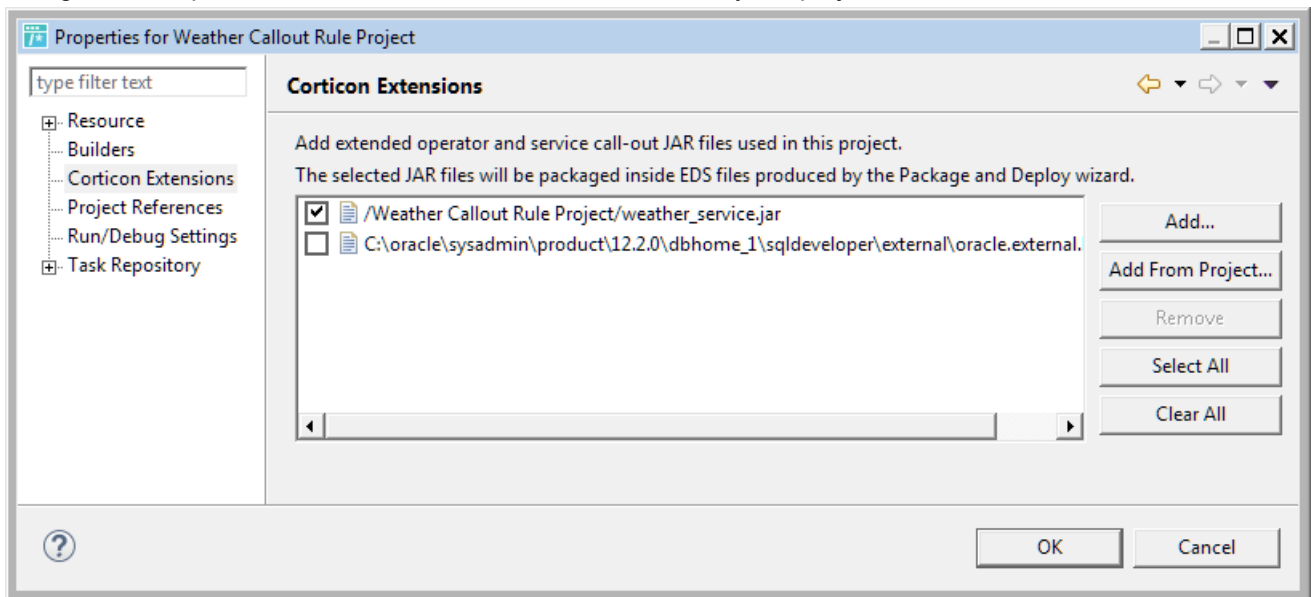
Within Corticon Studio you can package and deploy Decision Services. This is particularly useful during development and testing. In production, you typically would not deploy from Studio.

Adding additional JARs for selected projects

When projects require additional JARS, they can be included in the Decision Service package. These could be for extended operators, service callouts, ADC, or business object JARs for Java Object Messaging.

To add JARs to a project:

1. Right-click on a project name in the Studio's Project Explorer that requires additional JARs, and then choose **Properties**.
2. Click **Corticon Extensions**.
3. Navigate in the panel to locate and list all the JAR files used by the project, as illustrated:



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

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

It is a good idea to include a JAR in the package as it ensures that it won't be affected by any variations of that JAR elsewhere. However, there are cases -- such as where many Decision Services are dependent on the same large JAR -- where it is more efficient to reference it at a common location on servers.

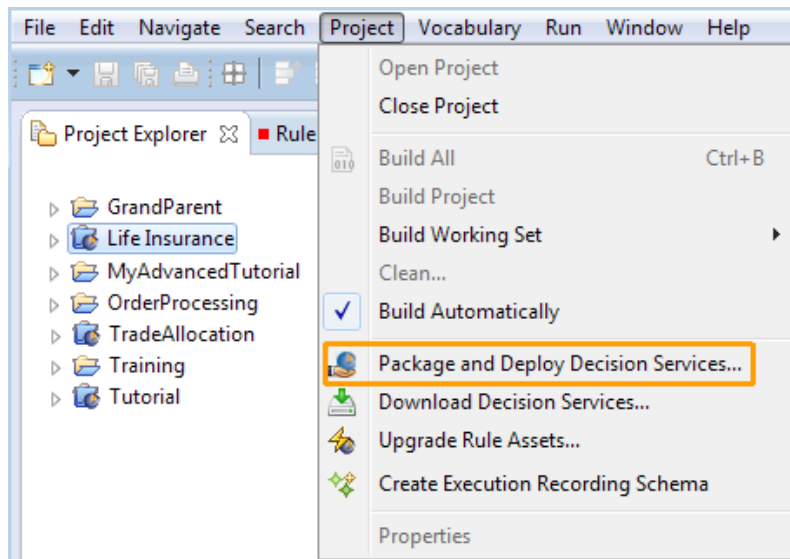
Setting the context for packaging

The projects that will be presented in the Studio's Package and Deploy Decision Services 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.

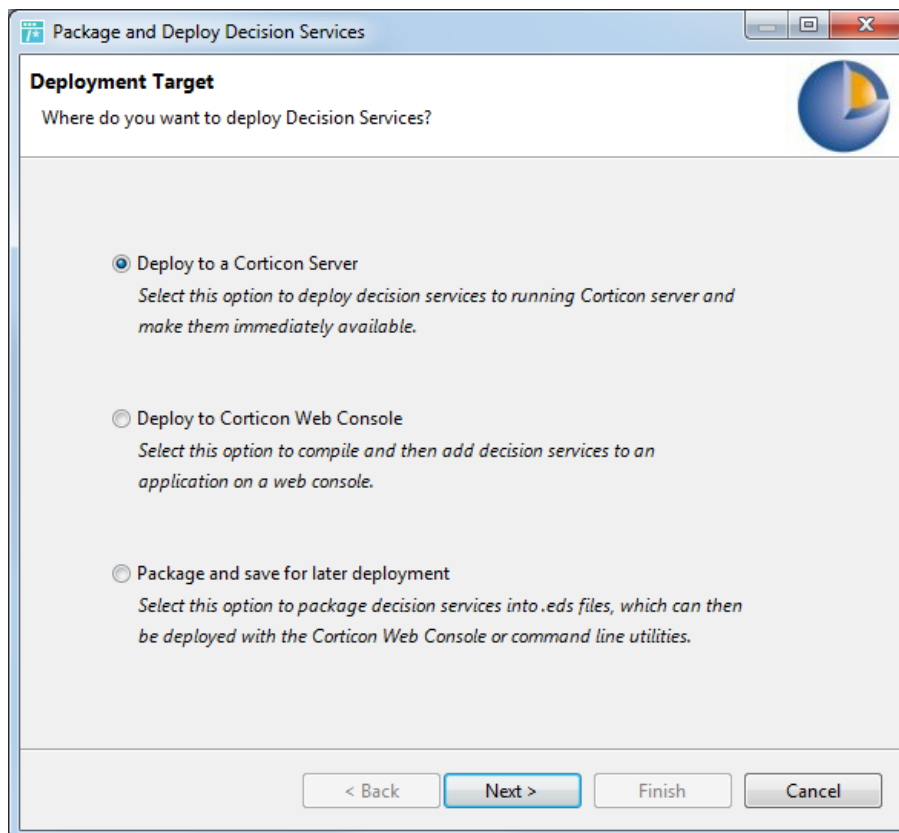
Starting the Package and Deploy Decision Services wizard

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:



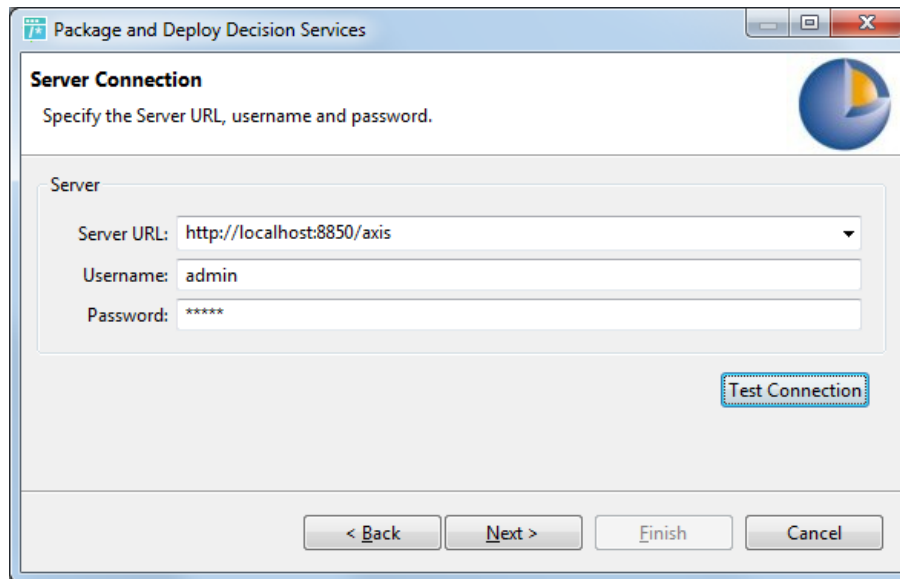
The packaged Decision Services can be saved in the file system, or deployed directly into an available Corticon Server at a specified location and port. Select your preference, and then click **Next**.

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.

Note: If your target server is an IIS Server, you must set the property `com.corticon.studio.client.soap.clienttype=IIS` in the Studio's `brms.properties` file and then restart Studio. If you intend to later connect to a Java server, set that property to `com.corticon.studio.client.soap.clienttype=JAVA`.

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

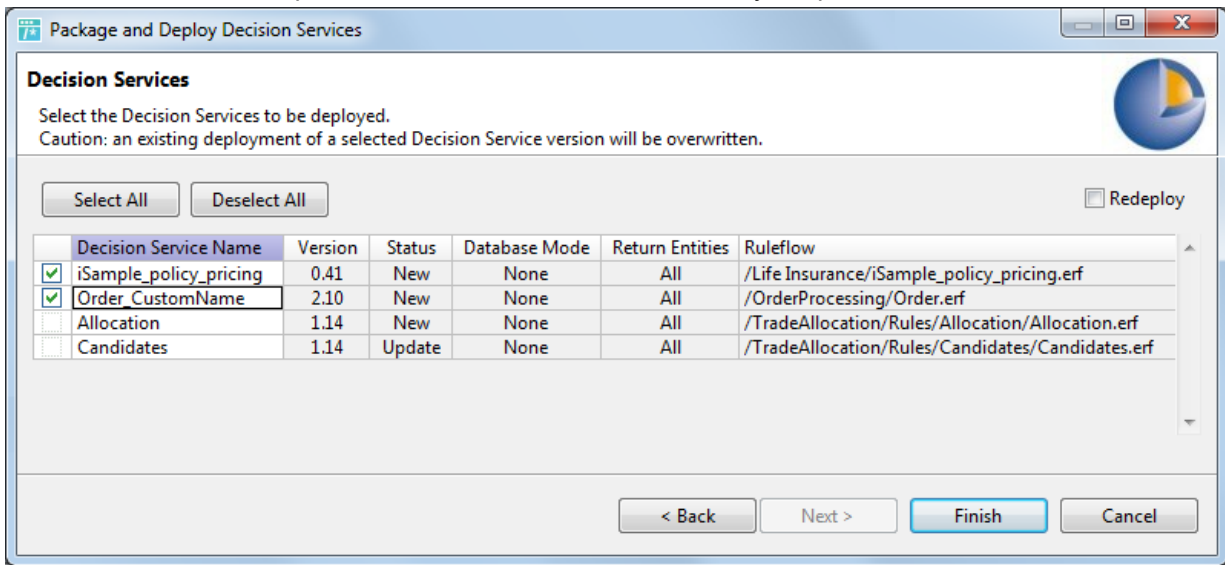


1. Enter the **Server URL** of the Corticon Server.
2. Enter the **Username** and **Password** for the server. For administrative permissions on Progress Application Server, try the default credentials `admin` and `admin`.
3. Click **Test Connection**.
 - On successful connection, the system displays: **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, security certificate not registered, or username/password is incorrect.**
 - 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**.

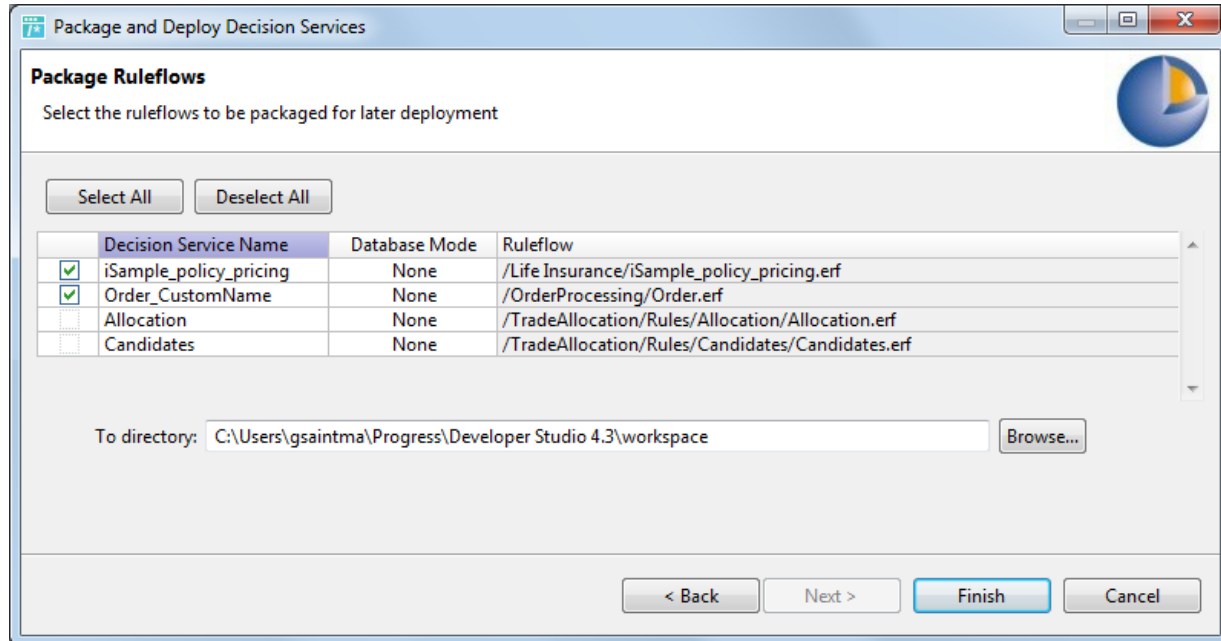
The packaging and deployment progress is shown. It can be stopped (although what has been completed is not backed out) by clicking the **Stop** button adjacent to the progress bar.

When all the packaging and deployment processes are successful, the wizard alerts you with a 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 save 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-accessible 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 **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.

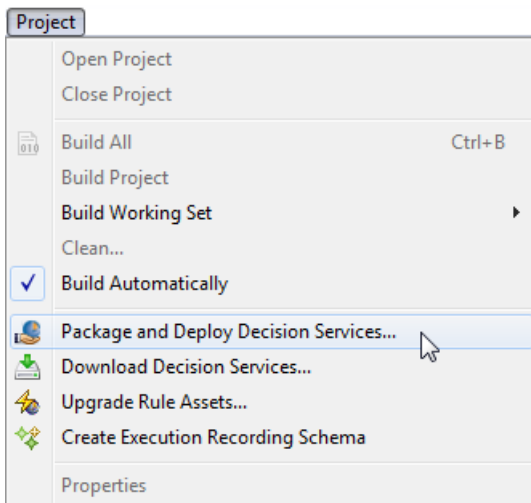
When the processes are successful, the wizard alerts you with a **Compilation Success** message. If there are problems, the wizard lists the errors.

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.

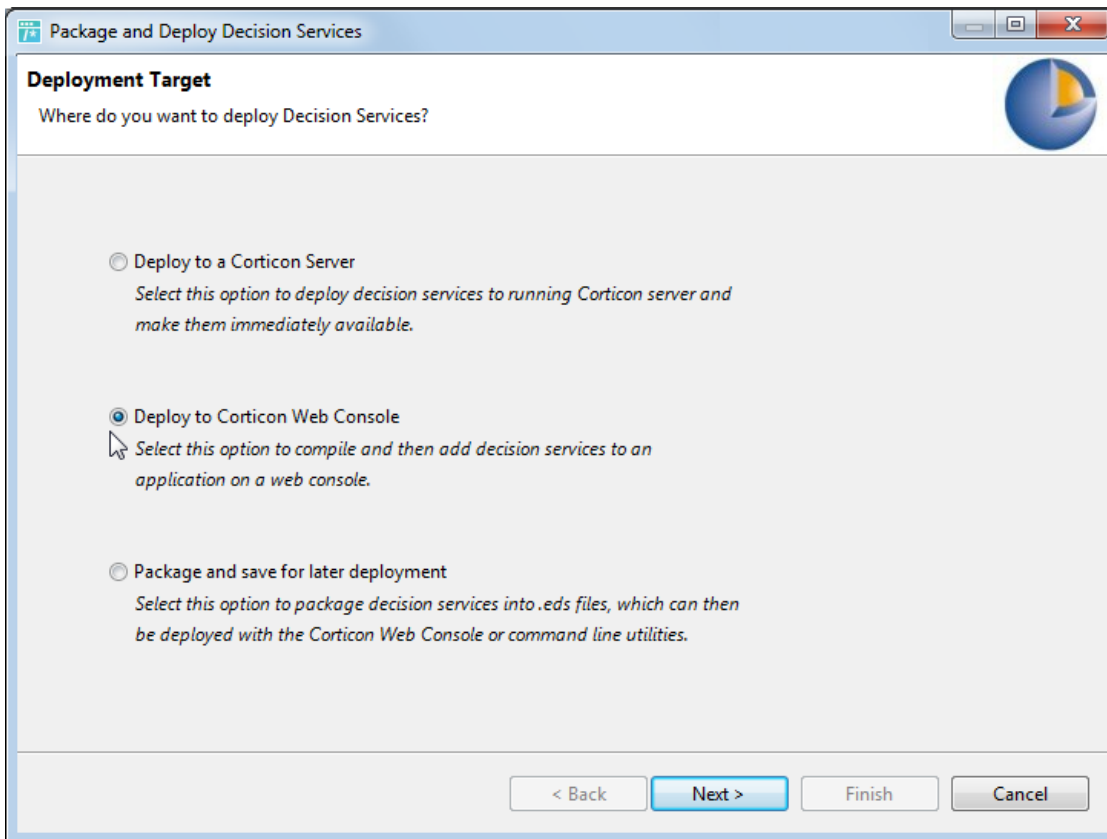
To deploy Ruleflows in Corticon Studio as Decision Services on servers managed by the Web Console:

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**:

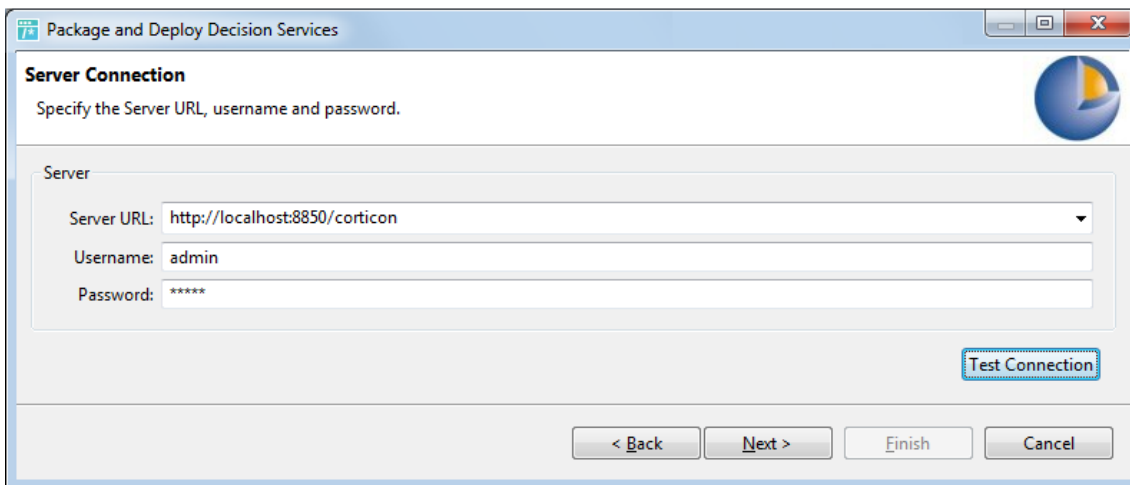


When a project is selected or there is an active file in its editor, the Ruleflows of only that project will be listed. When no projects are selected and no files are in their editor, the Ruleflows of all projects in the workspace will be listed.

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.

Application
Select the application to which Decision Services will be added.

☐ Add to Existing Application
Select Application:

☒ Create New Application
Name:
Description (optional):
Server Group:
local server
Server Group One

< Back Next > Finish Cancel

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:

Decision Services
Select the Decision Services to be deployed.
Caution, an existing deployment of a selected Decision Service version will be overwritten.

Select All Deselect All

	Decision Service Name	Version	Status	Database Mode	Return Entities	Ruleflow
<input checked="" type="checkbox"/>	MyAdvancedTutorial	1.0	New	None	All	/MyAdvancedTutorial/MyAdvancedTutorial.erf
<input checked="" type="checkbox"/>	tutorial_example	0.16	New	None	All	/Tutorial/Tutorial-Done/tutorial_example.erf

< Back Next > Finish Cancel

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.

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	<u>false</u> true
PROPERTY_MAX_POOL_SIZE	<u>1</u> [positive integer]
PROPERTY_MESSAGE_STRUCTURE_TYPE	<u><null></u> HIER FLAT
PROPERTY_DATABASE_ACCESS_MODE	<u><null></u> R RW
PROPERTY_DATABASE_ACCESS_RETURN_ENTITIES_MODE	<u>ALL</u> IN
PROPERTY_DATABASE_ACCESS_PROPERTIES_PATH	[explicit or relative path]
PROPERTY_DATABASE_ACCESS_CACHING_ENABLED	<u>false</u> (default) true
PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_INFO	<u>false</u> true
PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_WARNING	<u>false</u> true
PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_VIOLATION	<u>false</u> true
PROPERTY_EXECUTION_RESTRICT_RESPONSE_TO_RULEMESSAGES_ONLY	<u>false</u> true

Some of the options can be set in the `brms.properties` file:

Property name	Value
<code>com.corticon.server.restrict.rulemessages.info</code>	<u>false</u> true
<code>com.corticon.server.restrict.rulemessages.warning</code>	<u>false</u> true
<code>com.corticon.server.restrict.rulemessages.violation</code>	<u>false</u> true
<code>com.corticon.server.restrict.response.rulemessages.only</code>	<u>false</u> 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 213 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 choosing 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 73.

Note:

You might want to use the `brms.properties` setting that determines the path to an existing directory that will be used exclusively by Deployment Console to pre-compile Ruleflow files into .eds files. It is important that you use forward slashes as path separator. Example: `c:/Users/me/CcDeploymentSandbox`

Default value is `%CORTICON_WORK_DIR%/System/CcDeploymentSandbox`

```
com.corticon.ccdeployment.sandboxDir=%CORTICON_WORK_DIR%/CORTICON_SETTING%/CcDeploymentSandbox
```

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

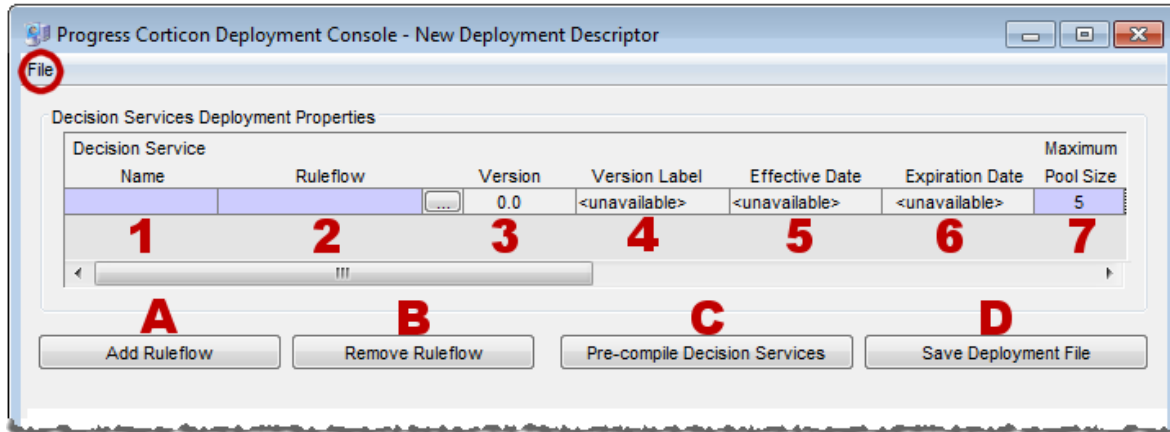
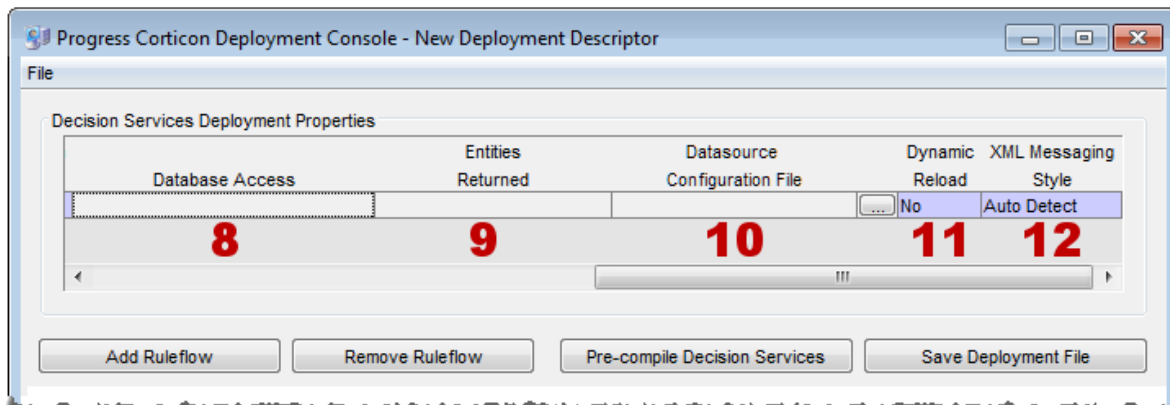


Figure 29: Right Portion of Deployment Console, with Deployment Descriptor File Settings Numbered



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** - 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** - 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. **Datasource Configuration File** - The path and filename of the database access properties file (that was typically created in Corticon Studio) to be used by Corticon Server during runtime database access. Use the adjacent  button to navigate to a Datasource configuration 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 default setting of the `autoloaddir` property is `[CORTICON_WORK_DIR]/cdd` where `[CORTICON_WORK_DIR]` is the absolute path of the work directory, typically `C:/Users/{username}/Progress/Corticon x.x`

You can specify a preferred location in the `brms.properties` file in the form:

```
com.corticon.ccservlet.autoloaddir=path
```

where *path* is your absolute path delimited with forward slashes.

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`. To run the utility, choose **Start > All Programs > Progress > Corticon 5.7 > Corticon 5.7 Command Prompt**, and then type `corticonManagement` to display its usage:

Table 9: usage: corticonManagement

Argument	Description
<code>-c, --compile</code>	Compile a Ruleflow into a Decision Service
<code>-e, --extractDiagnostics</code>	Extract diagnostic data from a log file
<code>-h, --help</code>	Print this message
<code>-m, --multicompile</code>	Compile Ruleflows in the specified input file
<code>-s, --schema</code>	Generate the WSDL/XSD schema for a vocabulary or Ruleflow
<code>-t, --test</code>	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 148 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 10: usage: corticonManagement --compile

Argument	Description
<code>-i, --input file</code>	Required. The source Ruleflow <code>.erf</code> file to be compiled.
<code>-o, --output folder</code>	Required. Explicit path to the output folder. If the folder does not exist, it is created.
<code>-s, --service name</code>	Required. The Decision Service file name. (Do not add the <code>.eds</code> extension, it will be done for you.)

Argument	Description
<code>-v, --version</code>	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 <code>service_vversionMajor_versionMinor.eds</code> .
<code>-e, --edc [R RW]</code>	Required when the Vocabulary has been mapped to a database. Sets the database access mode (read only or read write).
<code>-dj, --dependentjars dependentjar,...</code>	Required when using extensions. Explicit path to JAR files required for this Decision Service.
<code>-ij, --includedjars includedjar,...</code>	Required when using extensions. Explicit path to JAR files (that are specified as <code>dependentjars</code>) 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 11: usage: corticonManagement --test

Argument	Description
<code>-i, --input file</code>	Required. The source Ruletest .ert file to run.
<code>-o, --output file</code>	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.
<code>-a, --all</code>	Required unless <code>--sheet</code> 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.

Argument	Description
<code>-s,--sheet <i>sheet_names</i></code>	Required unless <code>--all</code> 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.
<code>-ll,--loglevel <i>level</i></code>	Sets the log level to the specified level of detail. Defaults to current server log level, typically <code>INFO</code> .
<code>-lp,--logpath <i>path</i></code>	Explicit path to the folder where <code>CcServer.log</code> will be saved. Defaults to the server's current log location, typically <code>[CORTICON_WORK_DIR]/logs</code> .

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 12: usage: corticonManagement --schema

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

Argument	Description
<code>-t, --type [WSDL XSD]</code>	Required. Type of schema to generate.
<code>-u, --url address</code>	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 13: usage: corticonManagement --multicompile

Argument	Description
<code>-i, --input file</code>	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>
      <DecisionServiceName>GroceryStore</DecisionServiceName>
      <RuleflowPath>C:\Corticon\staging\Ruleflows\grocery.erf</RuleflowPath>
      <OutputDirectory>C:\Corticon\staging\DecisionServices</OutputDirectory>
      <OverrideIfExists>true</OverrideIfExists>
      <DatabaseAccessMode></DatabaseAccessMode>
    </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.

Note: Unless you are already in the **Corticon Command Prompt**, choose **Start > All Programs > Progress > Corticon 5.7 > Corticon 5.7 Command Prompt**.

In the **Corticon Command Prompt**, enter Web Console deployment commands in this general format:

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

Commands supported in this utility are as follows, where many commands and options allow either their short form or long form, denoted by double dashes:

corticonWebConsole -help

```
usage: corticonWebConsole
  -application,--application    Perform actions on an application
  -ds,--decisionsservice        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: decisionsservice
  -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
  -datasource,--datasource <configuration file>  Path to datasource
                                                    configuration file
                                                    [optional]
  -dbaccessmode,--dbaccessmode <mode>           Database access mode to
                                                    use [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 (`--datasource`, `--dbaccessmode`, and `--dbreturnmode`) are used when the Decision Service is configured for 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 of the decision
                                                    service to be removed
                                                    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
                        print this message
  -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

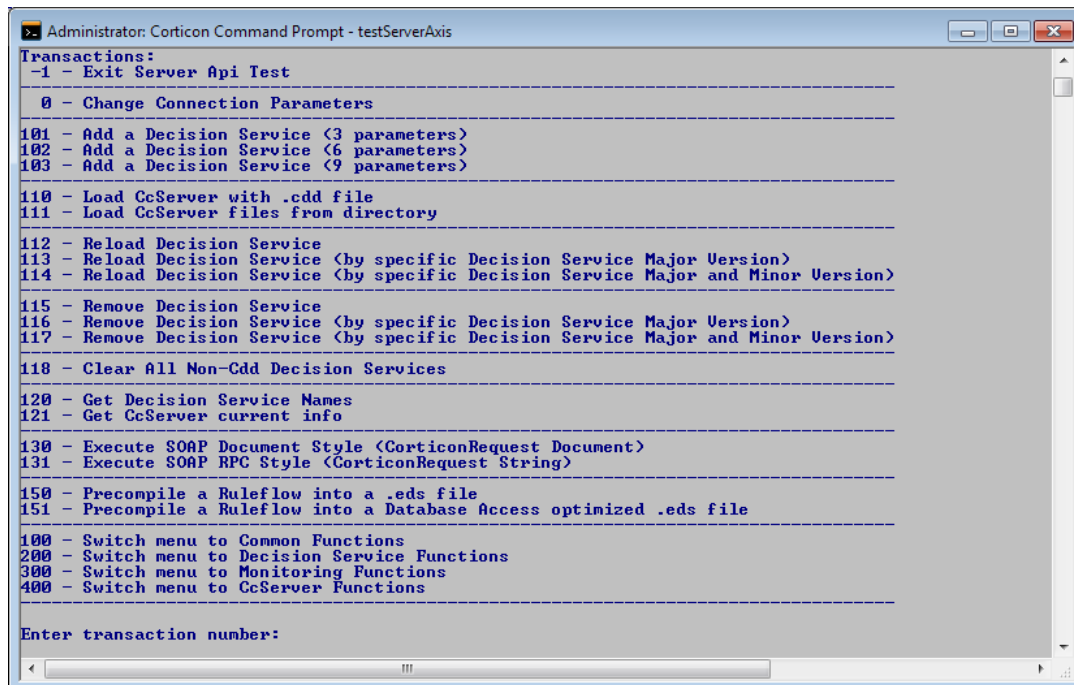
Corticon provides a [Java API](#) that can be used in custom code to compile, deploy, and manage Decision Services. The API can be used in code running an in-process Corticon Server, or can be used to manage a remote Corticon Server through the server's SOAP interface.

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 30: 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 107 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
        (lstrApacheAxisCorticonAdminUrl, 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](#).

Creating custom context URLs on a web server

Corticon web service deployment is preset to a singular web service identity, **axis**. To access that web service, you might enter `http://localhost:8850/axis`. You can choose to run multiple instances of Corticon Server on the same `host:port`, such as `http://localhost:8850/another`. This enables you to use one application server, yet create unique Decision Service URL end points that provide isolation to different sets of Decision Services.

To add another Java WAR file to a Java web server:

This example clones the default WAR file, `axis.war` that was deployed on the embedded Progress Application Server, and then configures and runs the new web archive successfully.

1. Stop the Java web server.
2. In `[CORTICON_WORK_DIR]\pas\server\webapps`, copy `axis.war`, and then paste it there to make a copy.
3. Edit the name `axis - Copy.war` to your preferred `.war` name. Use alphanumeric characters and underscores, and no spaces. For example `Java_UAT.war` for your user acceptance testing instance. The new WAR file will be expanded into its directory structure when you start the application server.
4. To configure settings specific to this new Corticon Server node, expand the WAR file with an archive utility, such as WinZip.
5. Navigate in the new context's file location to `\Server\pas\server\webapps\Java_UAT\WEB-INF\lib\CcConfig.jar`.
6. Extract and edit the `CcServer.properties` file as follows:
 - a. Change the line `com.corticon.ccserver.sandboxDir=%CORTICON_WORK_DIR%/%CORTICON_SETTING%/CcServerSandbox` to `com.corticon.ccserver.sandboxDir=%CORTICON_WORK_DIR%/%CORTICON_SETTING%/Java_UAT/CcServerSandbox`
 - b. Change the line `com.corticon.ccserver.autoloadaddir=%CORTICON_WORK_DIR%/cdd` to `com.corticon.ccserver.autoloadaddir=%CORTICON_WORK_DIR%/Java_UAT/cdd`
 - c. Save the edited file.
7. Extract and edit the `CcCommon.properties` file, and then change the line `logpath=%CORTICON_WORK_DIR%/logs` to `logpath=%CORTICON_WORK_DIR%/Java_UAT/logs`
8. Save both files and replace them in the `CcConfig.jar`.
9. Repackage the entire set of files back into its WAR file.
10. Restart the Java web server.

The two Java server deployments run. Using the Web Console, you can manage the two distinct server context URLs, for example, `myServer:8850/axis` and `myServer:8850/Java_UAT`.

To add another Corticon .NET web service to an IIS server:

This example clones the default .NET web service, `axis` that was installed from a Corticon Server for .NET installation to an IIS server configured for Corticon web services, and then runs the new application successfully.

1. In the **IIS Manager**, on the **Actions** panel, click **Stop**.
2. In the Server .NET's `[CORTICON_HOME]\IIS` folder, copy `install.bat`, and then paste it there to make a copy.

3. Edit the name `install - Copy.bat` to your preferred `.war` name. Use alphanumeric characters and underscores, and no spaces. For example `install_anotherNET.bat`.
4. Edit that file to replace all instances of `axis` with `anotherNET`, and then save the edited file.
5. Run the script.
6. Navigate in the new context to `C:\inetpub\wwwroot\anotherNET\conf`.
7. Edit the `CcServer.properties` file as follows:
 - a. Change the line
`com.corticon.ccserver.sandboxDir=%CORTICON_WORK_DIR%/%CORTICON_SETTING%/CcServerSandbox`
to
`com.corticon.ccserver.sandboxDir=%CORTICON_WORK_DIR%/%CORTICON_SETTING%/AnotherNET/CcServerSandbox`
 - b. Change the line `com.corticon.ccserver.autoloadaddir=%CORTICON_WORK_DIR%/cdd` to
`com.corticon.ccserver.autoloadaddir=%CORTICON_WORK_DIR%/AnotherNET/cdd`
 - c. Save the edited file.
8. Edit the `CcCommon.properties` file as follows:
 - a. Change the line `logpath=%CORTICON_WORK_DIR%/logs` to
`logpath=%CORTICON_WORK_DIR%/AnotherNET/logs`
 - b. Save the edited file.
9. In the **IIS Manager**, perform the same tasks that were performed when `axis` was installed into IIS. In brief:
 - a. Set Access Permissions for the application's directories,
 - b. Convert the new folder into an application.
 - c. Set the application to use the appropriate application pool.

See *"Setting up and updating IIS for .NET Server"* in the *Installation Guide* for more information and links to the various IIS setup articles in the Knowledgebase.
10. In the **IIS Manager**, on the **Actions** panel, click **Start**.

The two .NET server deployments run. Using the Web Console, you can manage the two distinct server context URLs, for example, `myServer:80/axis` and `myServer:80/anotherNET`.

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 99. 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](#)
- [Invoking a Decision Service from a different Decision Service](#)

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/invoke/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.

Setting Service Contract properties

The following properties are settings for Service Contracts that you can apply to your Corticon Server installation by adding the properties and appropriate values as lines in its `brms.properties` file, and then restarting Server.

Controls whether `minOccurs="0"` or `"1"` for Attributes that are marked as mandatory inside the Vocabulary. By default, all mandatory Attributes have `minOccurs="1"`.

Default value is 1

```
com.corticon.deployment.schema.attribute.mandatory.minOccurs=1
```

Controls whether `nillable="true"` or `"false"` for Attributes that are marked as mandatory inside the Vocabulary. By default, all Attributes are set to `nillable="true"`.

Default value is true

```
com.corticon.deployment.schema.attribute.mandatory.nillable=true
```

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 appear 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 value is true.

```
com.corticon.deployment.schema.useChoice=true
```

Add default namespace declaration to the XSD Generation

Default value is true.

```
com.corticon.schemagenerator.addDefaultNamespace=true
```

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 backward 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
```

Specifies whether the XSD and WSDL generators appends 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
```

The property `ensureComplianceWithDotNET` determines whether generated service contracts (WSDL/XSD) are compliant with Microsoft .NET requirements. This property must be set to true when the Corticon Server is deployed inside a Microsoft WCF container.

Default value is false

```
com.corticon.servicecontracts.ensureComplianceWithDotNET_WCF=false
```

For information about related properties

See:

- *"Setting Server execution properties" in the Integration and Deployment Guide's Performance Tuning section.*
- *"Setting Server build properties" in the Integration and Deployment Guide's Performance Tuning section.*
- *"Setting Server properties that are baked into Decision Services" in the Integration and Deployment Guide.*

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 193.

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.

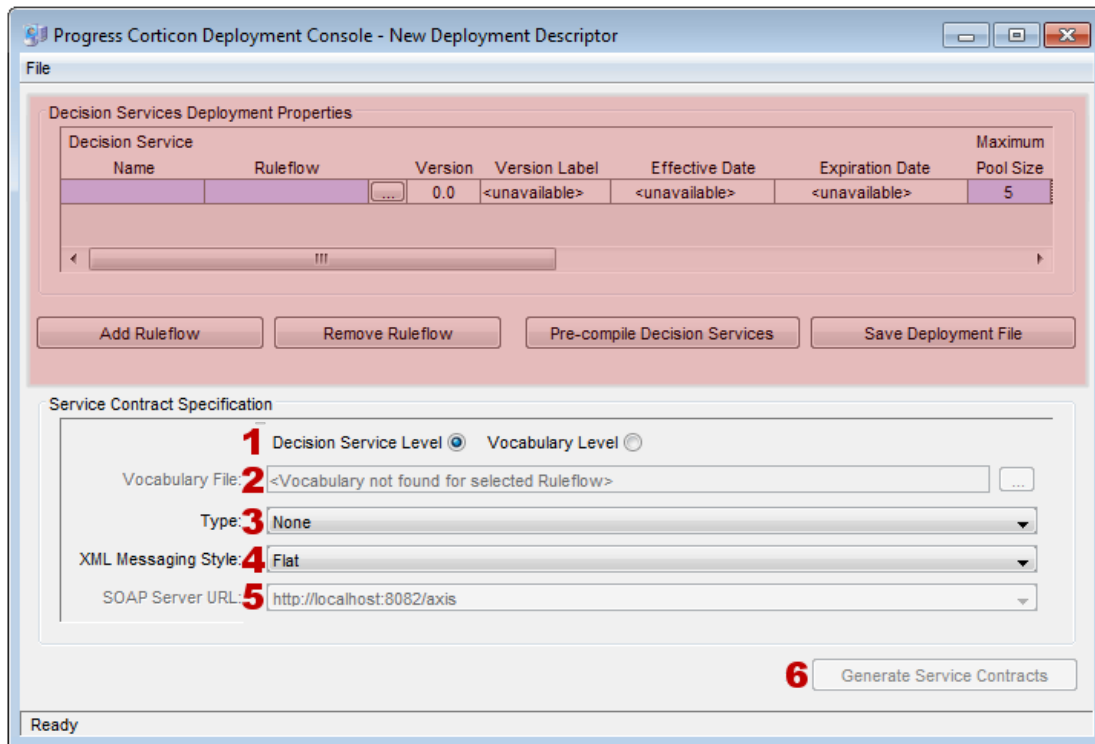
Creating XML service contracts with Corticon Deployment Console

In the topic [Using the Server Deployment Console](#) on page 54, 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 55 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 60 to achieve the result.

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

Figure 31: 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**.

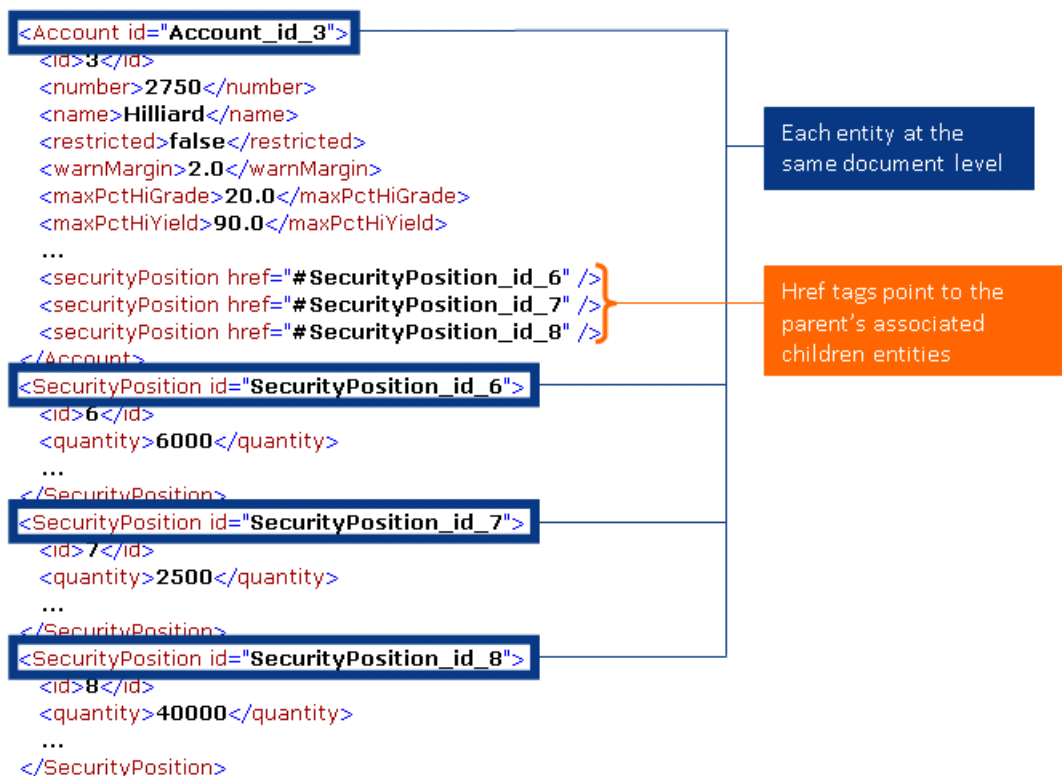
The downside to the Vocabulary-level service contract is its size. Any request message generated from a Vocabulary-level service contract will contain the XML structure for *every term* in the Vocabulary, even if a given Decision Service only requires a small fraction of that structure. Use of a Vocabulary-level 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.

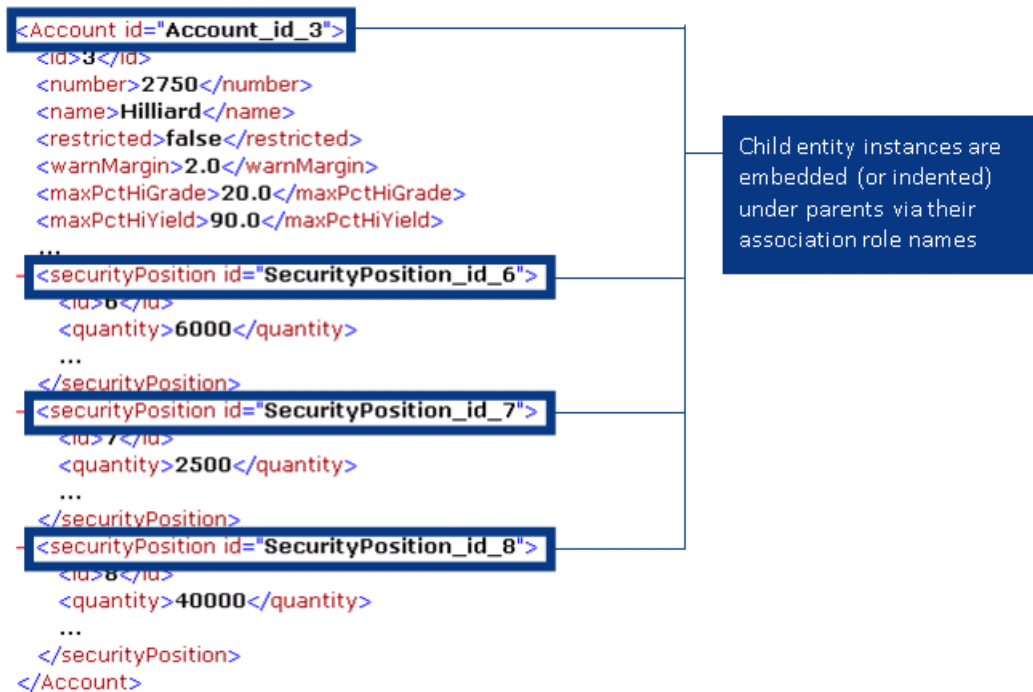
Both styles are illustrated below. Assume a Decision Service uses an `Account` and its associated `SecurityPositions`. In the Flat style, the payload is structured as:

Figure 32: Example of a Flat (FLAT) Message



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

Figure 33: Example of a Hierarchical (HIER) Message



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

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

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

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 as `com.corticon.deployment.soapbindingurl_1`.
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.

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 193, 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 <code>xsi:nil='1'</code> 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, <code><weight></weight></code> or simply <code><weight/></code> . 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 <code>href</code> 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 <code>complexType</code> 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
```

Invoking a Decision Service from a different Decision Service

You can use an extended operator or service call out to invoke a decision service that has its own vocabulary from within another decision service. The called decision service does not need to be on the same server. It can be executed in-process as shown below.

The following code example is an extended operator that uses in-process execution. You need to construct and pass in an XML SOAP message as the input. You also need to parse the response to extract the values of interest and any messages.

```
public class Corticon implements ICcStandAloneExtension{

/**
 * callCorticon makes an in-process call to the specified decision service
 * The decisionServicePath points to a Ruleflow file
 * The XML request conforms to the WSDL for the decision service
 * The return value is the XML response from the decision service
 * @param decisionServiceName
 * @param decisionServicePath
 * @param XMLRequest
 * @return
 */
}
```

```

public static String callCorticon(String decisionServiceName,
    String decisionServicePath,
    String XMLRequest, Boolean enableUpdate)
{
    int minPoolSize = 1;
    int maxPoolSize = 1;
    boolean enableUpdate = true;
    String msgStyle = ICcServer.XML_HIER_STYLE;
    try {
        ICcServer iServer = CcServerFactory.getCcServer();
        if(!iServer.isDecisionServiceDeployed(decisionServiceName))
        {
            iServer.addDecisionService(decisionServiceName,
                decisionServicePath, enableUpdate, minPoolSize,
                maxPoolSize, msgStyle);
        }
        return ( "Corticon Result = " + iServer.execute(XMLRequest));
    }
    catch (Exception e) {return "Corticon call to " +
        decisionServiceName + " failed" + e;
    } //End of try call corticon
} // End of method callCorticon

```

Summary of steps in this example:

1. Create the payload
2. Create the HTTP request
3. Prepare the HTTP POST using Apache HTTP libraries
4. Display the request JSON (for debugging)
5. Invoke Corticon
6. Display the response JSON (for debugging)
7. Extract the results
8. Display debugging information

```

public class restInvocationTest {
    public restInvocationTest() {
    }
    public static void main(String[] args) {
        try {
            executeREST();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    private static void executeREST() throws Exception
    {
        String ccServerURL = "corticon-demo.dyndns.org:8860/.../execute";

        // 1. CREATE PAYLOAD:
        //SQL to get values
        JSONObject applicantJSON = new JSONObject();
        // create a JSON object for the Applicant entity
        applicantJSON.put("skydiver", "false");
        // set its isSkydiver attribute
        applicantJSON.put("age", "25");
        // set its age attribute
        JSONObject metadataJSON = new JSONObject();
        // create a JSON object for metadata for Person
        metadataJSON.put("#type", "Applicant");
        // set its #type attribute to "Applicant"
        applicantJSON.put("__metadata", metadataJSON);
    }
}

```

```
// add the metadata object to the applicant object
JSONArray ccObjects = new JSONArray();
// create an array of JSON objects
ccObjects.put(applicantJSON);
// add the applicant to it
JSONObject ccRequestObj = new JSONObject();
// create a JSON request object
ccRequestObj.put("Objects", ccObjects);
//add DS name
ccRequestObj.put ("name","SkyDiver");
// set its objects attribute to the array of ccObjects

// 2. CREATE HTTP REQUEST
HttpEntity ccRequestEntity = EntityBuilder.create()
    .setContentType(ContentType.APPLICATION_JSON)
    .setText(ccRequestObj.toString()).build();

// 3. PREPARE THE HTTP PPOST Using Apache HTTP libs
HttpClient client = HttpClientBuilder.create().build();
HttpPost post = new HttpPost(ccServerURL);

// 4. DISPLAY THE REQUEST JSON before invoking Corticon
//                                     (only needed for debugging)
// [this is to display the request on the console]
BufferedReader rd = new BufferedReader
    (new InputStreamReader(post.getEntity().getContent()));
StringBuffer result = new StringBuffer();
String line = "";
while ((line = rd.readLine()) != null) {result.append(line);}
String postJSONString = result.toString();
System.out.println("\nccRequestObj.toString()
    :\n"+ccRequestObj.toString()); // Payload

// 5. INVOKE CORTICON
HttpResponse response = client.execute(post);

// 6. DISPLAY THE RESPONSE JSON (only needed for debugging)
rd = new BufferedReader(new InputStreamReader(response.getEntity()
    .getContent()));
result = new StringBuffer();
line = "";
while ((line = rd.readLine()) != null) {result.append(line);}
String responseJSONString = result.toString();

// 7. EXTRACT THE RESULTS: riskRating value from Corticon response JSON
JSONTokener tokenener = new JSONTokener(responseJSONString);
JSONObject respJSON = new JSONObject(tokenener);
JSONArray objects = respJSON.getJSONArray("Objects");
JSONObject messages = respJSON.getJSONObject("Messages");

// Get object values
String riskRating = objects.getJSONObject(0).getString("riskRating");

String age = objects.getJSONObject(0).getString("age");
String isSkydiver = objects.getJSONObject(0).getString("skydiver");

// Get message values
String message = messages.getJSONArray
    ("Message").getJSONObject(0).getString("text");
String severity = messages.getJSONArray
    ("Message").getJSONObject(0).getString("severity");
String entityRef = messages.getJSONArray
    ("Message").getJSONObject(0).getString("entityReference");

// Display results
System.out.println("\nRESULTS -----");
System.out.println("\nRisk: " + riskRating
    + " for age = " + age + " and skydiver = " + isSkydiver);
```

```
System.out.println("Message: " + message);
System.out.println("Severity: " + severity);
System.out.println("Entity Reference: " + entityRef);

// 8. DEBUGGING (only needed for debugging)
System.out.println("\nDEBUGGING -----");
System.out.println("\nccRequestObj.toString():\n"+ccRequestObj.toString());
// Payload
System.out.println("\nccRequestEntity.toString():\n"+
    ccRequestEntity.toString()); // Request entity
System.out.println("\nRequestJSONString:\n" + postJSONString);
// Request JSON
System.out.println("\nResponse:\n" + response.toString());
// Shows just a response summary NOT the response JSON
System.out.println("\nResponseJSONString:\n" + responseJSONString);
// Response
```

You could pass multiple `Person` objects in a single payload but then you need to use the `entityReference` to ensure that the correct message is matched up with the corresponding `Person`. Also if the rules return multiple messages for a single `Person` then you will need to iterate over the messages to extract all of them.

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 the passwords and 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 URLs 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 uses 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 must 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>
```

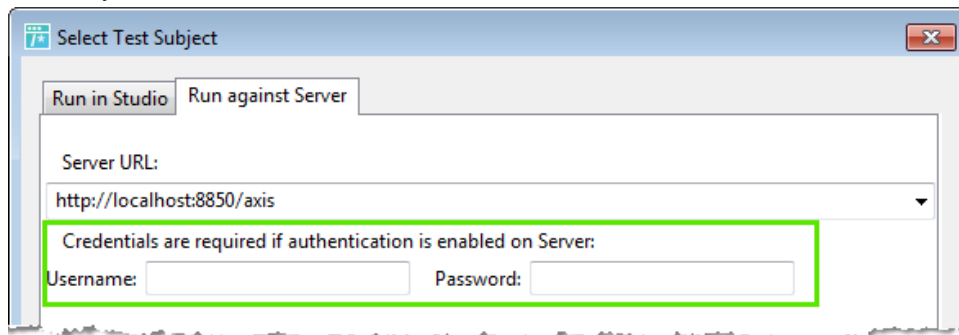
Example: (Two specific role names)

```
<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 add `*` in the XML Element. Even if you have set the constraint's role-name to `*`, you have to define a `<security-role>` role-name with `*`.

Using authentication in Corticon Studio

When a Corticon Studio Ruletest has 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 to connect to the URL on the Server:

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

2. Create an `encodedString` and add it 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:

Authentication

☒ Server Requires Authentication

admin

.....

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

Note: Using LDAP When your business needs require your users to be authenticated through an LDAP server, you can set up LDAP authentication for Web Console users. See the topic *"Using LDAP for Web Console Authentication"* in the *Corticon Web Console Guide*.

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://localhost: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 89.

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 a remote procedure call (RPC) 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</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 is 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 = {"name":"ProcessOrder",
"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",
}]}
*/

JSONObject jsonPayload = new JSONObject();
jsonPayload.put("name", "ProcessOrder");
// Create Order object
JSONObject order = new JSONObject();
order.put("dueDate", "1/1/2008");

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 HTTPS connections.

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.

Note: TLS for PAS requires clients to use Java 8 - The Progress Application Server bundled with Corticon is configured to use TLSv1.2 while the installed HTTPS client classes might have TLSv1.2 disabled, and then attempt to use TLSv1. Using the Java 8 JVM for clients resolves this issue.

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.

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

Corticon supplies a REST API that lets client applications execute against remote Decision Service. The endpoint to this REST API is:

```
http://<base>/axis/corticon
```

where `<base>` is the Corticon Server's IP or DNS-resolvable name and port.

The payload set to the REST API is a JSON object that contains the Decision Service and related data to execute against. The format of the JSON object is:

```
{
  "name": <string>,
  "majorVersion":<number - optional>,
  "minorVersion": <number - optional>,
  "Objects": <JSONArray of JSONObject>
}
```

where `name`, `majorVersion`, and `minorVersion` values specify which Decision Service will be used for the execution, and the `Objects` value is the payload of the data to be processed.

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
{
  "date": "Thu, 07 Jun 2018 19:50:52 GMT",
  "content-encoding": "gzip",
  "vary": "Accept-Encoding",
  "server": "Apache-Coyote/1.1",
  "transfer-encoding": "chunked",
  "content-type": "application/json"
}
```

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

{
  "date": "Thu, 07 Jun 2018 19:56:48 GMT",
  "server": "Apache-Coyote/1.1",
  "connection": "close",
  "error": "CcServer.lookupCcServerPoolForExecution() Decision Service: ProcessedOrder
is not registered. Update failed. (Missing pool manager)",
  "content-length": "0",
  "content-type": null
}
```

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 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 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 105 for more information.

Inside Corticon Server

This section describes how Corticon Server operates. The topics illustrate its enterprise-readiness.

For details, see the following topics:

- [Setting Server startup to auto load CDD files](#)
- [The basic path](#)
- [About Working Memory](#)
- [Ruleflow compilation into an EDS file](#)
- [Batch processing](#)
- [Multi-threading, concurrency reactors, and server pools](#)
- [State](#)
- [Dynamic discovery of new or changed Decision Services](#)
- [Replicas and load balancing](#)
- [Exception handling](#)

Setting Server startup to auto load CDD files

The following property can be set on your Corticon Server installation by adding the property and an appropriate value as a line in its `brms.properties` file, and then restarting Server.

Properties related to "auto loading" CDD files into the CcServer during initialization:

`com.corticon.ccserver.autoloadaddir.enable`

Specifies whether the CcServer will try to auto load CDD files

Default value: true

`com.corticon.ccserver.autoloadaddir`

The directory where the CDD files are located.

Note: This property can be changed using following method, which will override this setting.

`- ICcServer.setDeploymentDescriptorDirectoryPath(String)`

Note: Use forward slashes as path separator. Example: `c:/Program Files/Progress/cdd`

Default value: If the property value 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 container application was started.

```
com.corticon.ccserver.autoloadaddir.enable=true
com.corticon.ccserver.autoloadaddir=%CORTICON_WORK_DIR%/cdd
```

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 `DeploymentDescriptor` 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.

Note:

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

Batch processing

Batch processing enables Corticon Server to fetch chunks of requests, and then pass them in as a set to the Decision Service for processing, dispersed across multiple processing threads to concurrently process the incoming requests. See *"Getting Started with Batch" in the Data Integration Guide*.

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 “Sizes”) 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 determine whether Corticon Server will persist its state inside of the `ServerState.xml`, set the following property in your `brms.properties` file to true or false. By default this feature is turned on. Default value is true.

```
com.corticon.server.serverstate.persistchanges=true
```

To determine whether Corticon Server will initially load the `ServerState.xml` file to restore the Corticon Server to its previous state, set the following property in your `brms.properties` file to true or false. Default value is true.

```
com.corticon.server.serverstate.load=true
```

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 <code>ServerState.xml</code> during operation, and automatically reads it upon restart to restore to the old state.
true	false	Corticon Server maintains <code>ServerState.xml</code> during operation, but does NOT automatically read it upon restart. New Server state upon restart is unaffected by <code>ServerState.xml</code> . This allows a system administrator to manually control state restoration from the <code>ServerState.xml</code> , if preferred.
false	true	Corticon Server attempts to read <code>ServerState.xml</code> upon restart, but finds nothing there. No old state restored.
false	false	no <code>ServerState.xml</code> 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. 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).

To be precise: there are Intervals of time (ms) at which the Server checks for:

- Changes in any cdd loaded to the 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 (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.

Default is 30000 (30 secs)

```
com.corticon.ccserver.dynamicUpdateMonitoringService.serviceIntervals=30000
```

The maintenance thread that checks for these changes at the specified intervals can be shutdown and restarted using:

- `ICcServer.stopDynamicUpdateMonitoringService()`
- `ICcServer.startDynamicUpdateMonitoringService()`

Alternatively, an API call to the Corticon Server can directly load new Decision Services (or sets of Decision Services).

Note: The dynamic update monitor starts automatically by default but can be shut off by setting the property `com.corticon.ccserver.dynamicupdatemonitor.autoactivate` to `false` in your `brms.properties` 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*.

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 39 for a refresher.

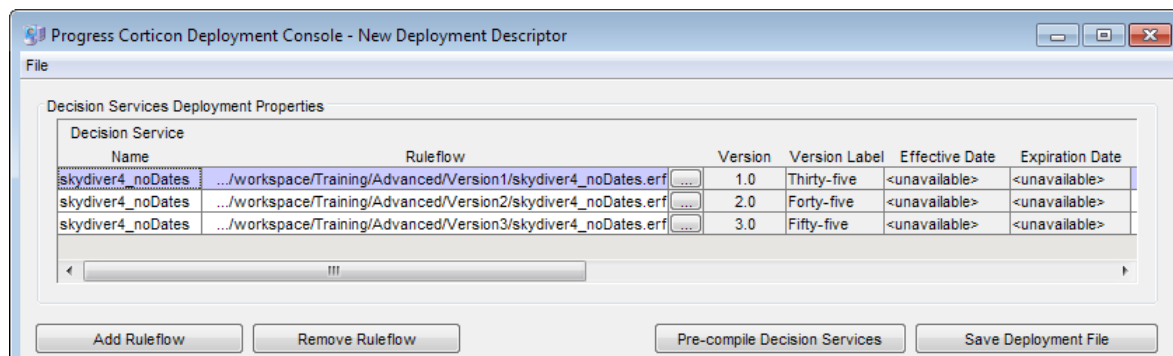
However, the Decision Service Versioning and Effective Dating feature makes an exception to this rule. Decision Services with identical Decision Service Names may be deployed on the same Corticon Server if and only if:

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

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

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

Figure 34: 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.

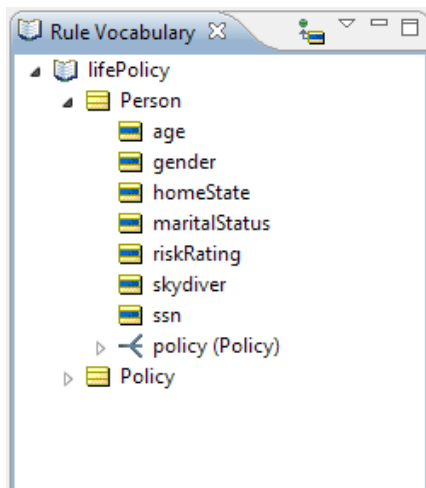
Invoking a Decision Service by version number

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 35: 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 36: Rulesheet skydiver4.ers in folder Version1

The screenshot shows the 'skydiver4.ers' Rulesheet editor. It is divided into two main sections: 'Conditions' and 'Actions'.

Conditions Section:

	0	1	2	3	4
a Person.skydiver		T	-	F	
b Person.age		-	<= 35	> 35	
c					
d					

Actions Section:

	0	1	2	3	4
Post Message(s)		✉	✉	✉	
A Person.riskRating		'high'	'low'	'medium'	
B					
C					

Rule Messages Section:

Ref	ID	Post	Alias	Text
1		Warning	Person	A person that skydives is rated as high risk.
2		Info	Person	A person 35 years old or younger is rated as low risk.
3		Info	Person	A person over 35 years old that does not skydive is rated as medium risk.

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.

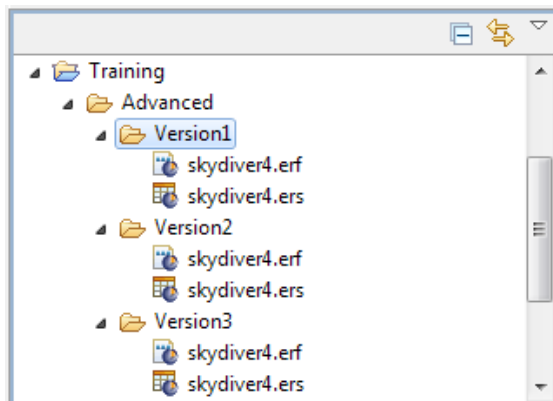
Figure 37: Ruleflow in folder Version1 and set as Version 1.0

The screenshot shows the 'Properties' dialog box for a Ruleflow. The 'Ruleflow' tab is selected, and the 'Rulers & Grid' sub-tab is active. The settings are as follows:

- Rule Vocabulary:** /Training/Advanced/lifePolicy.ecore (with a 'Browse...' button)
- Major Version:** 1
- Minor Version:** 0
- Version Label:** Thirty-five
- Effective Date:** / / (with a 'Time' field set to 0:00 AM and a 'Clear' button)
- Expiration Date:** / / (with a 'Time' field set to 0:00 AM and a 'Clear' button)
- Total Number of Rules:** 3

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:

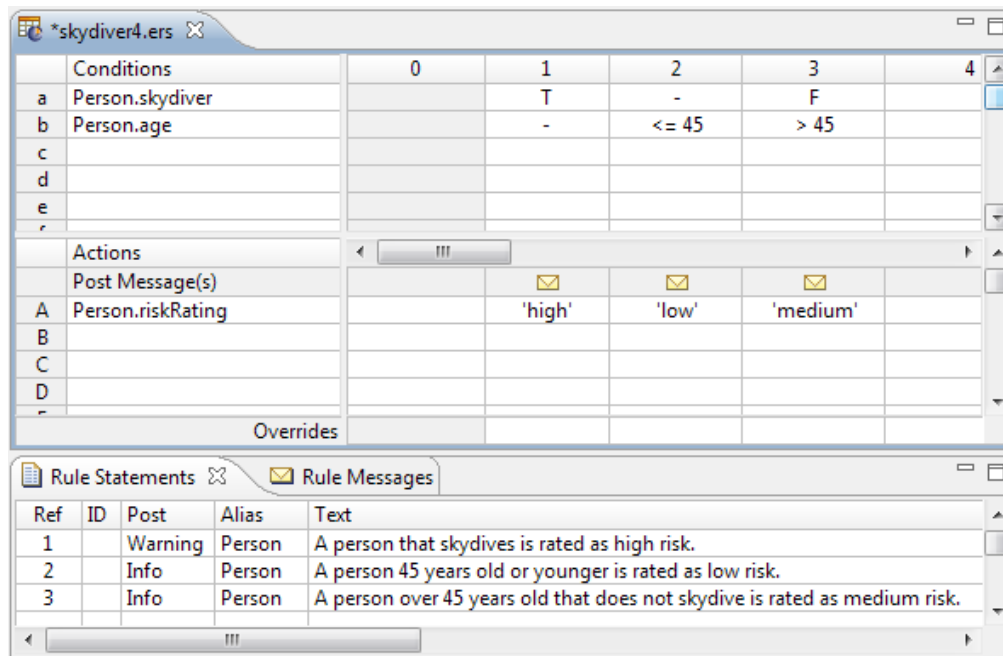
Figure 38: Folders that distinguish three versions



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 39: Rulesheet skydiver4.ers in folder Version2



The screenshot shows the 'skydiver4.ers' rulesheet editor. It has two main sections: 'Conditions' and 'Actions'.

Conditions Section:

	0	1	2	3	4
a Person.skydiver		T	-	F	
b Person.age		-	<= 45	> 45	
c					
d					
e					

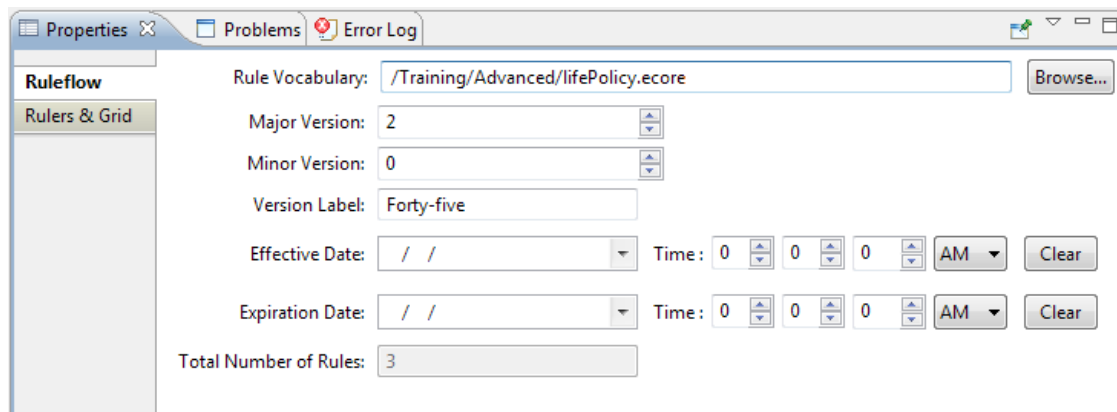
Actions Section:

	0	1	2	3	4
Post Message(s)		✉	✉	✉	
A Person.riskRating		'high'	'low'	'medium'	
B					
C					
D					

Rule Statements Section:

Ref	ID	Post	Alias	Text
1		Warning	Person	A person that skydives is rated as high risk.
2		Info	Person	A person 45 years old or younger is rated as low risk.
3		Info	Person	A person over 45 years old that does not skydive is rated as medium risk.

Figure 40: Ruleflow in folder Version2



The screenshot shows the 'Ruleflow' properties dialog box. It has tabs for 'Properties', 'Problems', and 'Error Log'. The 'Properties' tab is active.

Ruleflow Properties:

- Rule Vocabulary: /Training/Advanced/lifePolicy.ecore (with a 'Browse...' button)
- Major Version: 2
- Minor Version: 0
- Version Label: Forty-five
- Effective Date: / / (with a 'Time' field: 0 : 0 : 0 AM and a 'Clear' button)
- Expiration Date: / / (with a 'Time' field: 0 : 0 : 0 AM and a 'Clear' button)
- Total Number of Rules: 3

And then for Version 3:

Figure 41: Rulesheet skydiver4.ers in folder Version3

Conditions		0	1	2	3	4
a	Person.skydiver		T	-	F	
b	Person.age		-	<= 55	> 55	
c						
d						
e						
Actions						
Post Message(s)			✉	✉	✉	
A	Person.riskRating		'high'	'low'	'medium'	
B						
C						
D						
Overrides						

Ref	ID	Post	Alias	Text
1		Warning	Person	A person that skydives is rated as high risk.
2		Info	Person	A person 55 years old or younger is rated as low risk.
3		Info	Person	A person over 55 years old that does not skydive is rated as medium risk.

Figure 42: Ruleflow in folder Version3

Ruleflow
 Rule Vocabulary: /Training/Advanced/lifePolicy.ecore Browse...
 Major Version: 3
 Minor Version: 0
 Version Label: Fifty-five
 Effective Date: / / Time: 0 0 0 AM Clear
 Expiration Date: / / Time: 0 0 0 AM Clear
 Total Number of Rules: 3

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 193 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
```

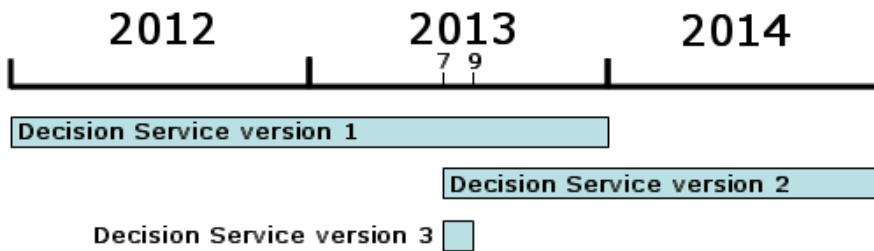
When no Version Number or EffectiveTimestamp is specified, the Server executes against v2.1 (if its Effective/Expiration range is valid). However, when Major Version 1 is passed in without an EffectiveTimestamp specified, the Server executes against v1.2 (if its Effective/Expiration range is valid).

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 43: 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 44: Ruleflow in folder Version1 with dateTime set

The screenshot shows the 'Ruleflow' configuration window for 'Version1'. The 'Rule Vocabulary' is set to '/Training/Advanced/lifePolicy.ecore'. The 'Major Version' is 1, 'Minor Version' is 0, and the 'Version Label' is 'Thirty-five'. The 'Effective Date' is set to '01/01/2012' with a time of '9:00 AM'. The 'Expiration Date' is set to '12/31/2013' with a time of '5:00 PM'. The 'Total Number of Rules' is 3.

We proceed to edit the other two Ruleflows as shown:

Figure 45: Ruleflow in folder Version2 with dateTime set

The screenshot shows the 'Properties' tab of a Ruleflow configuration window. The 'Rule Vocabulary' is set to '/Training/Advanced/lifePolicy.ecore'. The 'Major Version' is 1, 'Minor Version' is 0, and the 'Version Label' is 'Thirty-five'. The 'Effective Date' is 07/01/2013, and the 'Time' is 11:59:59 PM. The 'Expiration Date' is 12/31/2014, and the 'Time' is 11:59:59 PM. The 'Total Number of Rules' is 3.

Figure 46: Ruleflow in folder Version3 with dateTime set

The screenshot shows the 'Properties' tab of a Ruleflow configuration window. The 'Rule Vocabulary' is set to '/Training/Advanced/lifePolicy.ecore'. The 'Major Version' is 3, 'Minor Version' is 0, and the 'Version Label' is 'Fifty-five'. The 'Effective Date' is 07/01/2013, and the 'Time' is 9:00:00 AM. The 'Expiration Date' is 09/30/2013, and the 'Time' is 5:00:00 PM. The 'Total Number of Rules' is 3.

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 193 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 <i>Production or Test</i> 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

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 accepts 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, on Windows::

```
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 130 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 132 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 102 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

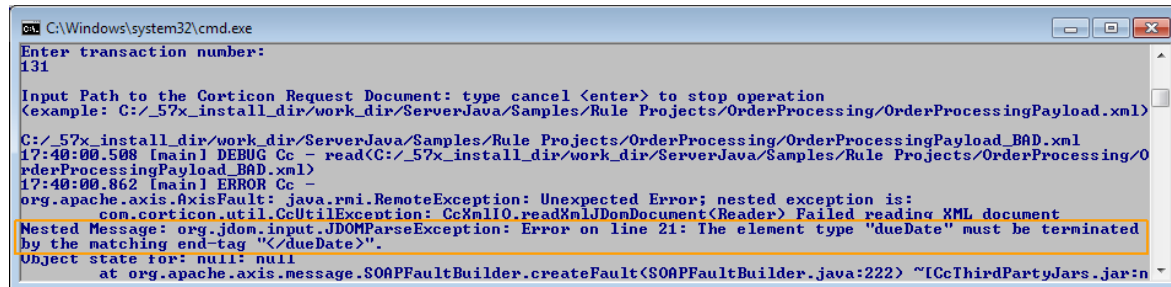
The most common causes of erroneous Corticon Server responses include:

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 47: Server Window Message Highlighting Incorrect SOAP/XML Request Structure



```

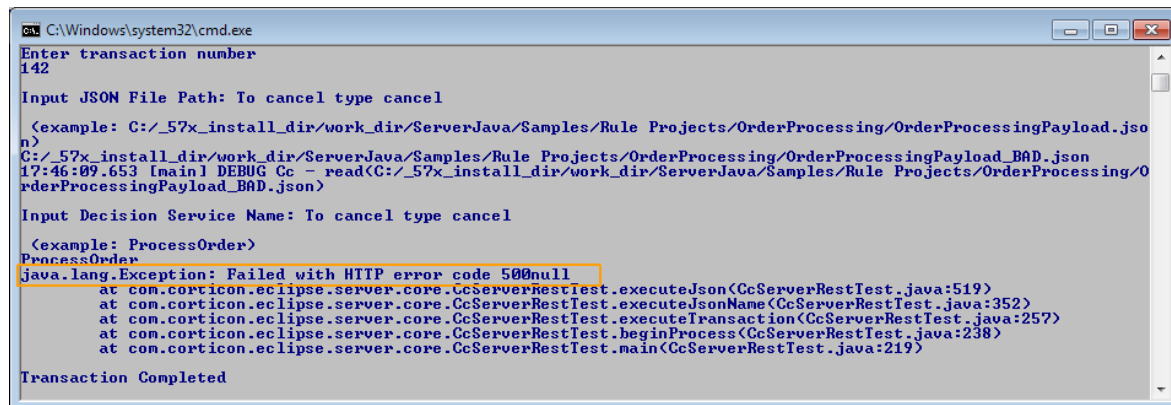
C:\Windows\system32\cmd.exe
Enter transaction number:
131

Input Path to the Corticon Request Document: type cancel <enter> to stop operation
(example: C:/_57x_install_dir/work_dir/ServerJava/Samples/Rule Projects/OrderProcessing/OrderProcessingPayload.xml)
C:/_57x_install_dir/work_dir/ServerJava/Samples/Rule Projects/OrderProcessing/OrderProcessingPayload_BAD.xml
17:40:00.508 [main] DEBUG Cc - read(C:/_57x_install_dir/work_dir/ServerJava/Samples/Rule Projects/OrderProcessing/OrderProcessingPayload_BAD.xml)
17:40:00.862 [main] ERROR Cc -
org.apache.axis.AxisFault: java.rmi.RemoteException: Unexpected Error; nested exception is:
com.corticon.util.CcUtilException: CcXmlIO.readXmlJDomDocument(Reader) Failed reading XML document
Nested Message: org.jdom.input.JDOMParseException: Error on line 21: The element type "dueDate" must be terminated
by the matching end-tag "</dueDate>".
Object state for: null
at org.apache.axis.message.SOAPFaultBuilder.createFault(SOAPFaultBuilder.java:222) ~[CcThirdPartyJars.jar:n

```

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

Figure 48: Server Window Message Highlighting Incorrect JSON Request Structure



```

C:\Windows\system32\cmd.exe
Enter transaction number
142

Input JSON File Path: To cancel type cancel
(example: C:/_57x_install_dir/work_dir/ServerJava/Samples/Rule Projects/OrderProcessing/OrderProcessingPayload.json)
C:/_57x_install_dir/work_dir/ServerJava/Samples/Rule Projects/OrderProcessing/OrderProcessingPayload_BAD.json
17:46:09.653 [main] DEBUG Cc - read(C:/_57x_install_dir/work_dir/ServerJava/Samples/Rule Projects/OrderProcessing/OrderProcessingPayload_BAD.json)

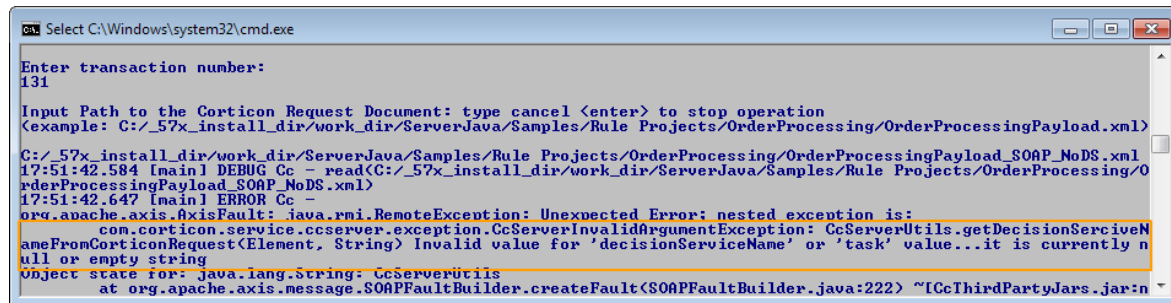
Input Decision Service Name: To cancel type cancel
(example: ProcessOrder)
ProcessOrder
java.lang.Exception: Failed with HTTP error code 500null
at com.corticon.eclipse.server.core.CcServerRestTest.executeJson(CcServerRestTest.java:519)
at com.corticon.eclipse.server.core.CcServerRestTest.executeJsonName(CcServerRestTest.java:352)
at com.corticon.eclipse.server.core.CcServerRestTest.executeTransaction(CcServerRestTest.java:257)
at com.corticon.eclipse.server.core.CcServerRestTest.beginProcess(CcServerRestTest.java:238)
at com.corticon.eclipse.server.core.CcServerRestTest.main(CcServerRestTest.java:219)

Transaction Completed

```

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 49: 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.

Note: To update Corticon Server's license file for Java and .NET deployments, see the task *"Updating your Corticon Server License" in the Corticon Installation Guide*.

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`.

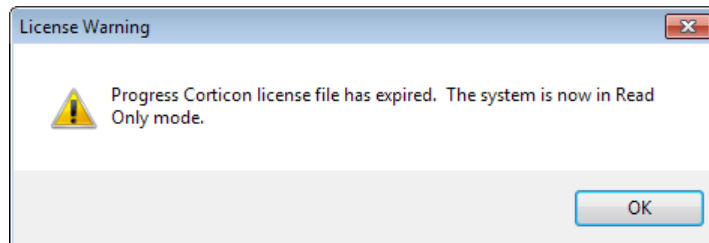
License expired - If your license indicates that it has expired, contact your Progress Corticon representative to obtain an updated license file.

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.
```

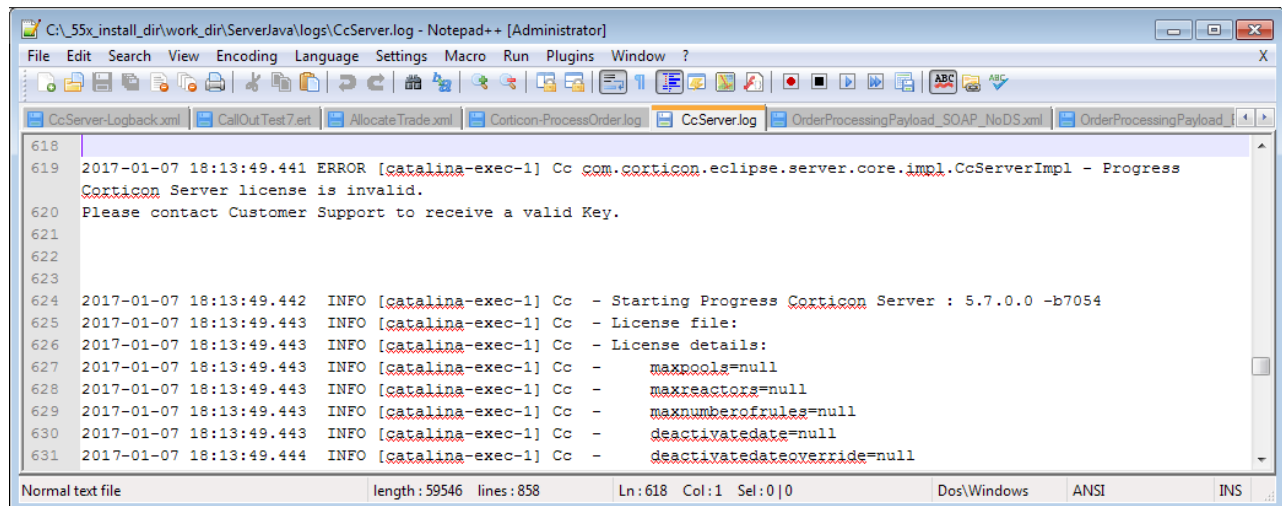
Corticon Studio alerts the user at startup, and then limits functionality:

Figure 50: 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 51: 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.CcServerInvalidArgumentException:
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.

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

Corticon Studio includes many features that help rule authors write efficient rules. Compression - One of the biggest contributors to Decision Service (Ruleflow) performance is the number of rules (columns) in the component Rulesheets, so 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.

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.

Setting Server build properties

The following properties are settings you can apply to your Corticon Server installation by adding the properties and appropriate values as lines in its `brms.properties` file, and then restarting Server.

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
```

Specifies 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 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 Decision Service compile process should dynamically detect the location of the JARs where the Business Objects reside. Primary focus is to incorporate customer Business Objects in the Ant Classpath so that Listener Generation will succeed.

Default value is `true`.

```
com.corticon.server.compile.classpath.include.bos=true
```

For information about related properties

See:

- *"Setting Server execution properties" in the Integration and Deployment Guide's Performance Tuning section.*
- *"Setting Service Contract properties" in the Integration and Deployment Guide's Performance Tuning section.*
- *"Setting Server properties that are baked into Decision Services" in the Integration and Deployment Guide.*

Setting Server execution properties

The following properties are settings you can apply to your Corticon Server installation by adding the properties and appropriate values as lines in its `brms.properties` file, and then restarting Server. These settings impact all the loaded Decision Services on the server.

Specifies whether the Server Execution start and stop times are appended to the CorticonResponse document after `ICcServer.execute(String)`, `ICcServer.execute(Document)`, or `ICcServer.execute(JSONObject)` is performed.

Default value is false

```
com.corticon.ccserver.appendservertime=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, particularly for EDC-enabled applications, 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 N-to-many associations; for N-to-1 associations, the CDOs will always return a direct reference to the "singleton" HashMap.

Default value is false

```
com.corticon.ccserver.cloneAssociationHashSets=false
```

The property `ensureComplianceWithServiceContract` determines whether the returning XML CorticonResponse documents must be valid with respect to the generated XSD/WSDL file. This requires dynamic sorting and results in slower performance.

The `lenientDateTimeFormat` sub-property does the following:

- * If false forces all dateTime values to Zulu format which is the XML standard
- * If true allow any dateTime format supported by Java to use in the payload

Default for `ensureComplianceWithServiceContract` is true (sort)

Default for `lenientDateTimeFormat` is true

```
com.corticon.ccserver.ensureComplianceWithServiceContract=true
```

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.

Support 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
```

Specifies whether the rule engine conducts an integrity check when adding to an existing association. This integrity check ensures that rules do not add redundant associations between the same two entities. Although, this is a rare that occurrence, it is possible. The downside of this integrity check is that Decision Services that create a significant number of new associations can experience a performance degradation. Such Decision Services would require this configuration property to be set to false.

Default value is true.

```
com.corticon.reactor.engine.checkForAssociationDuplicates=true
```

By default, newly created entities are added to the work document. If these entities are not needed in the output they can be created without registering them in the work document. If property value = false, newly created entities via rules will not be registered in the work document.

Default value is true

```
com.corticon.reactor.engine.registerNewEntities=true
```

Option to specify how many variable substitutions could be applied to an ADC PreparedStatement. The restriction on how many PreparedStatement variables is controlled by the Database Driver. Different Databases have different maximums.

Default value is 1000

```
com.corticon.server.adc.preparedstatements.maxvariables=1000
```

Batch Logging parameters that control whether the payload, response, and/or rule messages will be added to an batch execution log file. This will the system to log a batch payload that failed during execution.

Possible values are:

- ALL - log the type whether the execution is successful or not
- VIOLATION - log the type only when there is a Violation rule message in the response.
- NONE - don't log the type

Default value is NONE (for all types)

```
com.corticon.server.batch.logging.payloads=NONE
com.corticon.server.batch.logging.responses=NONE
com.corticon.server.batch.logging.rulemessages=NONE
```

Specifies the timeout setting for when an execution thread waits to get added to the Execution Queue when `com.corticon.server.decisionservice.allocation.enabled=true`.

Default value is 180000 (180000ms = 3 minutes)

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

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
```

Option that lets 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.

```
com.corticon.server.execution.xml.rulemessages.messagesinblock
```

Defines how many messages will be returned in the output

Default value is 5000

```
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 value is 1 (the first block)

```
com.corticon.server.execution.xml.rulemessages.messagesinblock=5000
com.corticon.server.execution.xml.rulemessages.blocknumber=1
```

Option to specify to capture HTTP Headers information

Default value is true

```
com.corticon.server.http=true
```

Provides the option to not add the Entity Name as an `xsi:type` value for those new Entities that are create 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
```

By default, newly created entities in a service call out are added to the work document. If these entities are not needed in the output they can be created without registering them the work document. If property value = false, the SCO created entities are not registered in the work document.

Default value is true

```
com.corticon.services.registerNewSCOEntities=true
```

Related to XML Translation. Base on this value, extra processing will ensure that the Incoming Document's Entities, 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 Entities, 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
```

Option to ignore the `xsi:type` related to Entities/Associations in an XML Payload.

Default value is false

```
com.corticon.xml.xsi.type.ignore=false
```

For information about related properties

See:

- *"Setting Server build properties" in the Integration and Deployment Guide's Performance Tuning section.*
- *"Setting Service Contract properties" in the Integration and Deployment Guide's Performance Tuning section.*
- *"Setting Server properties that are baked into Decision Services" in the Integration and Deployment Guide.*

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_5.7_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 5.7 - 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.
```

```
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. In the `brms.properties` file, enter the following line:

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

That sets the allocation property to true, so that 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 <code>addDecisionService()</code> 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 enable server diagnostic data collection (Data is only posted if service thread is running) Default value is `true`.

```
com.corticon.server.EnableServerDiagnostics=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 130 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,sth=509.6875,shp=356.08567810058594,sex=0,sthq=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,sth=509.6875,shp=371.9812469482422,sex=0,sthq=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 14: Content of a Server diagnostic entry

Item	Description
Diagnostic set ID (<code>id</code>)	Grouping of log lines from a common time slice
Date Time	Timestamp of log entry
Server Total Heap size (<code>sth</code>)	Corticon server JVM total memory in MB
Server Heap size (<code>shp</code>)	Corticon server JVM allocated memory in MB
Server Executing threads (<code>sex</code>)	Current count of threads actively executing Decision Services

Item	Description
Server Threads in Queue (<i>stq</i>)	Count of Decision Service invocations waiting in the queue to be executed
Server Execution Count (<i>sec</i>)	Total number of Decision Services executed since Corticon server startup
Server Failure Count (<i>sfc</i>)	Total execution failure count since Corticon server startup
Server Average Executions time (<i>saex</i>)	Average Decision Service execution time measured across all Decision Services, and provides the average since the Corticon server startup
Server Average Wait Time (<i>sawt</i>)	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}
```

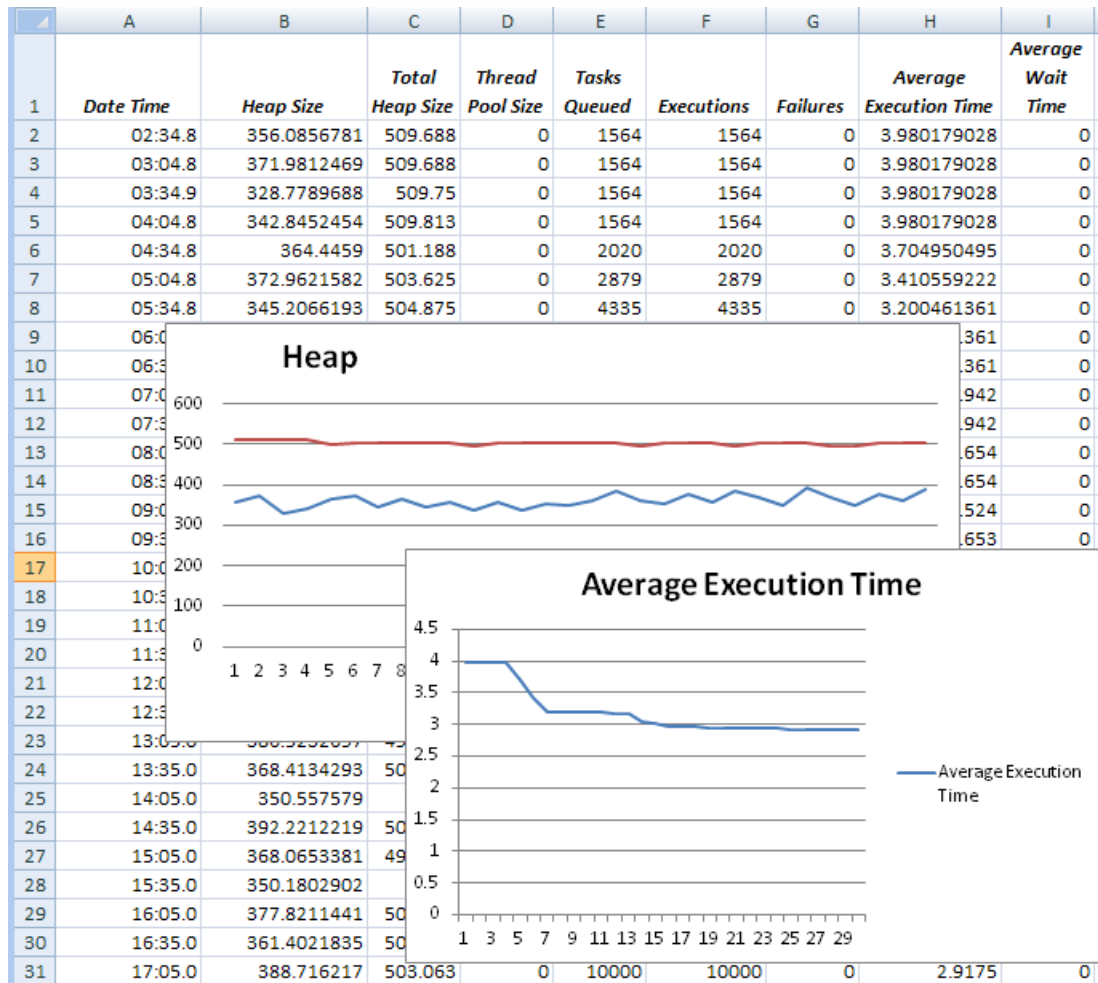
For example:

```
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:



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 15: 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

Item	Description
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}
```

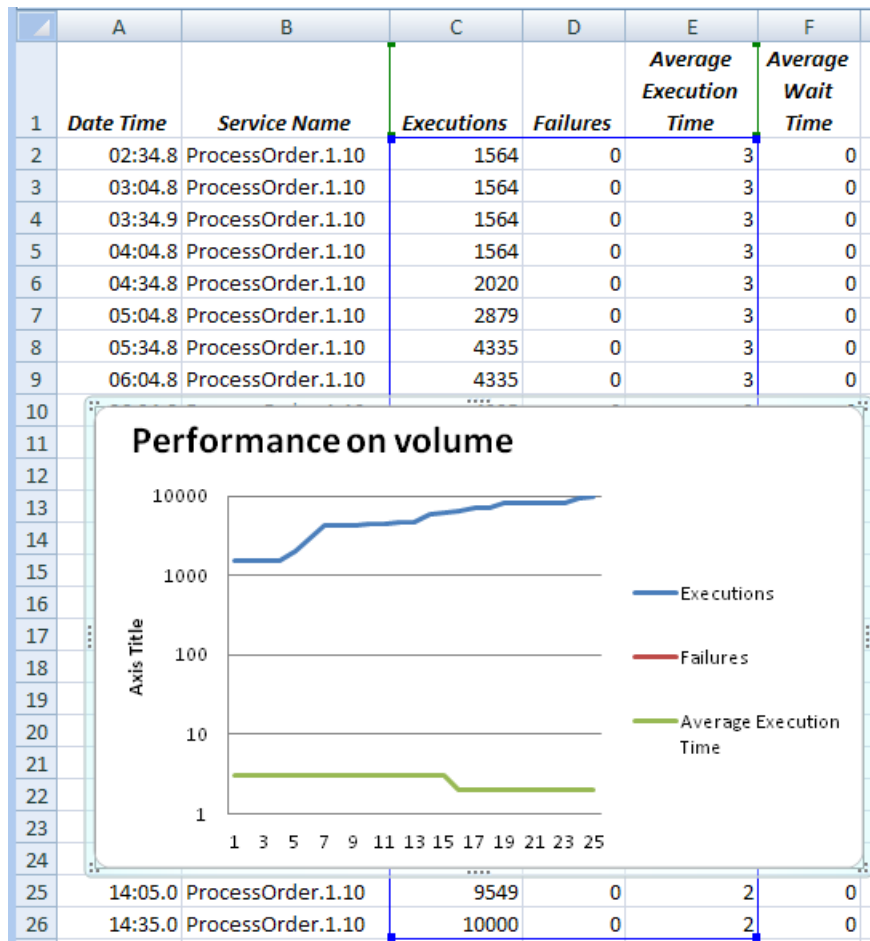
For example:

```
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:



Interpreting diagnostic data

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

- When the number of waiting threads (w_t) 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 (aw_t) 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.

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 Quick Reference 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 Quick Reference 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.

Note: Locale can be set in Studio for running Ruletest Testsheets in Studio and against a server. See *"Setting the locale for a Testsheet" in the Quick Reference Guide*.

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:

- [Handling requests and replies across locales](#)
- [Examples of cross-locale processing](#)
- [Example of cross-locale literal dates](#)
- [Example of requests that cross timezones](#)

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**.

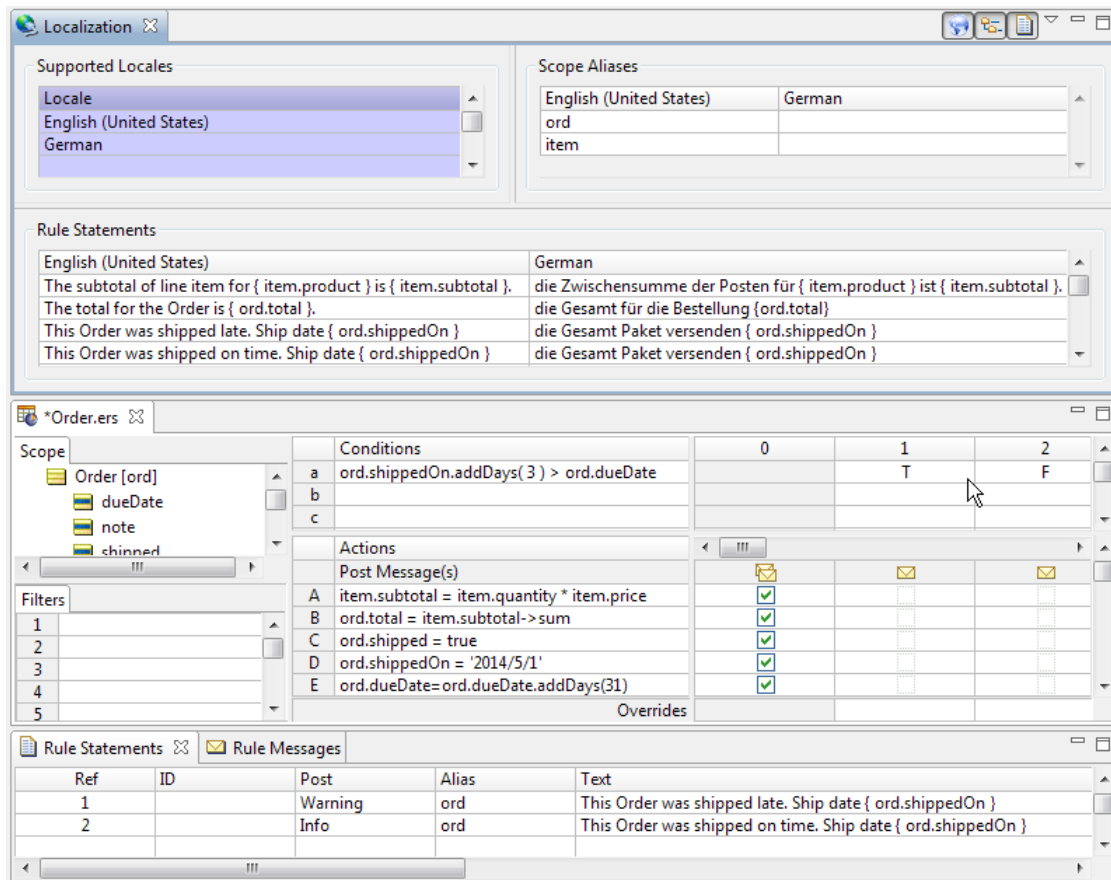
When the message's locale is specified, it is used at rule execution time regardless of the Server's default locale. If the Rulesheet has a matching locale, those rule statement messages are used. Whether or not there is a match, the JVMs functionality enables it to map the input request's decimal delimiters 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>
  ...
</ns1:Messages>
```

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 it 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>
      </myItems>
    </Order>
  </WorkDocuments>
</CorticonRequest>
```



```

    <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 it 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 due 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:Messages>
    </ns1:CorticonResponse>
  </soapenv:Body>
</soapenv:Envelope>

```

```

    </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>
```

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.

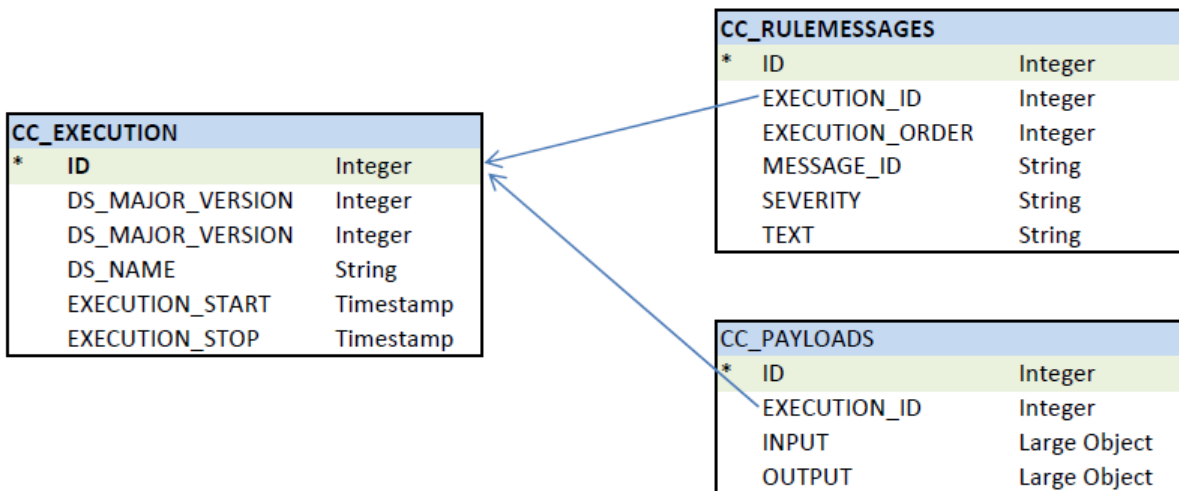
There are several reasons to want to retain payloads and rule messages:

- 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 can be prepended with the metadata of the rule that fired. You enable this feature by setting a line in `brms.properties` as `com.corticon.reactor.rulestatement.metadata=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

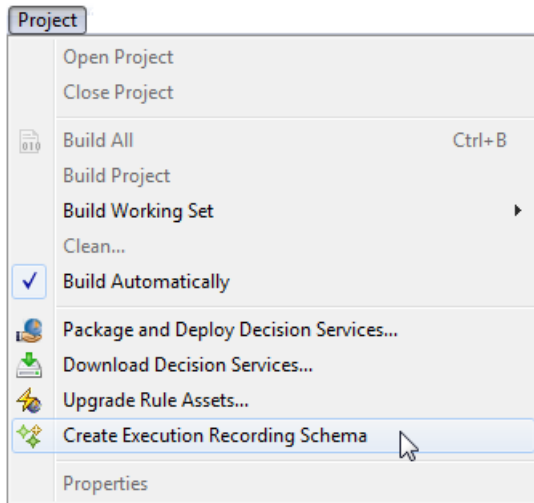
The first step is to setup the schema in your preferred database. Corticon Studio provides a wizard that lets you:

- 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

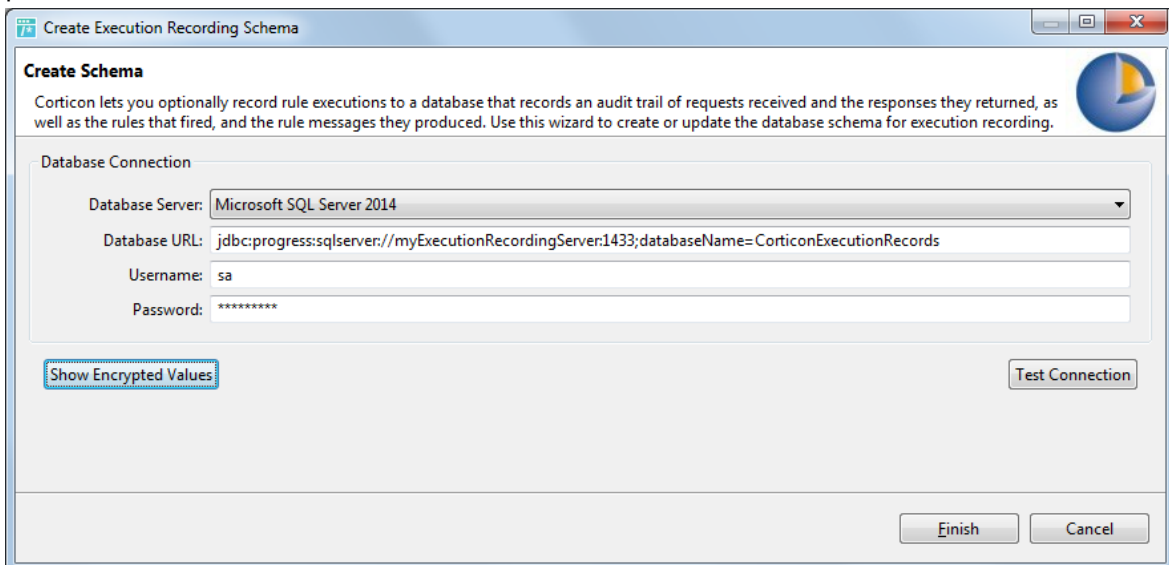
Note: Your database administrator might prefer to create the schema by using (or adapting) the SQL script samples supplied in a Corticon Server installation at `[CORTICON_HOME]/Server/src/sql`. The database connection parameters can also be created entirely in a server's `brms.properties` file, although 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

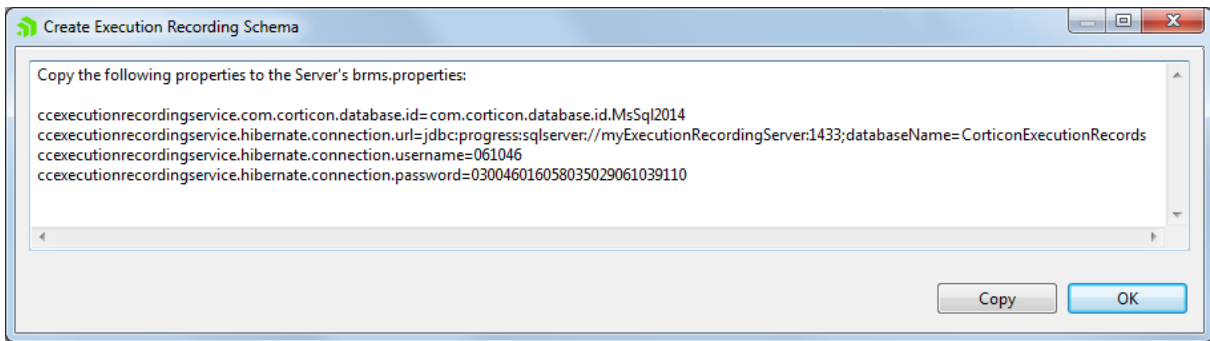
1. In the target database's administrative tool, locate or create the named database instance for recording executions.
2. In Studio, choose the menu command **Project > Create Execution Recording Schema**, as shown:



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.

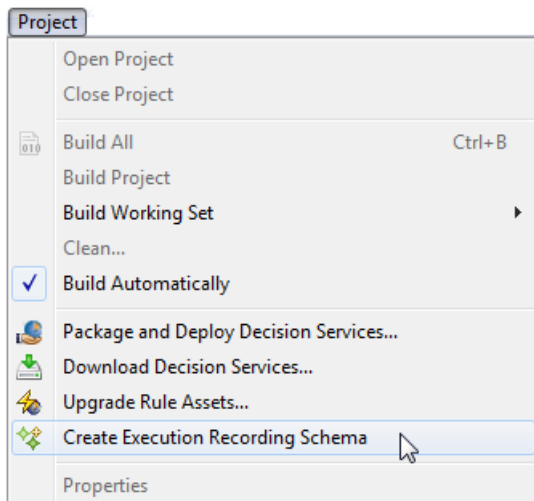
7. Click **Cancel**.

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`.

Settings for Rule Execution Recording

These are the complete rule execution recording settings that you specify in the `brms.properties` file to control and impact rule execution recording behaviors:

Determines how many threads will be allocated to the processing queue that writes to the database Default value is 2

```
com.corticon.server.execution.recording.processors.available
```

This property will define how many executions will be written to the database while wrapped in a single database transaction. The larger the number, the less times the CcServer has to make a connection to the database to establish a transaction. Better performance can be attained by increasing this number. Default value is 10

```
com.corticon.server.execution.recording.executions.per.transaction
```

This property defines how many execution transactions can be queued up before the next transaction is rejected. The executions associated with that transaction will not be written to the database. A log message will be written to the log informing the user that data was not inserted into the database. Default value is 10

```
com.corticon.server.execution.recording.transaction.max.queue.size
```

The following properties are used to setup the default Execution Recording Service.

The Database Id of the Database that the service will use

```
ccexecutionrecordingservice.com.corticon.database.id
```

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

The username to connect to the Database.

```
ccexecutionrecordingservice.hibernate.connection.username
```

The password to connect to the Database

```
ccexecutionrecordingservice.hibernate.connection.password
```

The URL to connect to the Database

```
ccexecutionrecordingservice.hibernate.connection.url
```

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

You can construct and execute JSON request tests by using the Swagger implementation on your server that you access through your browser by entering `http://localhost:8850/axis/swagger`.

The following illustration shows the request section of **Decision Service: Execute Decision Service**.

The screenshot shows the Swagger UI for a service at `localhost:8850/axis/swagger/#/`. The selected endpoint is `POST /execute`, which "Executes a Decision Service".

Implementation Notes: Executes a Decision Service. The Decision Service payload is enclosed in the message body as a JSON string.

Parameters:

Parameter	Value	Description	Parameter Type	Data Type
body	<pre>{ "name": "string", "majorVersion": "string", "minorVersion": "string", "effectiveTimestamp": "string", "Objects": [{}] }</pre>	JSON payload for Decision Service we are trying to execute	body	Model

Parameter content type: `application/json`

Example Value:

```
{
  "name": "string",
  "majorVersion": "string",
  "minorVersion": "string",
  "effectiveTimestamp": "string",
  "Objects": [
    {}
  ]
}
```

The following sections discuss the parameters in JSON/RESTful requests and responses.

About creating a JSON request message for a Decision Service

A JSON request message has a body that describes the parameters for handling the request payload, and the payload.

Parameters of a JSON Request

```
{
  "name": "string",
  "majorVersion": "string",
  "minorVersion": "string",
  "effectiveTimestamp": "string",
  "Objects": [
    {}
  ]
}
```

```
    ]
  }
```

where:

- `name` - the name of the Decision Service -String
- `majorVersion` - Major version number, optional - String, converted to an integer in Corticon
- `minorVersion` - Minor version number, optional - String, converted to an integer in Corticon
- `effectiveTimeStamp` - DateTime of the Decision Service, optional - String
- `Objects` - JSONArray of JSONObject that comprises the payload - String

Structure of JSON payload

A JSON payload is a standardized `JSONObject` that can be passed in to `ICcServer.execute(...)`. The payload contains:

- `"Objects": [<JSONArray>]` where the `JSONArray` must contain `JSONObject` that represent Root Entities of the payload.
- `__metadataRoot` (optional) - An optional Attribute inside the main `JSONObject` that 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 example shows the supported properties:

```
{
  "Objects": [ <JSONArray>
  ],
  "__metadataRoot": {
    "#restrictInfoRuleMessages": "true",
    "#restrictViolationRuleMessages": "true",
    "#restrictResponseToRuleMessagesOnly": "true",
    "#locale": "en-US"
  }
}
```

Root Level Entities

All `JSONObject` inside the `JSONArray` under `Objects` are **Root Level Entities**.

Every name-value in the `JSONObject` maps to a Corticon Vocabulary Entity name on the Root of the payload or as an Association Entity, each of which requires a `__metadata` String attribute with a value of a `JSONObject` that describes the Entity with name-value pairs.

Mandatory:

`#type` : The Entity type as defined in the Vocabulary.

Optional:

`#id`: A unique String value for each Entity.

Note:

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 JSON Datatypes that can be used as a *value* are `String`, `Boolean`, `Double`, `Int`, `Long`.

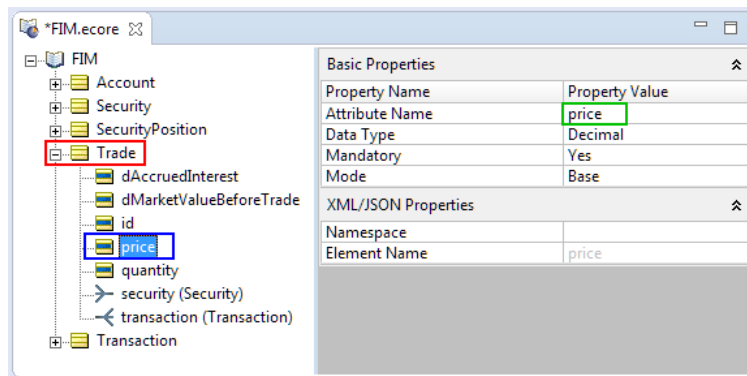
For a `Date` value, use a `String` to represent the `Date`, which will be converted into a proper `Date` object for rule processing.

The *value* associated with a *name* does not 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. 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's `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. Note that:

- The *value* associated with *name* can be either a `JSONObject` or a `JSONArray` (of other `JSONObject`s).
- If the original value was a `JSONObject`, a `JSONArray` could 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 `JSONObject`s can be associated with that *name*.
- 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.
- In an 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 used 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:

In the following example, there is one Association, `transaction` that has corresponding JSON Object as a value. It is an Embedded Association -- 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"
    }
  }
]

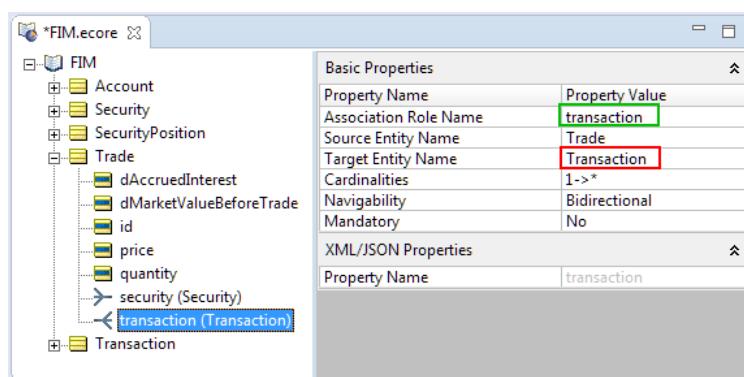
```

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"
    }
  }],
  {
    "maxPctHiYield": "35.000000",
    "_metadata": {
      "#id": "Transaction_id_1",
      "#type": "Transaction"
    }
  }
]

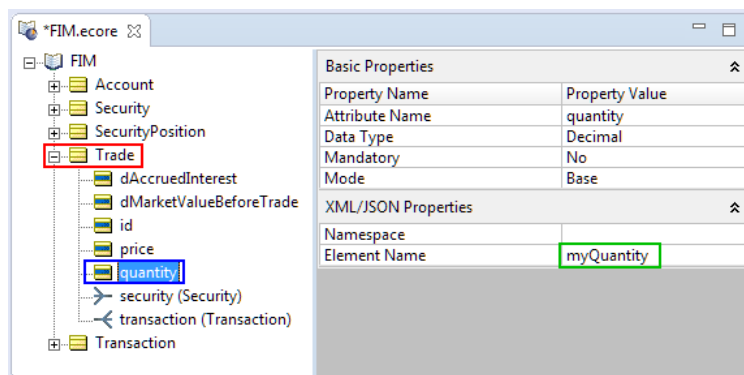
```



XML Element Name overrides for Attributes and Association names

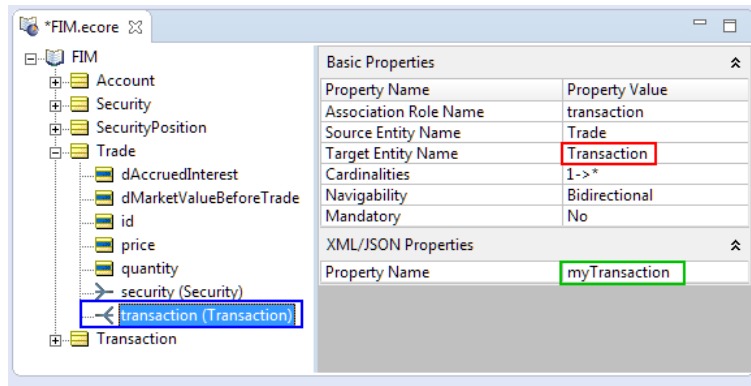
JSON Entity Attribute names are first matched against XML Name overrides, which are defined in the Vocabulary Attribute. If no XML Element Name is defined, then JSON Entity Attribute names are matched directly against the Vocabulary Attribute name.

```
{
  "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

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"
              }
            }
          ],
          "_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

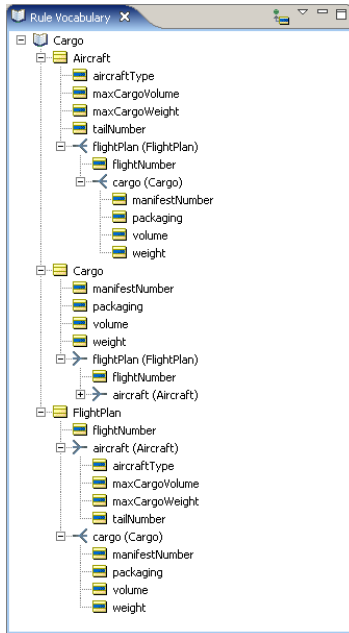
Your Corticon Server installation implements Swagger. From your browser at <http://localhost:8850/axis/swagger/> you can use **Execute Decision Service** to set up the `dsname` and a sample request's JSON Objects to run against a deployed Decision Service. See *"Swagger" in the Integration and Deployment Guide* for more information.

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>
```

Sample client applications

Corticon Server installations now include several sample applications that call a Decision Service. While the functionality of the samples is substantially identical, the contrast of SOAP and REST written in various languages -- Java, C#, JavaScript, Python -- provides developers a solid base to meet their project's client requirements. The source samples are heavily commented. The samples are located in a Corticon Server installation's `[CORTICON_WORK_DIR]\Samples\Clients` directory.

In that folder, the traditional comprehensive API tests are still included:

```
REST\CcServerRestTest.java
SOAP\CcServerApiTest.java
Deployment\CcdeployApiTest.java
```

The sample applications that call a Decision Service and their common sample payload are:

```
C-sharp\RESTClient
C-sharp\SOAPClient

Java\RESTClient
Java\SOAPClient

JavaScript\RESTClient

Python\RESTClient

Data
  OrderProcessingPayload.json
  OrderProcessingPayload.xml
```

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](#)
- [Examples of service contract reports generated from a Ruleflow](#)
- [Extended service contracts](#)
- [Extended datatypes](#)

Examples of XSD and WSDLs available in the Deployment Console

Messaging style	Type	Level	Style
Vocabulary-level XML schema, FLAT XML	XSD	Vocabulary	Flat
Vocabulary-level XML schema, HIER XML	XSD	Vocabulary	Hierarchical
Decision-service-level XML schema, FLAT XML	XSD	Decision Service	Flat
Decision-service-level XML schema, HIER XML	XSD	Decision Service	Hierarchical
Vocabulary-level WSDL, FLAT XML	WSDL	Vocabulary	Flat
Vocabulary-level WSDL, HIER XML	WSDL	Vocabulary	Hierarchical
Decision-service-level WSDL, FLAT XML	WSDL	Decision Service	Flat
Decision-service-level WSDL, HIER XML	WSDL	Decision Service	Hierarchical

Vocabulary-level XML schema, FLAT XML messaging style

This topic defines and annotates the FLAT Vocabulary-level XSD in the following sections:

- [Vocabulary-Level WSDL, FLAT XML Messaging Style](#)
- [Header](#)
- [CorticonRequestType and CorticonResponseType](#)
- [WorkDocumentsType](#)
- [MessagesType](#)
- [VocabularyEntityNameType](#)
- [VocabularyAttributeNameTypes](#)

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">
    <soap:address
location="http://localhost:8850/axis/services/Corticon" />
  </port>
</service>
</definitions>

```

Header

```

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

```

for details on <namespace> definition, see [XML Namespace Mapping](#)

CorticonRequestType and CorticonResponseType

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

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

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

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

Each CorticonRequestType must contain one WorkDocuments element:

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

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

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

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

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

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

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

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

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

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

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

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

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

Same as attribute in CorticonRequest.

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

Same as attribute in CorticonRequest.

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

WorkDocumentsType

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

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

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

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

In a Vocabulary-level XSD, a `WorkDocumentsType` element contains all of the entities from the Vocabulary file specified in the Deployment Console. 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>
```

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 the property `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. Default value is `true` (ensure compliance and perform the sorting, if necessary)
- `false`, the Server may return elements in any order. Consuming applications should be designed accordingly. This setting results in slightly better Server performance.

VocabularyEntityType

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

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

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

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

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

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

VocabularyAttributeNameTypes

Every attribute in a Corticon Vocabulary 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.

Vocabulary-level XML schema, HIER XML messaging style

This section formally defines and annotates the HIER Vocabulary-level XSD.

As many sections are identical from one type to another, the following links connect to the relevant sections in [Vocabulary-level XML schema, FLAT XML messaging style](#) on page 194, with variances noted:

- [Header](#)
- [CorticonRequestType and CorticonResponseType](#)
- [WorkDocumentsType](#) - One line in this section differs from the FLAT version. This attribute value indicates the HIER XML Messaging Style selected in the Deployment Console:

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

- [MessagesType](#)
- [VocabularyAttributeNameTypes](#)

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.

As many sections are identical from one type to another, the following links connect to the relevant sections in [Vocabulary-level XML schema, FLAT XML messaging style](#) on page 194, with variances noted:

- [Header](#)

- [CorticonRequestType and CorticonResponseType](#) - This section of the XSD is identical to the Vocabulary-level FLAT version 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`.

- [WorkDocumentsType](#) - One line in this section differs from the FLAT version. This attribute value indicates the HIER XML Messaging Style selected in the Deployment Console:

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

- [MessagesType](#)
- [VocabularyEntityNameType and VocabularyAttributeNameTypes](#) - The *structure* of this section of the XSD is identical to the Vocabulary-level FLAT version. 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.

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.

As many sections are identical from one type to another, the following links connect to the relevant sections in [Vocabulary-level XML schema, FLAT XML messaging style](#) on page 194, with variances noted:

- [Decision-Service-Level XSD, HIER XML Messaging Style](#) - See below.
- [Header](#)
- [CorticonRequestType and CorticonResponseType](#)
- [WorkDocumentsType](#)
- [MessagesType](#)
- [VocabularyEntityNameType and VocabularyAttributeNameTypes](#) - The *structure* of this section of the XSD is identical to the Vocabulary-level HIER version. 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.

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>
  </xsd:complexType>

```

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>
  </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>

```

Vocabulary-level WSDL, FLAT XML messaging style

This topic defines and annotates the Vocabulary-level WSDL, FLAT XML in the following sections:

- [SOAP Envelope](#)
- [Types](#)
- [Messages](#)
- [PortType](#)
- [Binding](#)
- [Service](#)

SOAP Envelope

```
<?xml version="1.0" encoding="UTF-8"?>
  <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="urn:<namespace>" xmlns:cc="urn:<namespace>"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    targetNamespace="urn:<namespace>">
```

for details on <namespace> definition, see [XML Namespace Mapping](#)

Types

```
<types>
  <xsd:schema xmlns:tns="urn:Corticon" targetNamespace="urn:<namespace>"
    elementFormDefault="qualified">
```

This <type> section contains the entire Vocabulary-level XSD, FLAT-style service contract, minus the XSD Header section...

or details on <namespace> definition, see [XML Namespace Mapping](#)

```
</types>
```

Messages

The SOAP service supports two messages, each with a single argument. See portType

```
<message name="CorticonRequestIn">
  <part name="parameters" element="cc:CorticonRequest" />
</message>
<message name="CorticonResponseOut">
  <part name="parameters" element="cc:CorticonResponse" />
</message>
```

PortType

```
<portType name="VocabularyNameDecisionServiceSoap">
```

Indicates service operation: one message in and one message out...

```
  <operation name="processRequest">
    <input message="tns:CorticonRequestIn" />
    <output message="tns:CorticonResponseOut" />
  </operation>
</portType>
```

Binding

Use HTTP transport for SOAP operation defined in <portType>

```
<binding name="VocabularyNameDecisionServiceSoap" type="tns:
  VocabularyNameDecisionServiceSoap">
```

All WSDLs generated by the Deployment Console use Document-style messaging:

```
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
    <operation name="processRequest">
```

Identifies the SOAP binding of the Decision Service:

```
<soap:operation soapAction="urn:Corticon" style="document" />
  <input>
    <soap:body use="literal" namespace="urn:Corticon" />
  </input>
  <output>
    <soap:body use="literal" namespace="urn:Corticon" />
  </output>
</operation>
</binding>
```

Service

```
<service name="VocabularyNameDecisionService">

  <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>
```

Vocabulary-level WSDL, HIER XML messaging style

As many sections are identical from one type to another, the following links connect to the relevant sections in [Vocabulary-level WSDL, FLAT XML messaging style](#) on page 204, with variances noted:

- [SOAP Envelope](#)
- [Types](#) - See below.
- [Messages](#)
- [PortType](#)
- [Binding](#)
- [Service](#)

Types in Vocabulary-level WSDL, HIER XML messaging style

```
<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>
```

Decision-service-level WSDL, FLAT XML messaging style

As many sections are identical from one type to another, the following links connect to the relevant sections in [Vocabulary-level WSDL, FLAT XML messaging style](#) on page 204, with variances noted:

- [SOAP Envelope](#)
- [Types](#) - See below.
- [Messages](#)
- [PortType](#) - Note that the line `<portType name=DecisionServiceNameSoap>` is different
- [Binding](#) - Note that the line `<binding name=DecisionServiceNameSoap type=tns:DecisionServiceNameSoap>` is different
- [Service](#) - Note that the following lines are different:

```
<service name=DecisionServiceName>
<port name=DecisionServiceNameSoap binding=tns:DecisionServiceNameSoap>
```

Types in Decision-service-level WSDL, FLAT XML messaging style

```
<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>
```

Decision-service-level WSDL, HIER XML messaging style

As many sections are identical from one type to another, the following links connect to the relevant sections in [Vocabulary-level WSDL, FLAT XML messaging style](#) on page 204, with variances noted:

- [SOAP Envelope](#)
- [Types](#) - See below.
- [Messages](#)
- [PortType](#)
- [Binding](#) on page 205
- [Service](#)

Types in Decision-service-level WSDL, HIER XML messaging style

```
<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](#)

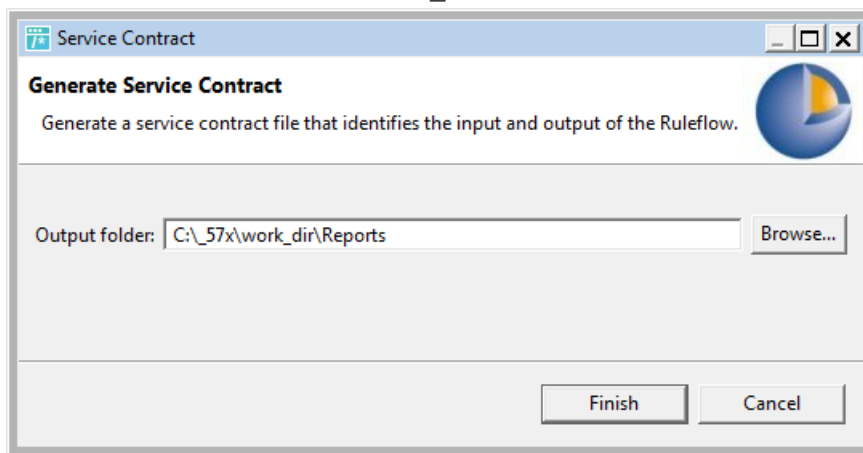
```
</types>
```

Examples of service contract reports generated from a Ruleflow

When you choose the Ruleflow menu command **Service Contract**, a document is produced based on the *RuleflowFileName*:

```
RuleflowFileName_ServiceContract.csv
```

at the default Studio location `[WORK_DIR]\Reports`. For example:



This file provides the basis for defining a JSON/REST service contract.

Controlling date and time format masks

You can specify date and time formats that you prefer in a generated service contract by setting properties in the `brms.properties` file, as shown:

```
# Properties for masking attributes in a Service Contract. Default values are:
#   com.corticon.serviceContract.date.masking=MM/dd/yy
#   com.corticon.serviceContract.dateTime.masking=MM/dd/yy h:mm:ss a
#   com.corticon.serviceContract.time.masking=h:mm:ss a
#
#com.corticon.serviceContract.date.masking=
#com.corticon.serviceContract.dateTime.masking=
#com.corticon.serviceContract.time.masking=
```

Uncomment a masking property you want to set, and then provide an appropriate mask value.

Guidelines for REST/JSON Service Contracts

When you generate a Service Contract from a Ruleflow, the output is a CSV file that provides the guidelines for a REST/JSON Service Contract.

For example, the CSV file for the Tutorial's `tutorial_example.erf` looks like this:

```
Service Contract for Ruleflow

,,Datatypes,Masking,Input/Output

Cargo,,Cargo,,I/O
,needsRefrigeration,Boolean,,I
,volume,Integer,,I
,container,containerType{String},CDT,O
,weight,Integer,,I

Custom Data Types(CDT)
containerType,standard,oversize,heavyweight,reefer,
```

When `tutorial_example_ServiceContract.csv` is opened in Excel, the CSV file looks like this:

	A	B	C	D	E
1	Service Contract for Ruleflow				
2					
3			Datatypes	Masking	Input/Output
4					
5	Cargo		Cargo		I/O
6		needsRefrigeration	Boolean		I
7		volume	Integer		I
8		container	containerType{String}	CDT	O
9		weight	Integer		I
10					
11					
12	Custom Data Types(CDT)				
13	containerType	standard	oversize	heavyweight	reefer

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` file.

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 `complexType`s 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>
```

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](#).

REST Management API

Overview

The Corticon REST Management API provides several REST methods for management of Corticon Server and Decision Service deployment.

Note: JSON datatypes require double-quotes only on String values. As JSON does not have a date datatype, Corticon treats time values as Strings. When the other datatypes -- Booleans and numbers -- are in quotes, Corticon interprets the datatype from its Vocabulary properties and removes the quotes. Output will conform to the JSON syntax.

Common REST request/response Types

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 in the REST Management API

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 this exception
                  (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.

Using the REST API Swagger documentation

Many of the execute and management REST API methods in a Corticon Server WAR file can be exposed in Swagger, the popular OpenAPI Specification tool for describing, producing, consuming, and visualizing RESTful Web services. Swagger is built-in to every Corticon Server WAR file.

You can use Swagger through BASIC authentication and HTTPS connections.

Note: Get more information about Swagger at swagger.io

Accessing Swagger on a Corticon Server for Java

To access Swagger and the exposed RESTful methods, enter the following URL into a browser on a machine where Corticon Server is installed and running:

```
http://localhost:port/context/swagger
```

where the URL for typical local installation using HTTP is:

```
http://localhost:8850/axis/swagger
```

Adjusting Swagger access for custom ports and contexts

If you want to specify a preferred port or a context other than `axis`, you need to adjust some Swagger configurations.

To use a different port, for example for secure HTTP on port 8851:

1. Edit the file:

```
[CORTICON_SERVER_WORK_DIR]\pas\server\webapps\axis\swagger\index.html
```

2. Edit the `url` line to:

```
url = "https://localhost:8851/axis/corticon/swagger.json";
```

To use a different context, for example `Java_UAT`:

1. Stop the server.

2. Edit the file:

```
[CORTICON_SERVER_WORK_DIR]\pas\server\webapps\axis\swagger\index.html
```

3. Edit the `url` line to:

```
url = "https://localhost:8851/Java_UAT/corticon/swagger.json";
```

4. Edit the file:

```
[CORTICON_SERVER_WORK_DIR]\pas\server\webapps\web.xml
```

5. Edit the `swagger.api.basepath` parameter value to:

```
<param-value>/Java_UAT/corticon</param-value>
```

6. Restart the server.

Disabling Swagger on a Server

To disable Swagger on a Server, edit its `web.xml` file to add comment markers as shown:

```
<servlet>
  <servlet-name>Corticon REST Service</servlet-name>
  <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>

  <init-param>
    <param-name>javax.ws.rs.Application</param-name>
    <param-value>com.corticon.eclipse.rest.CorticonRestApplication</param-value>
  </init-param>
  <!-- Swagger documentation for the REST APIs -->
  <!-- disable Swagger
    <init-param>
      <param-name>com.sun.jersey.config.property.packages</param-name>
      <param-value>io.swagger.jaxrs.json;com.corticon.eclipse.rest;
                  io.swagger.jaxrs.listing</param-value>
    </init-param>
  -->
  <!-- End of Swagger documentation for the REST APIs -->
```

and

```
  <!-- Swagger Rest API documentation -->
  <!-- disable Swagger
    <servlet>
      <servlet-name>DefaultJaxrsConfig</servlet-name>
      <servlet-class>io.swagger.jaxrs.config.DefaultJaxrsConfig</servlet-class>
      <init-param>
        <param-name>api.version</param-name>
        <param-value>1.1.8</param-value>
      </init-param>
      <!-- Change this if you deployed you server in a different location -->
      <init-param>
        <param-name>swagger.api.basepath</param-name>
        <param-value>/axis/corticon</param-value>
      </init-param>
      <load-on-startup>2</load-on-startup>
    </servlet>
  -->
```

When you restart the server, access to Swagger on the Server is disabled. Clear the comment delimiters to again enable Swagger on the server.

Accessing the Vocabulary metadata of a Decision Service

There is a REST API for retrieving vocabulary metadata from a deployed Decision Service. This is useful for integrating Corticon with other applications that need to format REST or SOAP calls to a Decision Service.

Structure of a request

To retrieve vocabulary metadata, make an HTTP GET request to the `getVocabularyMetadata` endpoint specifying the Decision Service name and version as the following URL parameters:

- name=*Decision Service name*
- majorVersion= *Major version of the Decision Service*
- minorVersion= *Minor version of the Decision Service*

For example:

```
http://localhost:8850/axis/corticon/decisionService/getVocabularyMetadata  
?name=ProcessOrder&majorVersion=1&minorVersion=1
```

The metadata API is available for Corticon Decision Services deployed for either a Java Server or a .NET Server.

The following JSON-formatted document is an example of a response:

```
{  
  "majorVersion": 1,  
  "name": "MetadataTest",  
  "minorVersion": 0,  
  "entities": [  
    {  
      "name": "Entity_1",  
      "associations": [  
        {  
          "name": "associationObject1",  
          "targetEntity": "AssociationObject1",  
          "inContext": true,  
  
          "mandatory": false  
          "cardinality": "1"  
        },  
        {  
          "name": "associationObjectOverride",  
          "targetEntity": "AssociationObject2",  
          "inContext": false,  
          "mandatory": true  
          "cardinality": ""  
        }  
      ],  
      "attributes": [  
        {  
          "dataType": "Boolean",  
          "name": "boolean1",  
          "inContext": true,  
          "type": "Base",  
          "mandatory": true  
        },  
        {  
          "dataType": "Date",  
          "name": "date1",  
          "inContext": false,  
          "type": "Transient",  
          "mandatory": false  
        },  
        {  
          "dataType": "DateTime",  
          "name": "datetime1",  
          "inContext": true,  
          "type": "Base",  
          "mandatory": true  
        },  
        {  
          "cdtConstraintExpr": "value < 100.0",  
            (Constraint expression associated with this Attribute.)  
          "dataType": "Decimal",  
          "name": "decimal1",  
          "inContext": false,  
          "type": "Transient",
```

```

        "mandatory": false
    },
    {
        "dataType": "Integer",
        "name": "int1",
        "cdtEnumeration": [    (A CDT that is values only, no labels.)
            {"value": "1"},
            {"value": "2"},
            {"value": "3"},
            {"value": "4"}
        ],
        "inContext": true
        "type": "Base",
        "mandatory": true
    },
    {
        "dataType": "String",
        "name": "string1",
        "cdtEnumeration": [    (A CDT that has labels and values.)
            {
                "value": "s"
                "label": "Small"
            },

            {
                "value": "m"
                "label": "Medium"
            },

            {
                "value": "l"
                "label": "Large"
            }
        ],
        "inContext": false,
        "type": "Transient",
        "mandatory": false
    },
    {
        "dataType": "Time",
        "name": "time1",
        "inContext": true,
        "type": "Base",
        "mandatory": true
    },
]
},
{
    "name": "AssociationObject1",
    "associations": [],
    "attributes": [
        {
            "dataType": "String",
            "name": "string1",
            "inContext": true
            "mandatory": true
        },
    ]
},
{
    "name": "AssociationObject2",
    "associations": [],
    "attributes": [
        {
            "dataType": "String",
            "name": "string1",
            "inContext": false
            "mandatory": false
        },
    ],
}
]

```

```

    }
  ]
}

```

Summary of REST methods for management of Decision Services

Summary of REST methods for management of Corticon Servers and Decision Services

Method	Type	Syntax and function
API ListDecisionServices	HTTP GET	<code><base>/decisionService/list</code> 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	<code><base>/decisionService/deploy</code> Attempts to add a Decision Service to the server. Request objects will be sent as the content.
API Undeploy Decision Service	HTTP POST	<code><base>/decisionService/undeploy</code> 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	<code><base>/decisionService/getProperties</code> 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	<code><base>/decisionService/setProperties</code> Attempts to modify the properties of a specified Decision Service.
API Ping Server	HTTP GET	<code><base>/server/ping</code> Returns an object containing the current uptime of the server, which confirms that the server is reachable and running.

Method	Type	Syntax and function
API Retrieve Metrics	HTTP POST	<code><base>/server/metrics</code> 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	<code><base>/server/info</code> 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	<code><base>/server/log</code> Returns the server log entries after the specified timestamp.
API Get Server Properties	HTTP GET	<code><base>/server/getProperties</code> Returns the properties pertaining to the server. The returned object will be a list of key/value pairs consisting of key: <i>propertyName</i> value: <i>value</i> with information about the property.
API Set Server Properties	HTTP POST	<code><base>/server/setProperties</code> Sets properties on the server using a JSON object. The object will contain a key/value format system: <i>propertyName</i> value: <i>value</i> intended to be set for this property.
API Set Server License	HTTP POST	<code><base>/server/setLicense</code> 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 213, 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 223. 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/ping

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: 5.6.1.0",
  "serverDiagnosticWaitTime": "",
  "serviceReleaseNumber": "1.0",
  "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 with 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

Setting Web Console server properties

The following properties are settings you can apply to your Web Console Server installation by adding the properties and appropriate values as lines in its `brms.properties` file, and then restarting Server. The effect of these settings will be realized by users of the Web Console browser clients connected to this Web Console server.

Properties related to monitoring execution times of Decision Service - Version over defined interval periods.

`com.corticon.server.monitoring.decisionservice.interval.record.times`

Specifies whether the Server will auto-start recording time interval measurements.

Note: The time interval monitoring service can be shutdown and restarted using the following methods, which will override this setting.

- `ICcServer.stopServerExecutionTimesIntervalService()`
- `ICcServer.startServerExecutionTimesIntervalService()`

Default value is true

`com.corticon.server.monitoring.decisionservice.interval.record.times=true`

Properties related to Decision Service - Version level monitoring.

`com.corticon.server.monitoring.decisionservice.record.data`

Specifies whether the Server will auto-start recording time measurements.

Note: The data recording monitoring service can be shutdown and restarted using the following methods,

which will override this setting.

- `ICcServer.stopServerResultsDistributionMonitoringService()`
- `ICcServer.startServerResultsDistributionMonitoringService()`

Default value is true

```
com.corticon.server.monitoring.decisionservice.trackingattribute.
```

```
<number>=<ds name>;<ds version number>;<tracking attribute>;<attribute type>;<bucket definitions>
```

where:

- `<ds name>` = Name of the Decision Service to be monitored
- `<ds major version number>` = Major Version number of the Decision Service to be monitored
- `<ds minor version number>` = Minor Version number of the Decision Service to be monitored
- `<tracking attribute>` = Fully qualified path to the attribute as defined in Vocabulary
- `<attribute type>` = Datatype of `<tracking attribute>`. Supported values include: Boolean, Date, Decimal, Integer, and String.
- `<bucket definitions>` = 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
```

except for bucket definitions which uses

```
com.corticon.server.monitoring.decisionservice.bucket.registration.delimiter
```

Example:

```
com.corticon.server.monitoring.decisionservice.trackingattribute.1=
```

```
AllocateTrade;1;1;Trade.transaction.dPrice;Decimal
```

or

```
com.corticon.server.monitoring.decisionservice.trackingattribute.1=
```

```
AllocateTrade;1;1;Trade.transaction.dPrice;Decimal;<100, [100..200), >= 200
```

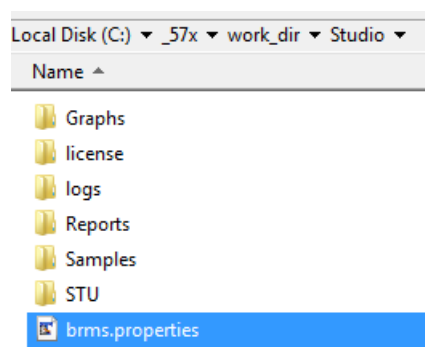
```
com.corticon.server.monitoring.decisionservice.record.data=true
```

D

Configuring Corticon properties and settings

Corticon provides properties that specify property names and default values of user-configurable behaviors in Corticon Studio and Corticon Servers.

The settings file `brms.properties` is installed at the root of `[CORTICON_WORK_DIR]` for each Studio and Server installation. If you install Studio and Server on one machine and accept the default colocating paths, one `brms.properties` file is installed to be shared by Studio and Server:



About the `brms.properties` file

- It is good practice to back up the file before you start to make changes.
- When installed separately, the Studio and Server `brms.properties` files are identical.
- If you delete the file, it does not get recreated at restart. However, as these are overrides to default properties, there is no loss of features or functionality when the file is not present.
- In the absence of a `brms.properties` file, you can simply list property settings in a text file, and then save it to its proper location as `brms.properties`.
- An update of the installation will preserve a modified `brms.properties` file, and will add the default file if none is present.

Enabling settings listed in the default `brms.properties` file

The file lists properties that users commonly want to change. Each group of properties provides descriptive comments and the commented default name=value pair.

To specify a preferred value for a listed property, 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 express a preference for decimal values displayed and rounded to two places instead of the six places preset for this property, locate the line:

```
#decimalscale=6
```

and then change it to

```
decimalscale=2
```

Adding unlisted settings to `brms.properties` file

Some locations in the documentation tell you about other property settings that you might want to add to the settings file. Or you might be directed by technical support or your Progress representative to add or change settings to provide certain behaviors or functions.

For example, to change interval of diagnostic readings from five minutes to two minutes, 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:

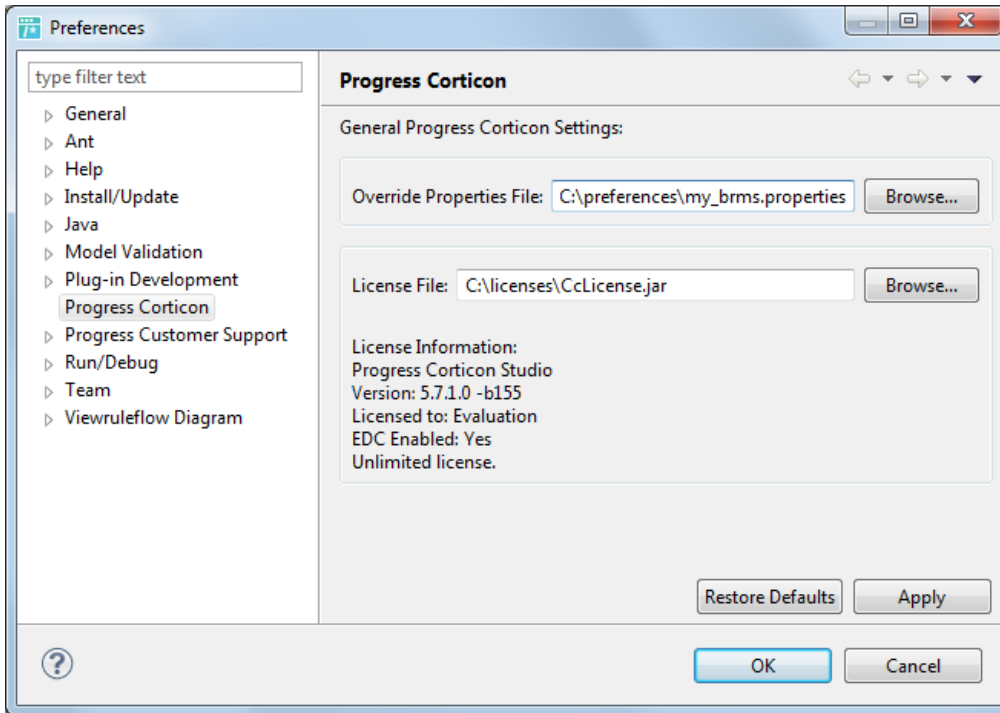
```
com.corticon.server.DiagnosticWaitTime=120000
```

If you add the same property more than once in the settings file, the last instance takes precedence.

Saving and applying the revised property settings

When your changes are complete, you can choose to save the settings file with its default name and location, but you could save a copy with a useful name, such as `debuggingLogSettingsbrms.properties`.

- **Server** - Requires the file name `brms.properties` located at its default location. You can copy an alternative properties file to rename it, and then paste into the standard location with the expected name.
- **Studio** - You can save multiple settings files, and then use Studio's **Preferences** to specify the **Override Properties File** for the `brms.properties` to use, as illustrated:



For the revised settings to take effect, save the edited file, and then restart the Corticon Studio and Corticon Servers.

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.

Note: Administrative APIs for certain transient property settings - 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 use its default settings and apply the `brms.properties` file.

Corticon 5.7.2 Online Tutorials and Documentation

TUTORIALS: Learn about Corticon from online lessons at the [Corticon Learning Center](#).

DOCUMENTATION: About this release	
<i>What's New in Corticon</i> Online Help PDF	Describes the enhancements and changes to the product since its last point release.
<i>Corticon Installation Guide</i> Online Help PDF	Step-by-step procedures for installing Corticon Studio and Servers in this release on Windows and Linux platforms.

DEVELOPMENT DOCUMENTATION: Define and Model Business Rules	
<i>Rule Modeling Guide</i> Online Help PDF	Introduces how business rules are modeled in Corticon Studio including the creation of Vocabularies, Rulesheets, Ruleflows, and Ruletests. This is the starting point for new rule modelers.
<i>Quick Reference Guide</i> Online Help PDF	Reference guide to the Corticon Studio user interface, including descriptions of all menu options, buttons, and actions.
<i>Rule Language Guide</i> Online Help PDF	Reference information for all operators available in the Corticon Studio Vocabulary.

<i>Extensions Guide</i> Online Help PDF	Detailed technical information about the Corticon extension framework for extended operators and service callouts.
<i>Javadoc for Extensions API</i> Javadoc	Complete Java API reference for Corticon Extensions.

DEPLOYMENT DOCUMENTATION: Run Decision Services on Servers

<i>Integration and Deployment Guide</i> Online Help PDF	Provides details on deploying and managing Corticon Decision Services. Start here to learn about the options for deploying Corticon, the best practices for managing your deployment, and the APIs for integrating Corticon with clients executing rules.
<i>Data Integration Guide</i> Online Help PDF	Provides details on how to integrate Corticon with external data sources and how to use Corticon for batch rule processing. Includes details on Corticon EDC and ADC features for accessing data from Decision Services.
<i>Web Console Guide</i> Online Help PDF	Presents the Corticon Web Console which can be used to manage your Corticon deployment. This GUI interface simplifies the management and monitoring of Corticon servers and Decision Services.
<i>Deploying Web Services with Java</i> Online Help PDF	Provides details on deploying Corticon as a REST or SOAP web service on Java application servers.
<i>Deploying Web Services with .NET</i> Online Help PDF	Provides details on deploying Corticon as a REST or SOAP web service on Microsoft IIS.
<i>Javadoc for Corticon Server API</i> Javadoc	Complete Java API reference for Corticon Server.