Progress®
Corticon®

Corticon
Deployment

Copyright

Last updated: Corticon 7.3

Updated: 2025/11/19

Table of Contents

1

Introduction to Corticon deployment

Choose the deployment architecture

When choosing how to deploy Decision Services you first need to consider how they will be called by your application. Corticon supports both web service and in-process deployment. Web service deployment allows your Decision Services to be called as a REST or SOAP service. This is the most common way of deploying and integrating Corticon. In-process deployment allows you to embed Corticon in your application. This requires custom code within your application but has the performance advantage of not making a network call to execute rules.

Corticon Decision Services are deployed into a Corticon Server. For both web service and in-process deployment you first deploy the Corticon Server and then deploy your Decision Services to it. Once deployed to the server, they become available to your application.

Deploying as a web service requires deploying the Corticon `axis.war` to your application server. Once deployed you can configure security and access control using the services of your application server. When accessing your Decision Services as a REST service, you will pass the data to process as a JSON payload. When accessing them as a SOAP service, you will pass the data as an XML payload. *See the Web Services guide for more information*.

Deploying in-process requires custom code in your application to instantiate a Corticon server, load Decision Services, and invoke them with your application data. When deployed in-process you can pass data to your Decision Services either as JSON or XML, or as Business Objects using Corticon's Java Object Messaging. *See the In-Process Guide for more information.*

Packaging the Decision Service

To deploy your rules to a Corticon Server you must package them as a Decision Service. Packaging takes all the required rule assets to produce an `.eds` file ready for deployment. Corticon provides multiple options for packaging your rule assets into Decision Services including:

- Packaging Decision Services from Corticon Studio
- Packaging using ant macros or command line utilities

Once packaged, you can deploy your Decision Service to either a Corticon Server deployed to an application server or a Corticon Server instantiated in-process in a custom application.

Deploying the Decision Service

The options available for deploying your Decision Service to a Corticon Server depend on how the server is deployed.

If deployed to an application server:

- **Deploy using cdd files**—A Corticon Deployment Descriptors (`cdd`) file is a text file identifying one or more Decision Services to be deployed and properties to be set on the Decision Service. The Corticon Server has the option to scan for new or updated `cdd` files, allowing a running Corticon Server to automatically load new or updated Decision Services.
- **Deploy using Corticon Web Console**—The Web Console provides a web UI for managing your Corticon Servers. It allows you to deploy Decision Services as well as get metrics on their operation.
- **Deploy using REST APIs**—The Corticon Server provides a set of REST APIs for managing the server and the Deployment of Decision Services. These APIs allow you to integrate the management of Corticon Servers with other applications.
- **Deploy using command line tools**—The Corticon Server provides command line tools for the deployment of Decision Services. This allows the deployment to be scripted.
- **Deploy from Corticon Studio**—From Corticon Studio you can package Decision Services for deployment as well as deploy them directly to a Corticon Server. This is useful in development environments but is not recommended for production.

If deployed in-process:

- **Deploy using APIs**—When using Corticon in-process you have access to a rich set of APIs for managing the Corticon Server and the deployment of Decision Services. These APIs are available for both Java and .NET applications.
- **Deploy using cdd files**—The option to use `cdd` files for deployment is also available when deploying in-process. The advantage of using `cdd` with in-process deployment is it externalizes the deployment of your decision services from your application.

Deployment Best Practices

Corticon provides multiple options for packaging and deploying Decision Services, giving you flexibility in managing the process. What counts as best practice can vary by organization but here are recommendations to follow:

- **Keep your rule assets in a source control**—Using a source control system such as git allows you to manage access to rule assets, track changes, and track the revisions used when packaging a Decision Service for deployment.
- **Automate the packaging of your Decision Services**—Corticon provides ant macros and command line tools for packaging your rule assets into eds files for deployment. Automating the packaging of your Decision Services will give you greater control and reproducibility of your deployment process.
- **Automate the testing of your Decision Services**—Corticon provides `ant` macros allowing you to script the execution of rule tests created by your rule modelers. Automating the execution of these test allows you to establish a quality gate where you don't deploy Decision Services until rule tests have passed.
- **Automate the deployment of your Decision Services**—Corticon provides command line tools and REST APIs for the deployment of Decision Services. You can also script the deployment when using `cdd` files

Following these best practices allows you to adhere to Continuous Integration/Continuous Deployment (CI/CD) practices and fully automate the package, testing and deployment of your Decision Services and to use automation management tools such as Jenkins and Teamcity.

Cloud and Container Deployment

Corticon Server can be deployed to any cloud platform supporting deployment of web services or custom applications. This includes deployment to Amazon AWS and Microsoft Azure. When deploying to a cloud platform the considerations above for choosing the deployment architecture and methods still apply.

Corticon Server can be deployed using containers such as Docker. Deploying to a container typically involves copying the basic configuration on an existing appserver such as tomcat, and then adding it to your Corticon license and your Decision Services to create a config definition that is ready to run.

About Corticon's Bundled Tomcat

Corticon Server and Web Console install a standard Tomcat distribution to help you quickly get started. This is a standard Tomcat distribution at the time of Corticon release. It may not have the latest security patches or other security configuration changes recommended for production use. When moving to production, it is recommended to deploy Corticon Server and Web Console to a supported application server that you have supplied and secured. If you choose to use the bundled Tomcat in production, you assume responsibility for applying Tomcat security patches and performing security configuration.

2

How to prepare Studio files for deployment

Prior to packaging a Decision Service for deployment you need to ensure the Vocabulary used aligns with the naming of data which will be passed to the Decision Service when it is invoked. Adding a mapping to your Corticon vocabulary allows you to view and customize the mapping.

Note: Decision Service compilation no longer includes the WSDL and the report in the Decision Service (EDS) file by default. If you use WSDL or reports in the EDS, add the following lines to your `brms.properties` file:

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

For more information, see the topic *"Properties that impact Decision Service compilation" in the Corticon Deployment Guide*.

For details, see the following topics:

- XML Mapping
- Java object mapping
- JSON Mapping
- Java Object Mapping

XML Mapping

If the data payload of your call will be in the form of an XML document, then your Vocabulary might need to be configured to match the naming convention of the elements in your XML payload.

Displaying XML Mapping

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

Entity Mapping

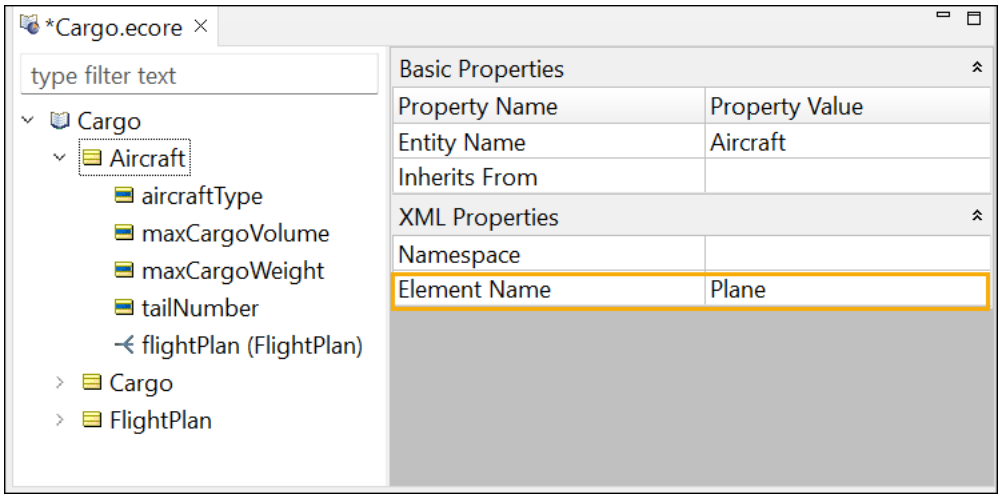
When XML mapping has been added to the Vocabulary, its Entity properties are displayed.

Table 1: XML 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 (`complexTypes`). If the `complexType` matches exactly (spelling, case, special characters, *everything*), then no mapping is necessary. However, if the `complexType` name differs in any way from the Vocabulary entity name, then the `complexType` name must be entered in 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

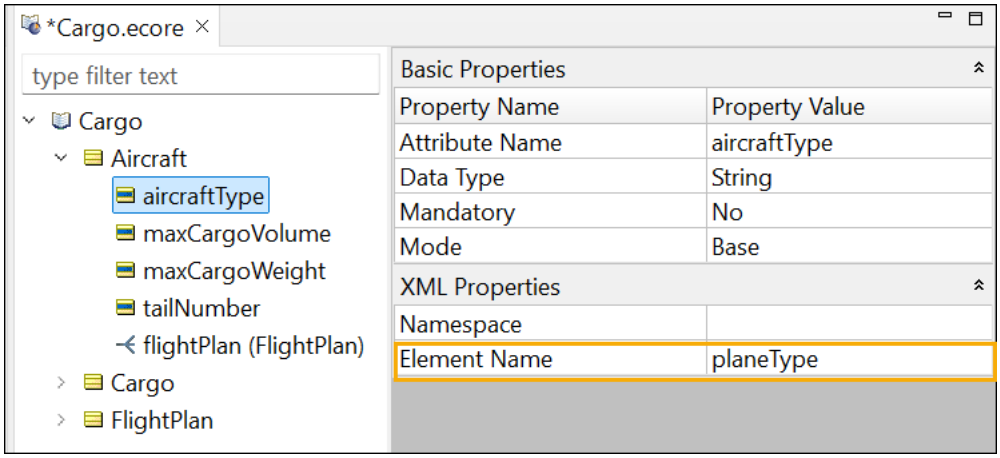
When XML mapping has been added to the Vocabulary, its Attribute properties are displayed.

Table 2: XML 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 in 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

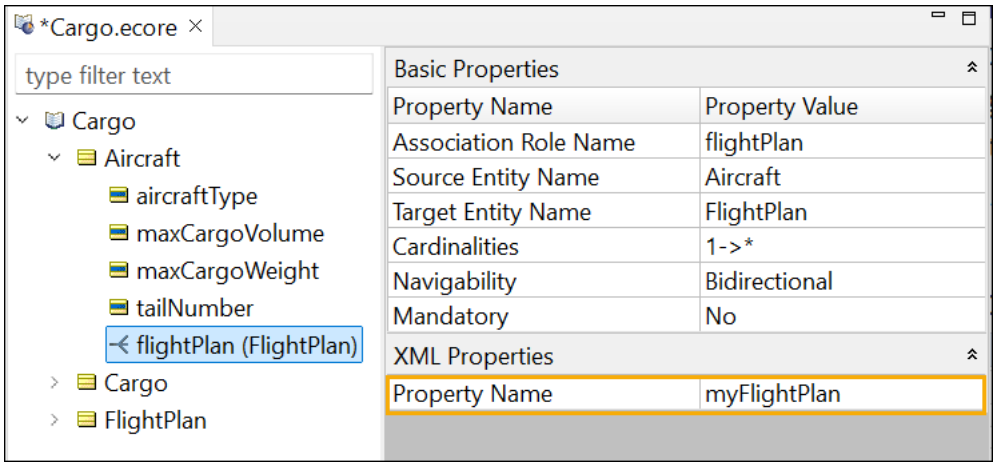
When the Vocabulary has added XML mapping, you set their properties on the properties page of the Association.

Table 3: 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 in 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 specific Decision Service 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:CorticonRequest" />
  <xsd:element name="CorticonResponse" type="tns:CorticonResponse" />
```

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" elementFormDefa
    "qualified">
      <xsd:element name="CorticonRequest" type="tns:CorticonRequest" />
      <xsd:element name="CorticonResponse" type="tns:CorticonResponse" />
```

Setting XML Namespace Mapping preference for unique target namespaces

Systems that are strict 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 respective `brms.properties` files 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 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:CorticonRequest" />
  <xsd:element name="CorticonResponse" type="tns:CorticonResponse" />
```

Figure 7: WSDL with unique Namespace

```
<?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">
```

Java object mapping

If 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: See the Tutorial: Deploy as a Java in-process Server

for an example of building the get and set methods

that are then packaged into a JAR.

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.

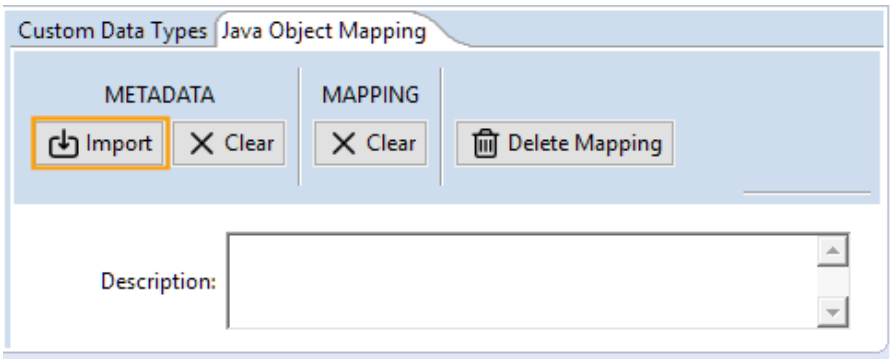
To import package metadata:

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.

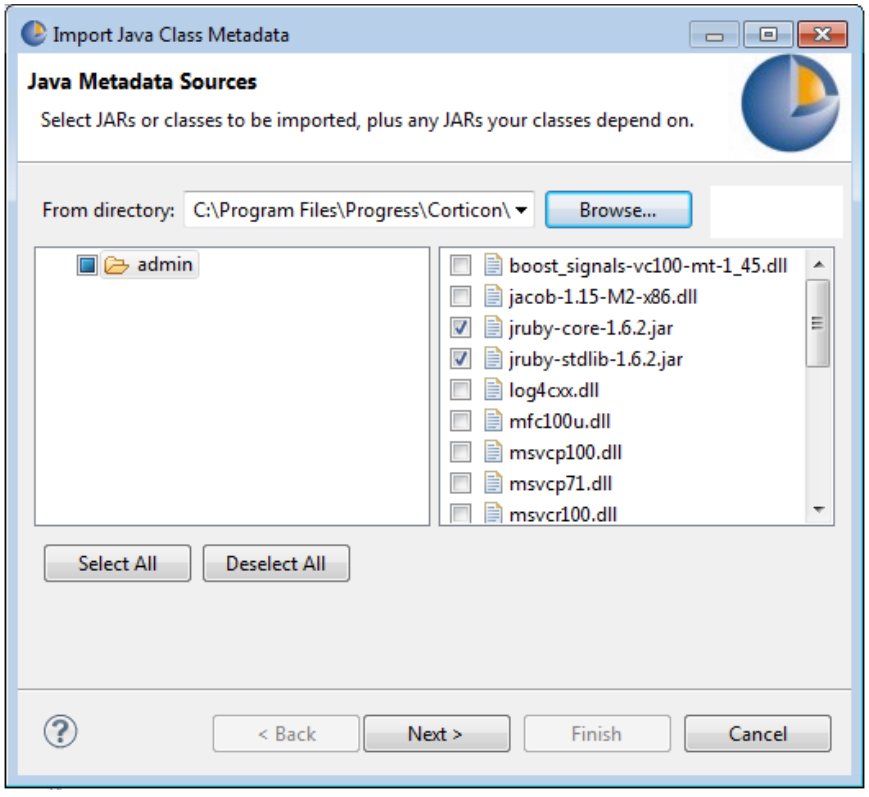
3. On the **Java Object Mapping** tab, click **Import**

Figure 8: Importing Java Class Metadata for Mapping



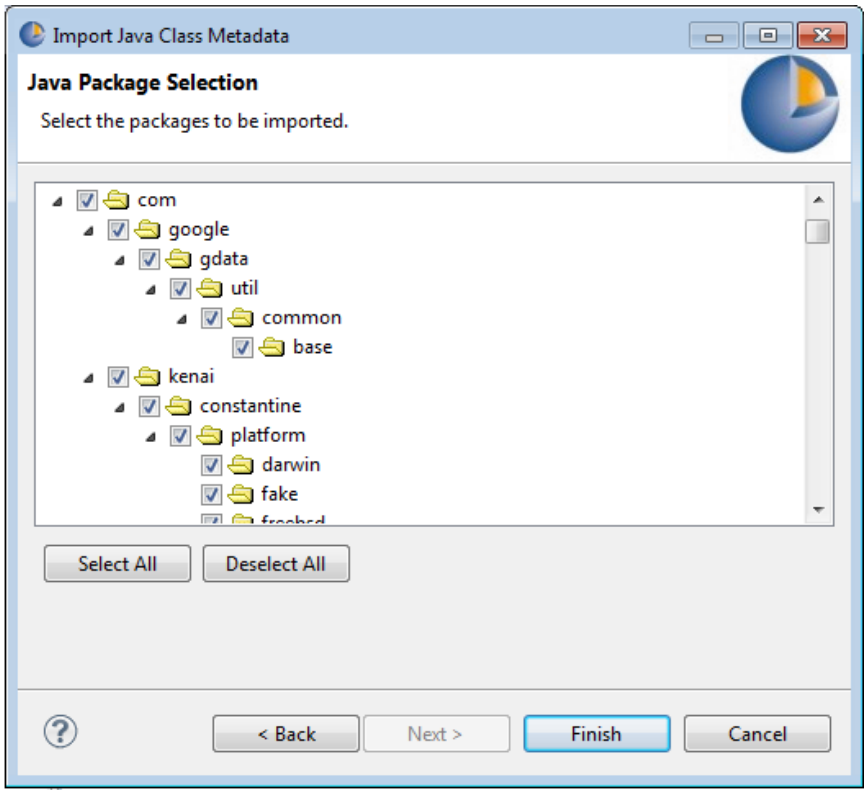
4. Use the **Browse** button to select the location of your Java Business Objects. They should be compiled `class` files or Java archives (`.jar` files).

Figure 9: Browsing to your Java Class files



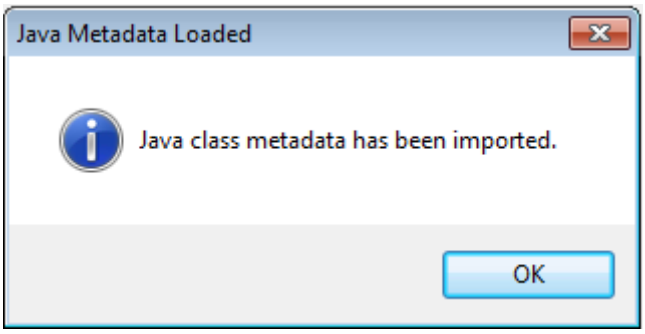
5. Select the package containing the Java business objects as shown:

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.

Entity mapping

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

Figure 12: First Portion of MyAircraft Class

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

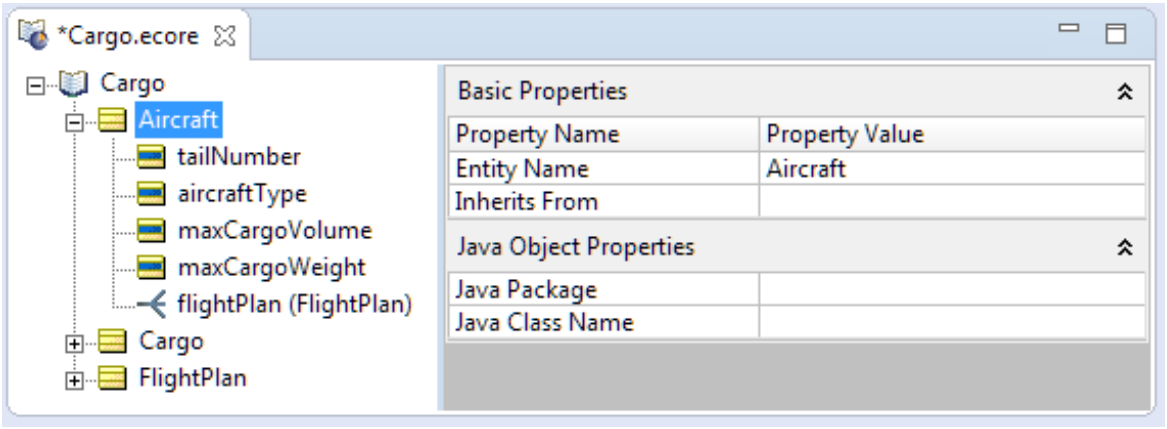
We can see in line 6 of this figure that this class is not actually named `Aircraft` – it is named `MyAircraft`. The automatic mapper attempts to locate a class by the same name as each entity. In the case of `Aircraft`, it looks for a class named `Aircraft`. Not finding one, it leaves the field empty,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

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

Table 4: Java Object Mapping Entity Properties

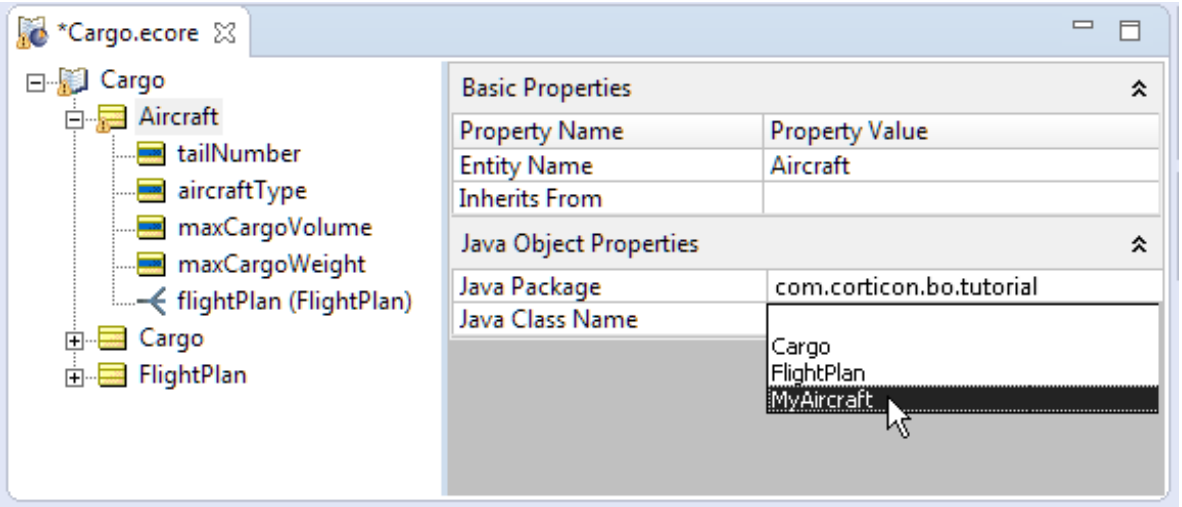
Property	Value
Java Package	Specifies the package to be used for Java class metadata mapping.
Java Class Name	Maps the entity to the specified class when no class exists with the entity name.

Figure 13: Default Map of Class to Entity



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

Figure 14: Manually Mapping MyAircraft Class to Aircraft Entity



Attribute mapping

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

Figure 15: Second Portion of MyAircraft Class

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

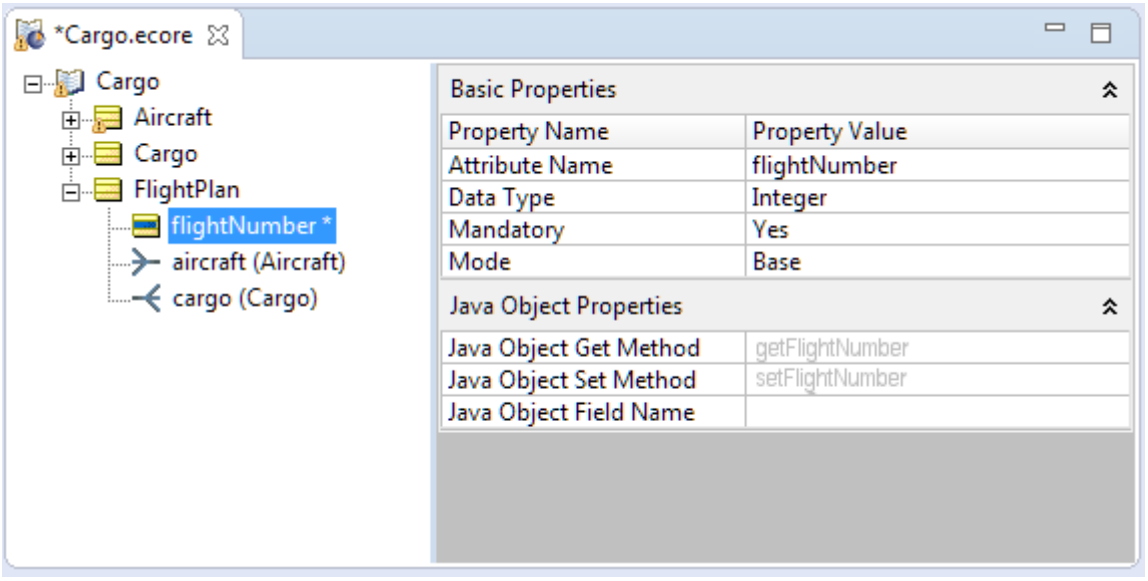
When Java Object mapping has been added to the Vocabulary, its Attribute properties are displayed.

Table 5: 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.

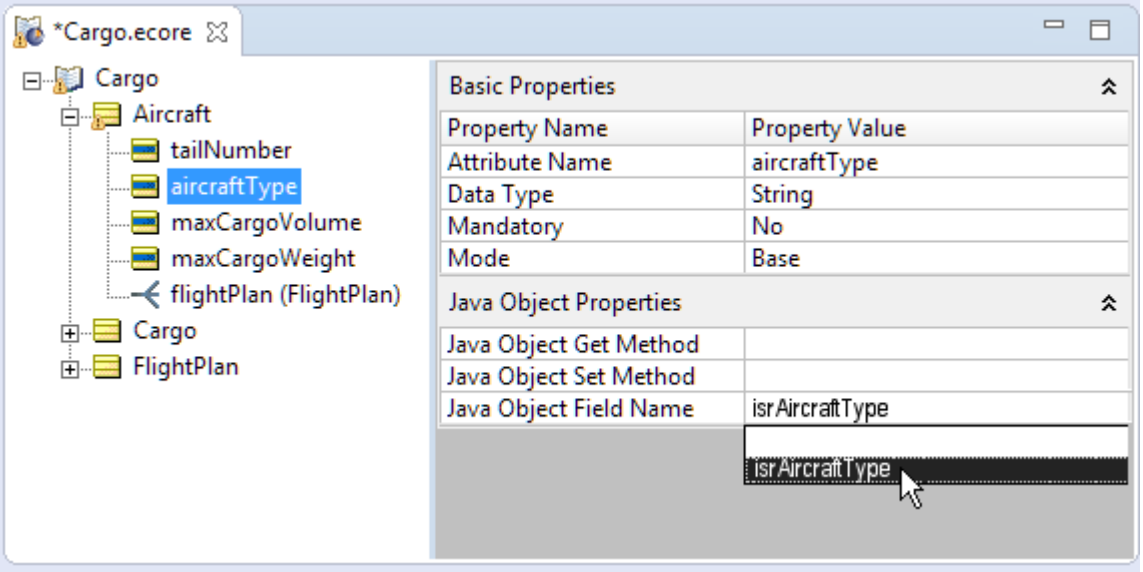
In the case of attribute `flightNumber`, the mapper finds get and set methods that conform to this naming convention, so the method names are inserted into the fields in gray type, as shown in the following figure.

Figure 16: Auto-Mapped Attribute Method Names



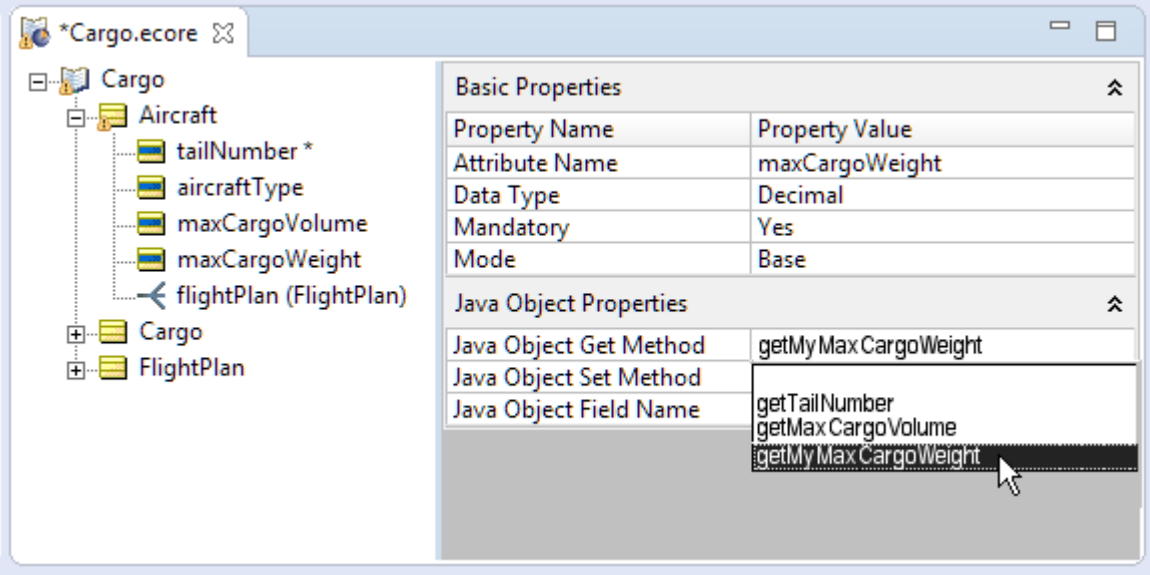
In those cases where the mapper cannot locate the corresponding methods, you will need to select them manually. Notice in the `MyAircraft` class shown in First Portion of MyAircraft Class, no get and set methods exist for `istrAircraftType` since it is a public instance variable. Therefore, we need to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

Figure 17: Manually Mapped Public Instance Variable Name



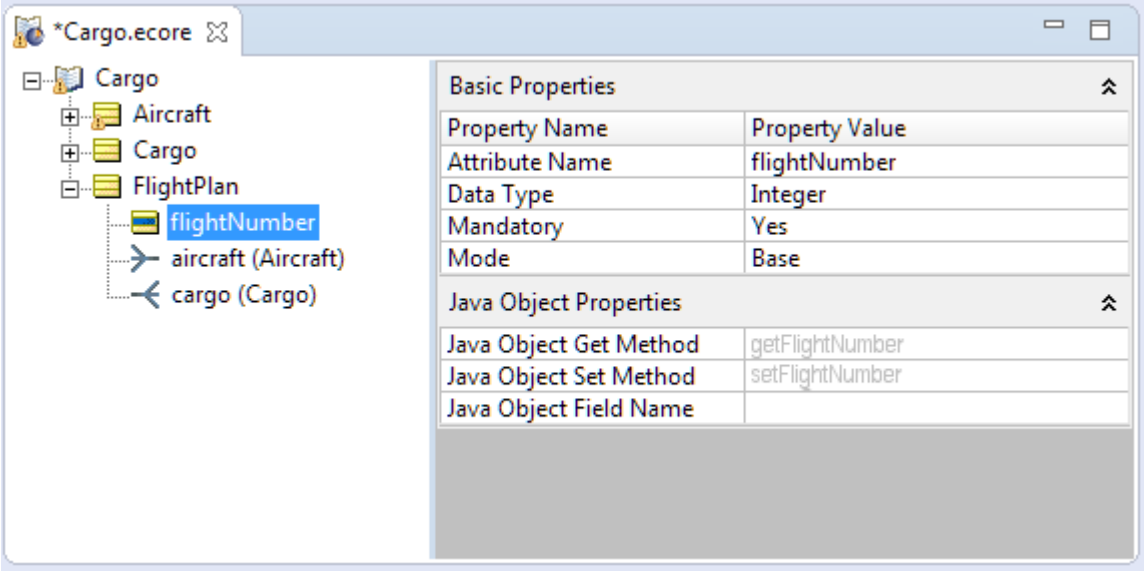
When a class property contains get and set methods, but their names do not conform to the naming convention assumed by the auto-mapper, we must select the method names from the **Java Object Get Method** and **Set Method** drop-downs, as shown in the following figure.

Figure 18: Manually Mapped Property Get and Set Method Names



A property's data type is detected by the auto-mapper, so there is no need to manually enter it. This is shown by the `flightNumber` attribute in the following figure.

Figure 19: Auto-Mapped Property Despite Different Data Type



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

Association mapping

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

Table 6: Java Object Mapping Association Properties

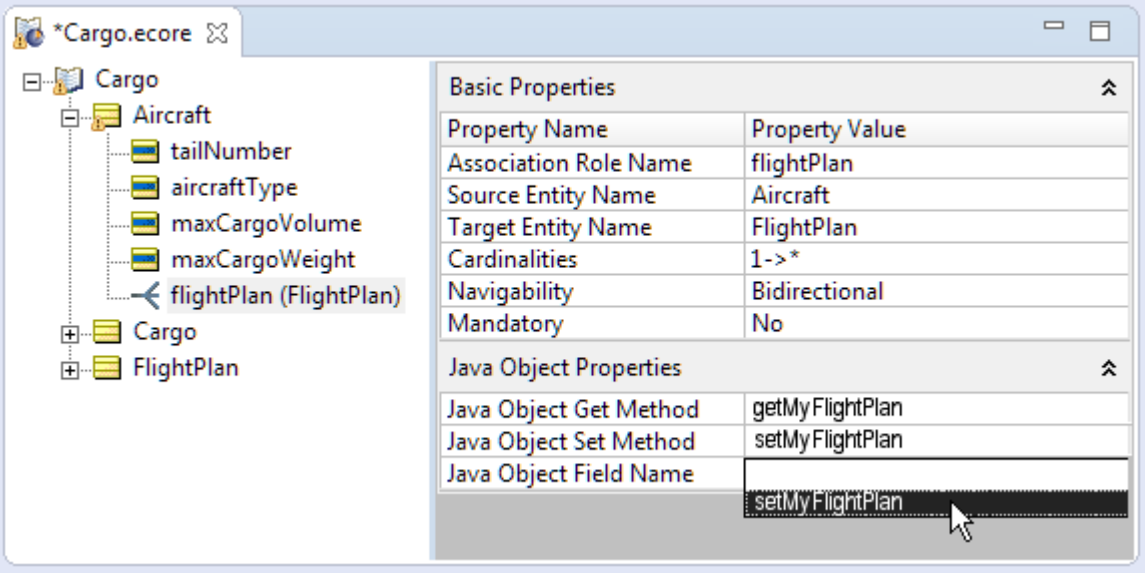
Property	Value
Java Object Get Method	Manually specifies the GET method of a class property that does not conform to naming conventions of the auto-mapper.
Java Object Set Method	Manually specifies the SET method of a class property that does not conform to naming conventions of the auto-mapper.
Java 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 20: Third Portion of MyAircraft Class

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

Figure 21: Manually Mapped Association Get and Set Method Names

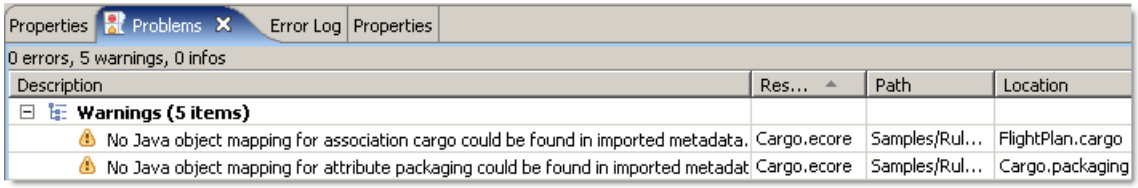


Note: Java generics - Support for type-casted collections is included in Corticon Studio. If your Java Business Objects include type-casted collections (introduced in Java 5), then Corticon will ensure these constraints are interpreted correctly in association processing.

Verify Java object mapping

If there are warning markers on any of the Vocabulary nodes in Entity mapping on page 20, Attribute mapping on page 21, or Association mapping on page 24 whose mappings have not yet been found, each warning has a corresponding message entered in the **Problems** window, as shown:

Figure 22: Problem window showing list of current mapping problems



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

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

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 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`}. 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 23: Vocabulary CDT Labels must match Business Object Enumeration Labels (Types)

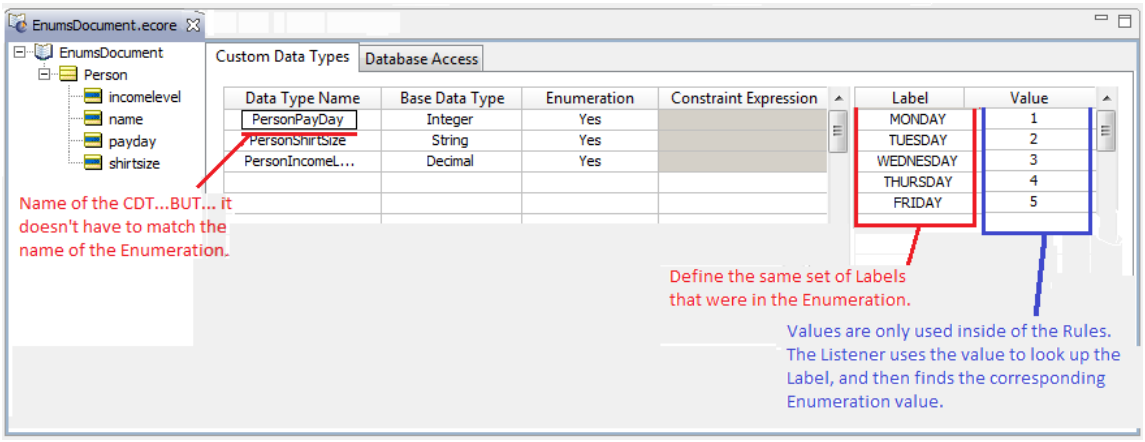
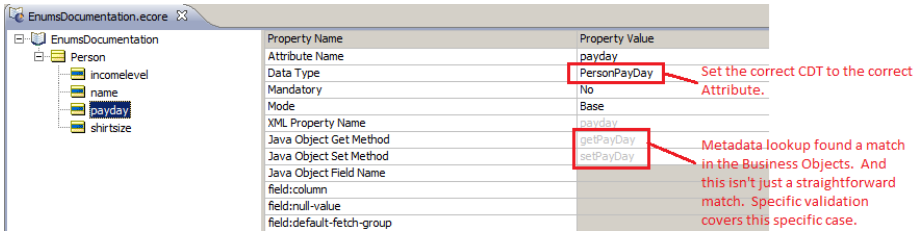


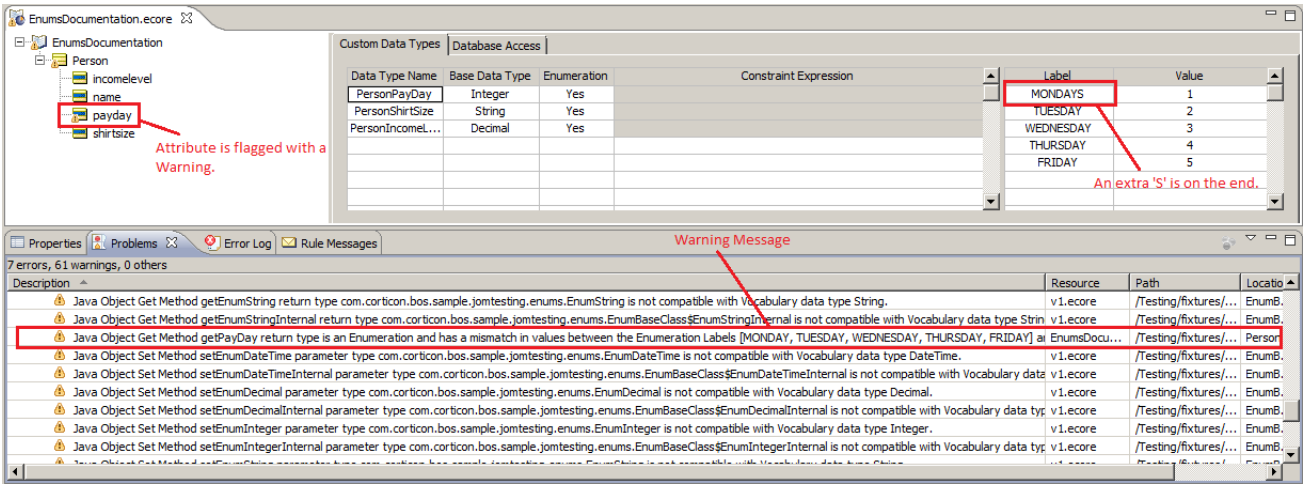
Figure 24: Vocabulary Mapper found correct Metadata based on matching enumeration labels



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 25: Vocabulary CDT Label / Object Enumeration Label Mismatch



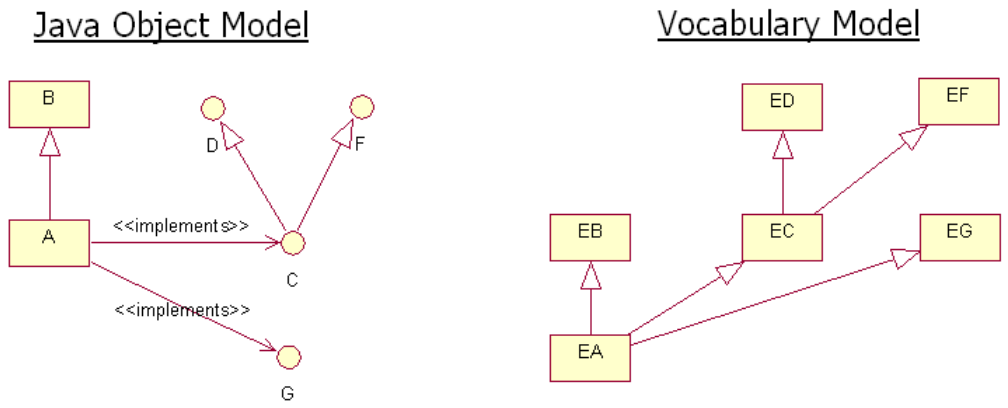
Notice the Vocabulary attribute is “flagged” with the small orange warning icon (shown in the upper left of the figure above). The associated warning message states:

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].`

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 26: 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:

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

Naming conventions used in the algorithm above:

- 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 descendants found in DS context
- AE = An Ancestor Entity (one with descendants) found in DS context
- CDO = In memory Java Data Object created by Corticon for use in rule execution

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.

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 Decision Service 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

```
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
```

JSON Mapping

If the data payload of your call will be in the form of a JSON document, then your Vocabulary might need to be configured to match the naming convention of the elements in your JSON payload.

Displaying JSON Mapping

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

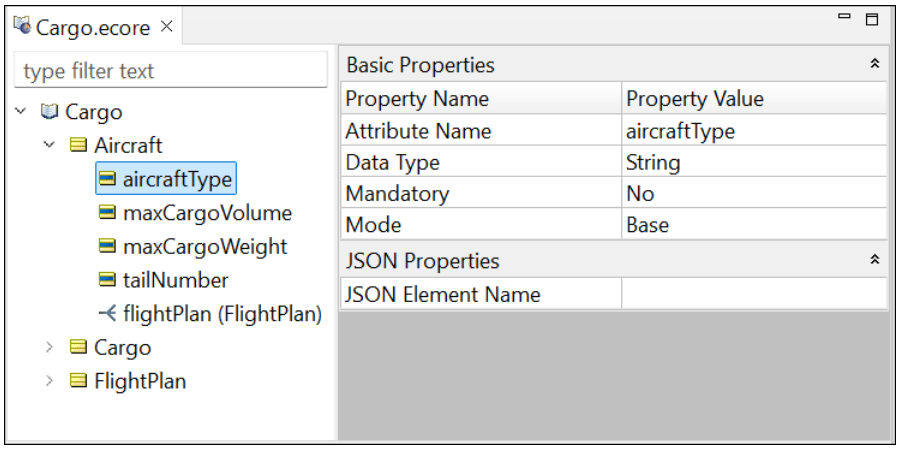
Entity Mapping

When JSON mapping has been added to the Vocabulary, no Entity properties are displayed.

Attribute Mapping

When JSON mapping has been added to the Vocabulary, its Attribute properties are displayed. 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 in the **JSON Element Name** property.

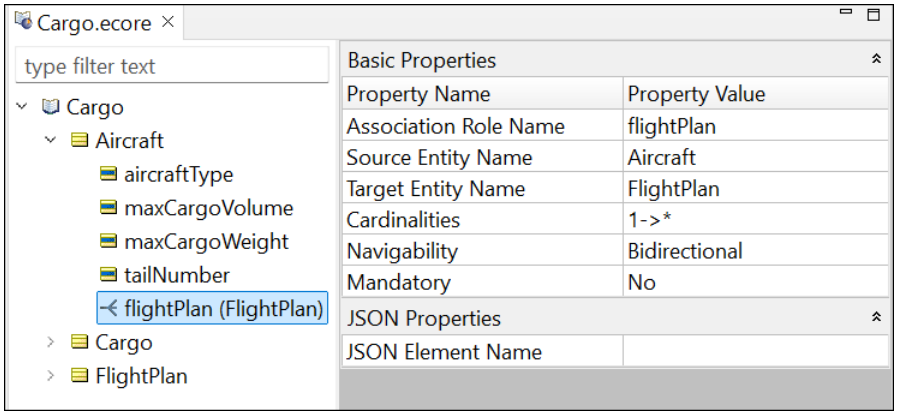
Figure 28: Mapping a Vocabulary Attribute to a JSON SimpleType



Association Mapping

When the Vocabulary has added JSON mapping, you set their properties on the properties page of the Association. 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 in the **JSON Element Name** property.

Figure 29: Mapping a Vocabulary Association to a JSON ComplexType



Native JSON Mapping

Once you have defined the JSON Mappings you might want to prepare your Decision Services for native JSON. You need to anticipate a few characteristics of native JSON:

- The parameters of a native JSON request are in the request line.
- The format, while simple, is consistent.
- The request processing algorithm will need to disambiguate.
- The Decision Service root entities will help or constrain the request.

Java Object Mapping

If the data payload of your call will be in the form of a JSON document, then your Vocabulary may need to be configured to match the naming convention of the elements in your JSON payload.

Note: See the Tutorial: Deploy as a Java in-process Server for an example of building the get and set methods that are then packaged into a JAR.

Displaying Java Object Mapping

On the Vocabulary menu, choose **Add Document Mapping > Add Java Object Mapping.**

Entity Mapping

In an Entity property, the Java package and class names, as well as the Java object field name.

Attribute Mapping

For each attribute, the get and set object methods are displayed, as well as the Java object field name.

Association Mapping

For each association, the get and set object methods are displayed, as well as the Java object field name.

3

How to package and deploy 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. As your solution moves toward production, refine your approach to better manage your deployed rules and Corticon Servers.

Note: The Server's major.minor version must accept the EDS file. Corticon Server will refuse to deploy invalid versions, and will log or display an error.

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 CDD (Corticon Deployment Descriptor) file, a text file that identifies one or more 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.

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

- Use the Server API to add and manage Decision Services. See the Deploy Corticon Server in an Application topic for more details.

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

For details, see the following topics:

- Deployment related files
- Use Studio to package and deploy Decision Services
- Use Web Console to deploy Decision Services
- Use Deployment Descriptors to deploy Decision Services
- Automate packaging and testing of Decision Services
- Use Server API to compile and deploy Decision Services
- Properties that impact Decision Service compilation
- Properties that are incorporated into Decision Services

Deployment related files

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

Rule asset 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 (`.ecore`) onto worksheets (`.ers`) that define the rules and embeds other worksheets into a Ruleflow (`.erf`) 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 embedded in a compiled Decision Service.

Test asset files

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

Corticon Deployment Descriptor files

One option for deploying Decision Services is through CDD (Corticon Deployment Descriptor) files. These files let you package Ruleflows compiled into Decision Services, and their deployment parameters in an XML-formatted text file.

When Corticon Server reads a CDD file, it reads in each instance defined in the file to load its Decision Service, and then sets its execution and configuration parameters.

(See Setting the autoloaddir property on page 48 for additional information.)

Note: If you are using the bundled Apache Tomcat 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.

Decision Service files

A Decision Service file (`.eds`) is a self-contained, complete deployment asset that includes compiled versions of all its component rule assets and any extensions used. Only the EDS file needs to be deployed to Corticon Server. The related rule assets are not needed. If using CDD deployment, Corticon Server will automatically reload updated EDS files.

Note: If your Ruleflow uses custom Extensions or Service Call-Outs (SCOs), be sure to add their classes to the project as described in *"How to use extensions when creating Decision Services" in the Extensions Guide.*

Schema files

Schema files define a *service contract* -- the interface to a service for client 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`).

This section includes Generate WSDL and XSD schema files on page 57 as part of the command line utilities, while Service contract options on page 74 discusses its usage as part of the section "Integrating Corticon Decision Services."

Datasource Configuration files

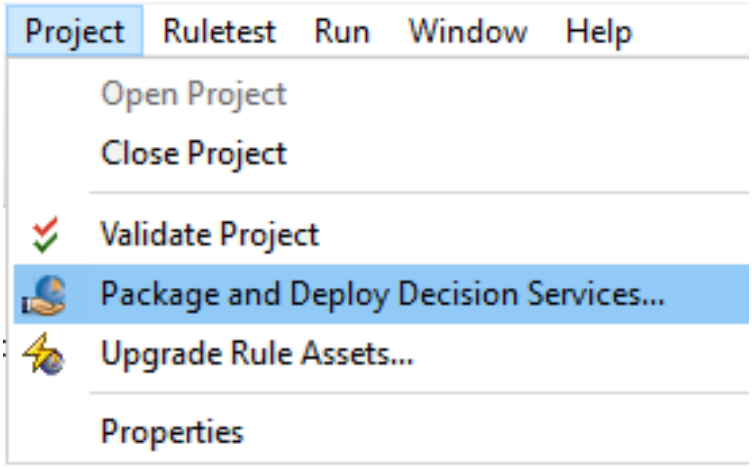
If your Decision Service integrates with databases or REST services, you will need a Datasource Configuration File when deploying the Decision Service. This is an XML file which defines the connection parameters for the data sources used by your Decision Service. It can be exported from the Vocabulary editor and modified for deployment. Separating Datasource connection parameters from the EDS file for your Decision Service allows you to easily change these parameters when, for example, moving from a test to a production environment.

Use Studio to package 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.

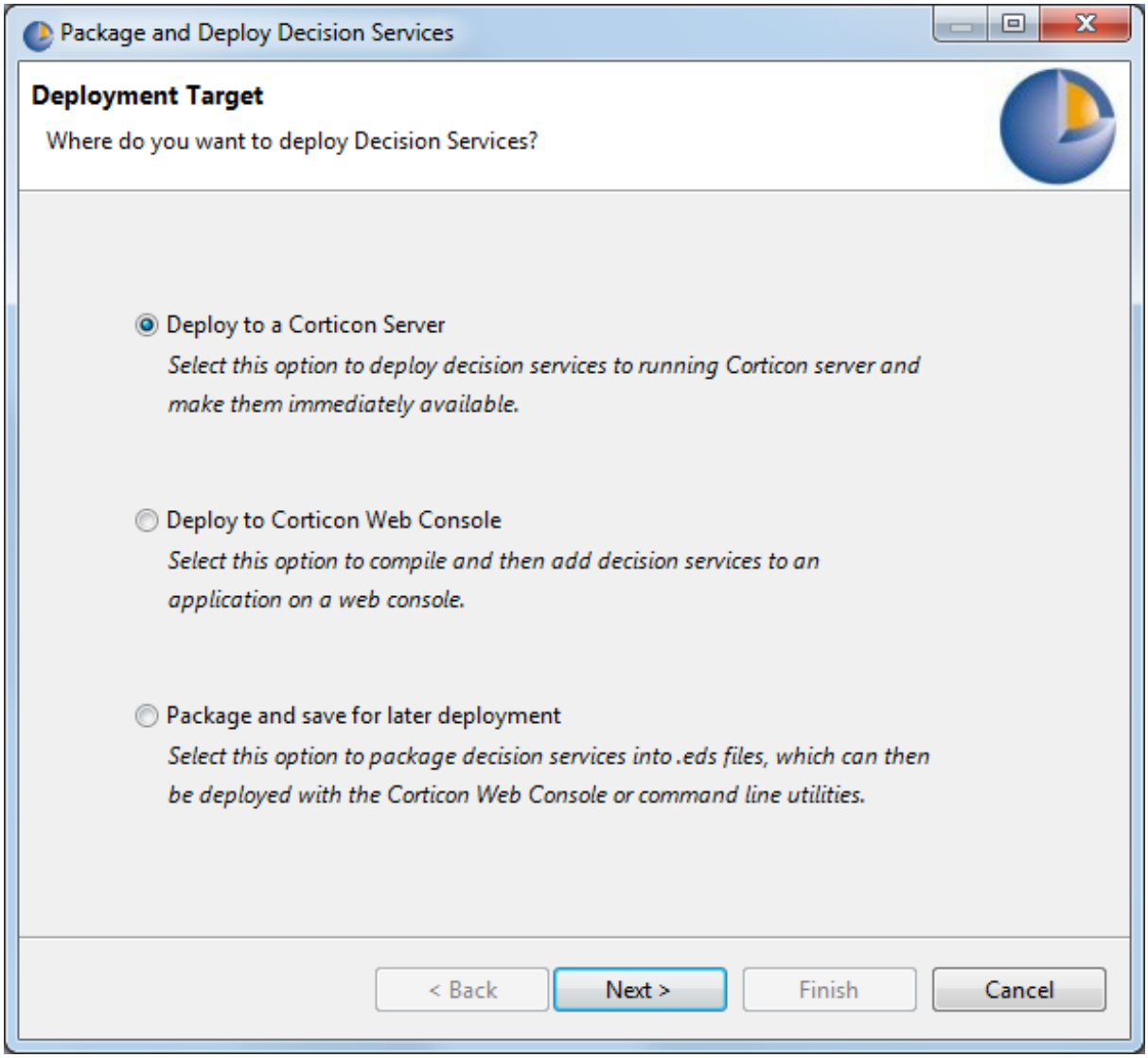
Starting the Package and Deploy Decision Services wizard

To package and deploy a Decision Service, choose the **Project** menu's **Package and Deploy Decision Services** action, as shown:



You can select multiple projects by multi-selecting the several projects in the Project Explorer, and then right-clicking to choose **Package and Deploy Decision Services**.

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



The packaged Decision Services can be deployed directly to a Corticon Server, deployed to the Web Console which will then deploy it to one or more Corticon Servers, or saved as an EDS file for later deployment. Select your preference, and then click **Next**.

See:

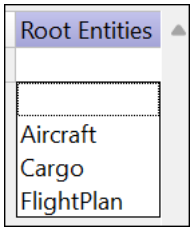
Deploy to a Corticon Server on page 37

Deploy to Corticon Web Console on page 40

Package and save for later deployment on page 43

Root Entities

One aspect of creating decision services is easing the burden in the algorithm that determines the root of the request data.



If the root Entity does not have a defined `#type` in the `__metadata` (both are optional), then an algorithm is applied to find the best match between the data in the payload root Entity and that in the Vocabulary.

The **Root Entities** setting allows a user to define a set of possible Entities that the algorithm will be applied to. This will limit the number of Vocabulary Entities to be compared against the payload Entity. Having the algorithm only compare a limited set of Vocabulary Entities, the matching may be more accurate, and with less processing time, and faster execution. If no Root Entities are defined for a decision service, the algorithm will need to compare each root Entity in the payload against all Entities in the Vocabulary to determine best match.

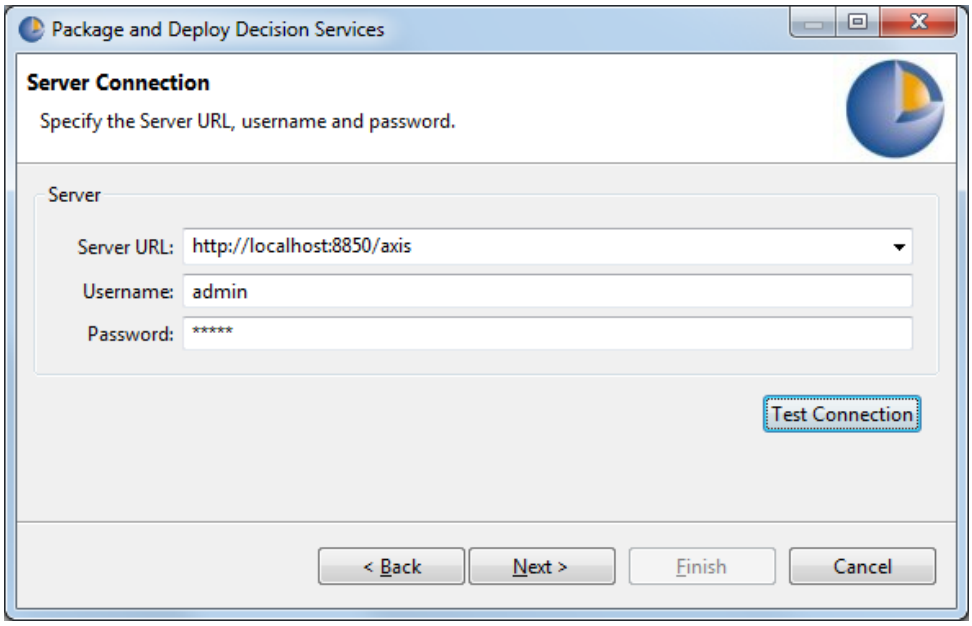
You can multiselect in this dialog.

It is important to note that the selection is part of the Decision Service. If you have entities A, B, C, D, and you chose C and D as the Root entities, the algorithm will only assign the Root entities to C or D, even if the root Entity in the payload better matches against an A or B. If you make no selection, the algorithm will make best effort.

Deploy to a Corticon Server

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

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



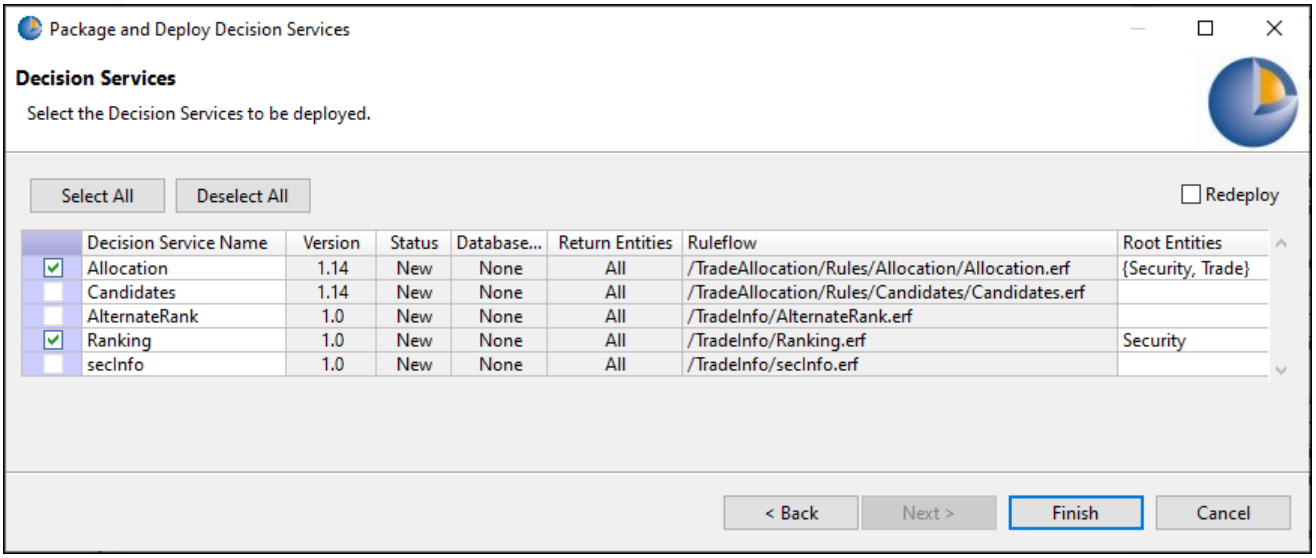
1. Enter the **Server URL** of the Corticon Server.
2. Enter the **Username** and **Password** for the server. For administrative permissions on the application server, try the default credentials `admin` and `admin`.
3. Click **Test Connection**.

   - 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 can 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.
   - **Root Entities** for the one or more (CTRL+click) entities that are the top-level entities of the Decision Service. When you specify the top-level entities in the JSON payload, those entities allow the Decision Service to map the payload to the vocabulary much more efficiently.

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 **Compilation Success** message. If there are problems, the wizard lists the errors.

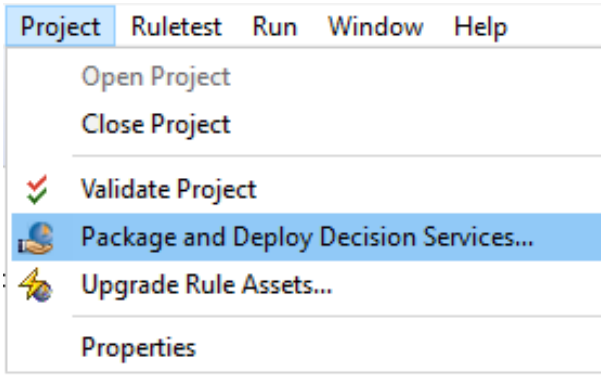
Deploy to Corticon Web Console

You can deploy from Studio to servers managed by a Web Console, a distinct installation option as described in *"Installing Corticon Web Console" in the Installation Guide*.

The Corticon Web Console provides a browser UI for managing and monitoring your Corticon Servers. Deploying to the Web Console from Studio will add the Decision Services to an Application on the Web Console and deploy them to the Corticon Servers specified for the Application in the Web Console.

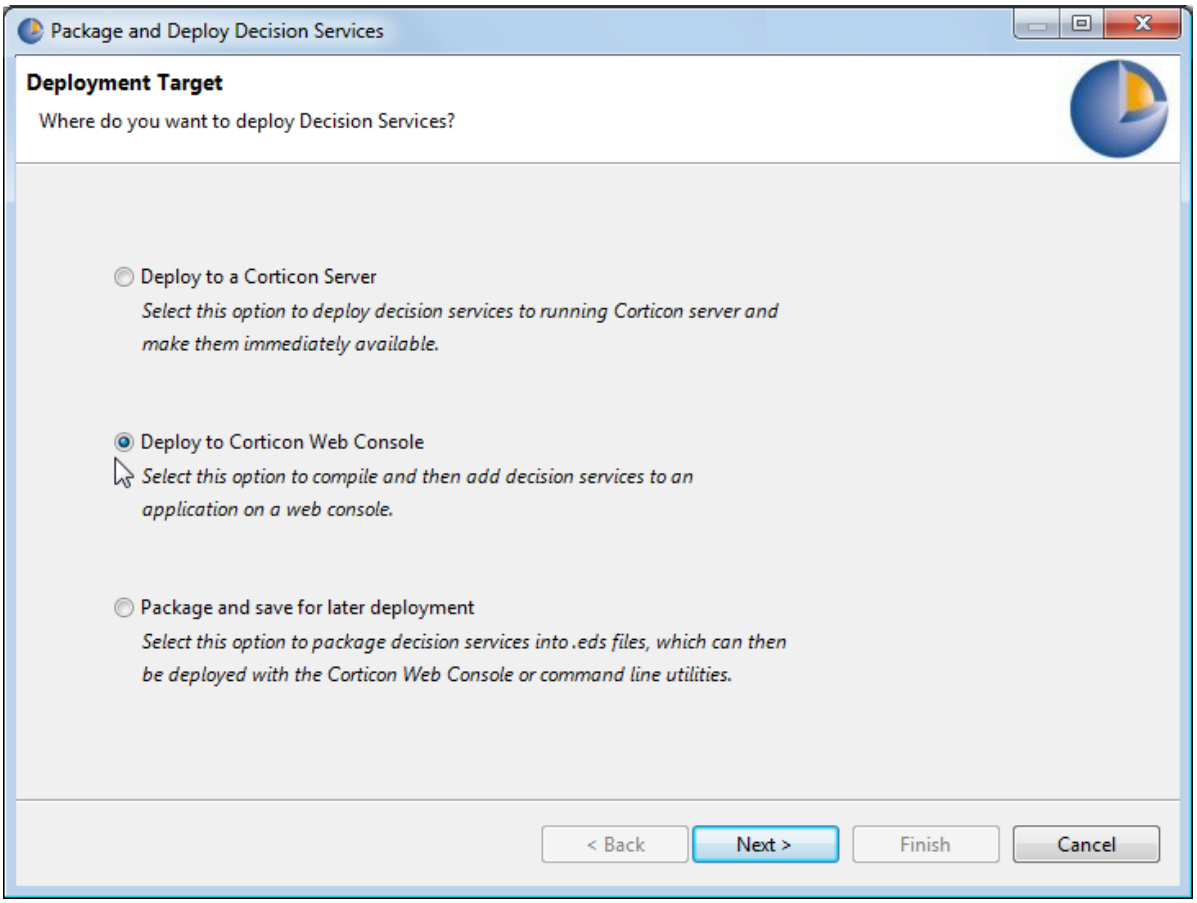
To deploy Ruleflows in Corticon Studio as Decision Services on servers managed by the Web Console:

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**:

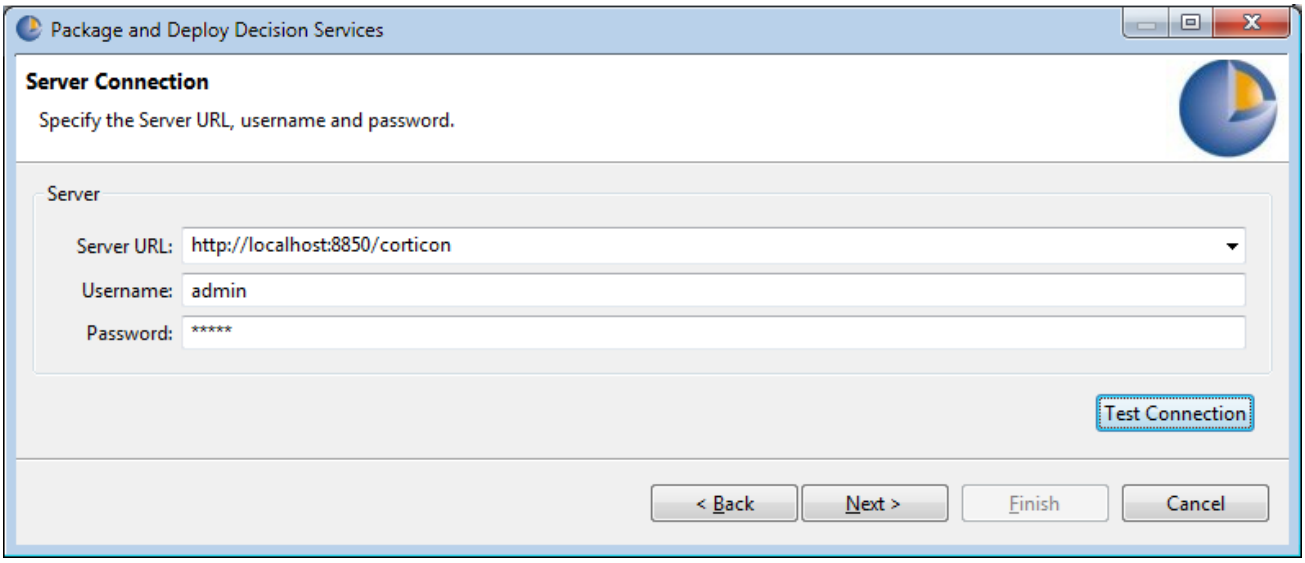


When a project is selected or there is an active file in its editor, the Ruleflows of only that project will be listed. When no projects are selected and no files are in their editor, the Ruleflows of all projects in the workspace will be listed. If one or more Ruleflows are selected in Project Explorer, only those Ruleflows will be displayed in the Package and Deploy wizard. If you have many Ruleflows in a project, you may want to organize them into folders to make it easy to identify the main Ruleflows which should be deployed.

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



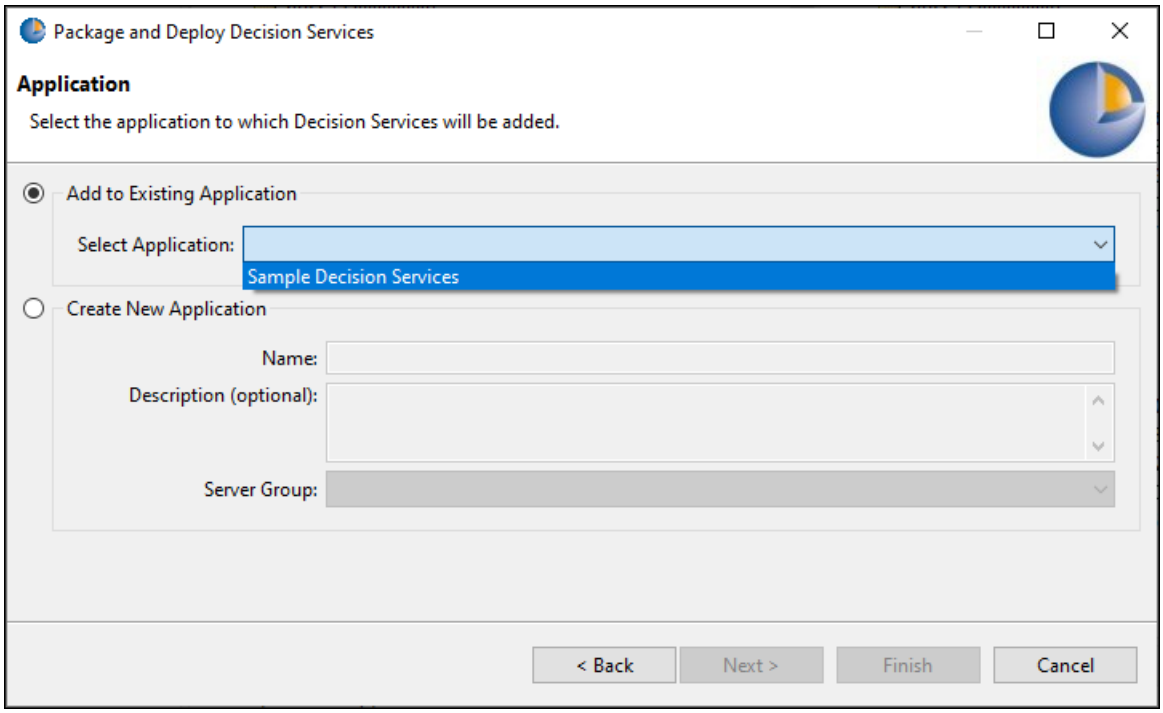
4. Click **Next**.
5. Enter the server connection URL with its port and `/corticon`, then the username and password for that Web Console. The administrative username is `admin` with the initial password `admin`.



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

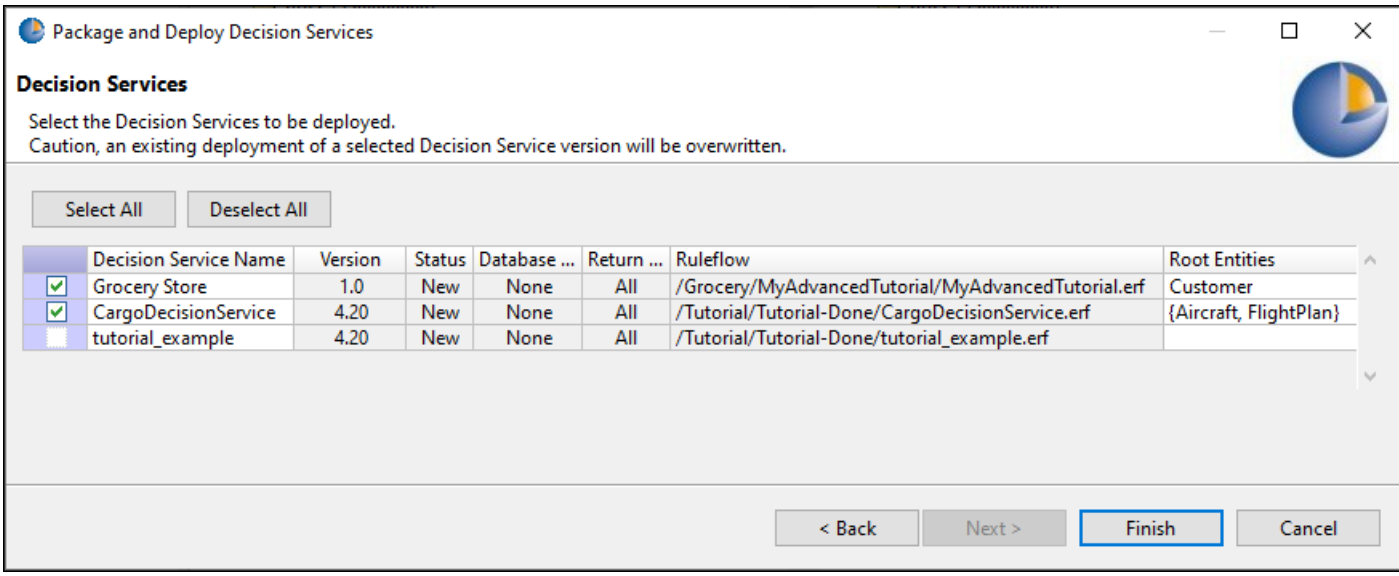
6. Select whether to use an existing Application or to create a new one:
   - To add to an existing Application, choose **Add to Existing Application**, select an Application on the pull-down list, and then click **Next**.

- To create a new Application, choose **Create New Application**, and then enter a new Application name and its description.



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

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



Select the Ruleflows to deploy as Decision Services.

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

- The wizard copies the **Decision Service Name** from the Ruleflow file name.
- **Version** from the Ruleflow properties. This field can only be changed in the Ruleflow editor.
- **Database mode** for a Ruleflow that will have a database connection.
- **Ruleflow** location within the current workspace.

- **Root Entities** are the one or more top-level entities in the payload.

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.

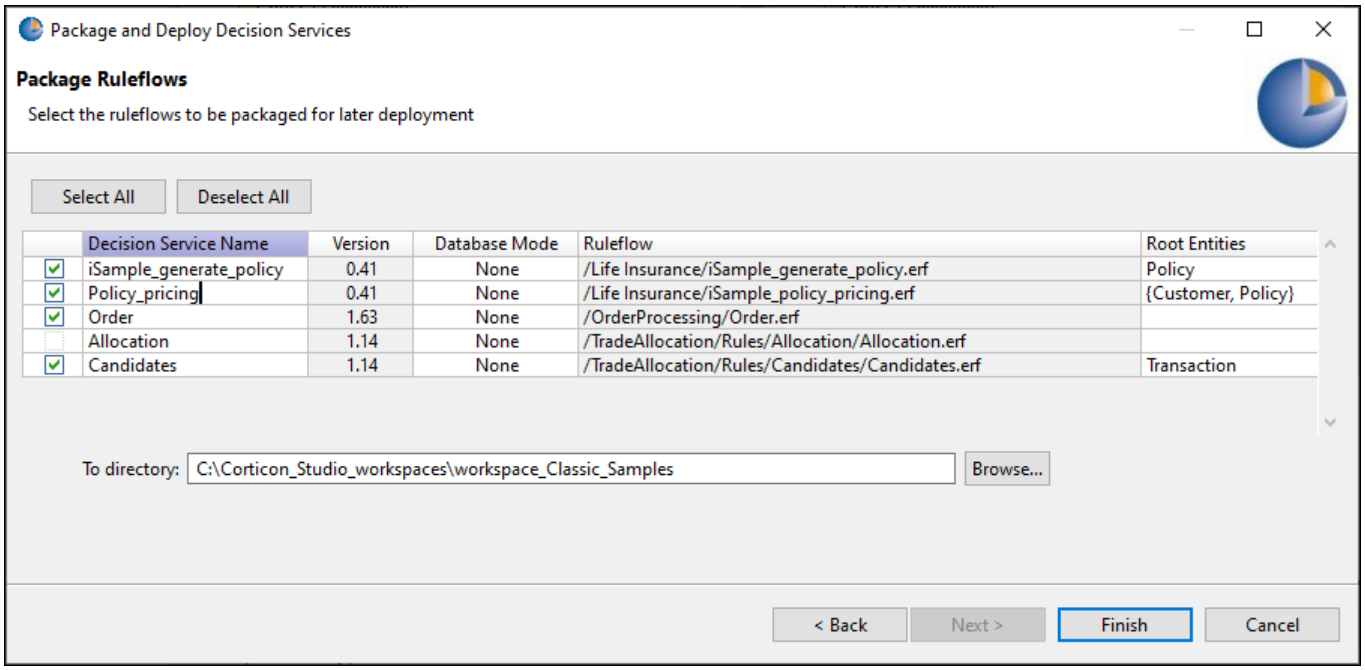
When your selections are complete, click **Finish**.

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.

Package and save 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 for deployment by the Corticon Web Console, an option in a Web Console, a distinct installation option as described in *"Installing Corticon Web Console" in the Installation Guide*. When you start the Corticon Server, it starts the Web Console Server (`corticon.war`).

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.
- **Version** from the Ruleflow properties. This field can only be changed in the Ruleflow editor.
- **Database mode** for a Ruleflow that will have a database connection.
- **Ruleflow** location within the current workspace.
- **Root Entities** are the one or more top-level entities in the payload.

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.

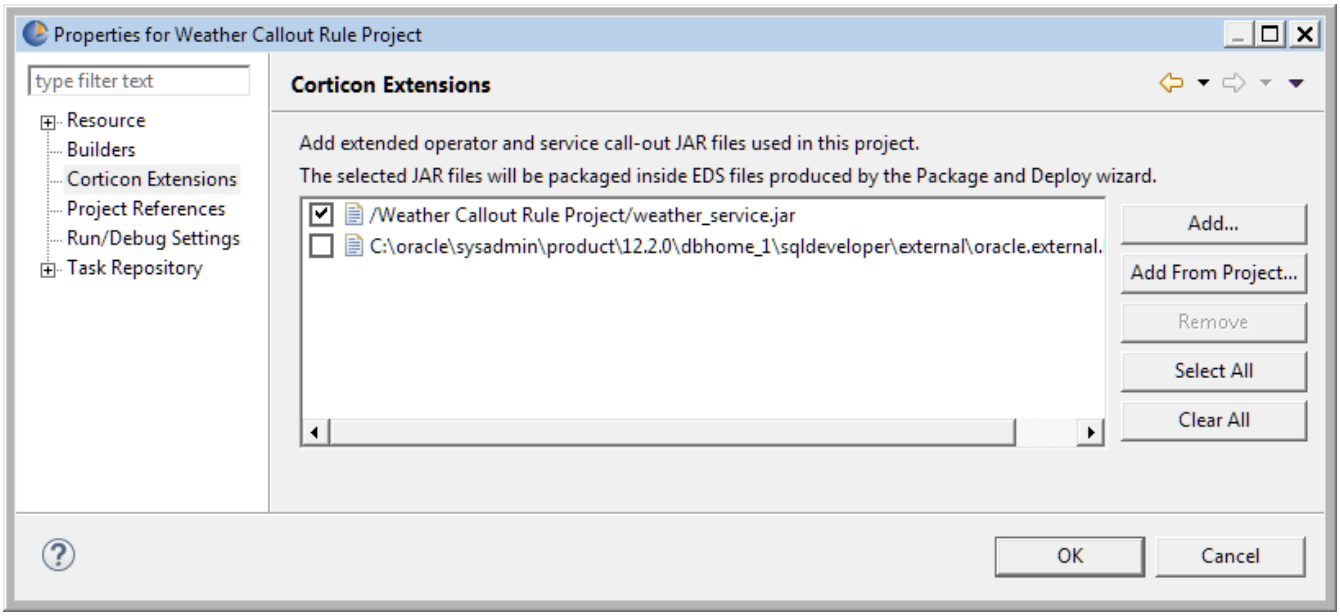
The packaged Decision Service can be deployed the either the Web Console *"Decision Services and Applications" in the Web Console Guide* or the `CorticonWebConsole` command line utility described in *"Deploying a Decision Service" in this guide*

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, 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.

Use 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 *"Decision Services and Applications" in the Web Console Guide.*

Use Deployment Descriptors to deploy Decision Services

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

You can them with the Corticon management command line utility, or create them in a text editor.

Open a `.cdd` file in a text editor to see how it is formatted. The `TradeAllocation.cdd` sample located at a server installation's `[WORK_DIR]\Samples\Rule Projects\Trade Allocation` is a good example.

Structure of a Deployment Descriptor 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>AllocateTrade</name>
     <path>AllocateTrade.eds</path>
     <options>
        <option name="PROPERTY_AUTO_RELOAD" value="true" />
        <option name="PROPERTY_MAX_POOL_SIZE" value="1" />
     </options>
  </decisionservice>

  <decisionservice>
      <name>Candidates</name>
      <path>Candidates.eds</path>
      <options>
         <option name="PROPERTY_AUTO_RELOAD" value="true" />
         <option name="PROPERTY_MAX_POOL_SIZE" value="1" />
      </options>
  </decisionservice>

</cdd>
```

For each Decision Service you must specify a name and the EDS file. The options, if any, specified allow you to configure properties of a Decision Services. A CDD file can contain one or more Decision Services.

Note: The path names to the 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 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.

How to set 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 in CDD file are set in name-value pairs, as shown:

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

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

Option name	Value
PROPERTY_AUTO_RELOAD	false | true
PROPERTY_MAX_POOL_SIZE	1 | [positive integer]
PROPERTY_MESSAGE_STRUCTURE_TYPE	<null> | HIER | FLAT
PROPERTY_DATABASE_ACCESS_MODE	<null> | R | RW
PROPERTY_DATABASE_ACCESS_RETURN_ENTITIES_MODE	ALL | IN
PROPERTY_DATASOURCE_CONFIG_FILE_PATH	[explicit or relative path] to datasource.xml
PROPERTY_DATABASE_ACCESS_CACHING_ENABLED	false (default) | true
PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_INFO	false | true
PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_WARNING	false | true
PROPERTY_EXECUTION_RESTRICT_RULEMESSAGES_VIOLATION	false | true
PROPERTY_EXECUTION_RESTRICT_ RESPONSE_TO_RULEMESSAGES_ONLY	false | true

Note: The default values of the RULEMESSAGES options can be overridden by settings in a server's `brms.properties` file. However, a payload can dynamically override these properties for each execution by adding execution properties to its payload.

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_DATASOURCE_CONFIG_FILE_PATH"
                   value="../datasource.xml" />
         <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.eds</path>
      <options/>
   </decisionservice>
</cdd>
```

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

- There are two `<decisionservice>` sections.
- 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 Apache Tomcat to test and deploy your Decision Service, 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 that list deployment of ERF files, those ERF files must be compiled into EDS files prior to deployment. You can generate EDS files with Corticon Studio, Corticon command line utilities, or Corticon ant macros.

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.ccserver.autoloaddir=path
```

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

Automate packaging and testing of 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
- Run Ruletests to validate that the Decision Services perform as expected
- Export a decision service as a JSON file

Note: The `corticonManagement` utilities are installed by the distinct installer `PROGRESS_CORTICON_7.x_UTILITIES_WIN_64.exe` that defaults to locating the utilities at `C:\Progress\Corticon 7.x\Utilities`. You need to install a license to enable it. To run the utility, choose **Start > Progress > Corticon Command Prompt,** and then navigate to the installation location's `bin` directory. Type `corticonManagement` to display its usage.

Note: The default process of Decision Service compilation does not include WSDL and a report in the EDS file. The selections in the Studio export Service contracts and reports that can be passed to system integrators as a `.wsdl` or `.xsd` text and a report `.html` file. If you want WSDL or reports in the Decision Service package, add the following lines to the `brms.properties` file where Corticon will do the compilation:

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

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.

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, run `C:\Progress\Corticon 7.x\Utilities\bin\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" />
<attribute name="dependentjars" default="" />
<attribute name="includedjars" default="" />
<attribute name="toplevelentities" default="" />
```

Example of a call to the compile macro:

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

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 " />
```

Syntax of the compile and test commands

The `corticonManagement` utilities are installed by the distinct installer `PROGRESS_CORTICON_7.x_UTILITIES_WIN_64.exe` that defaults to locating the utilities at `C:\Progress\Corticon 7.x\Utilities`. You need to install a license to enable it.

Note: If you did not apply a license during the installation of the Corticon Utilities, add a valid license at: `C:\Progress\Corticon_Utilities_Work_7.x\license\utilities`

Note: you might need to `set JAVA_HOME=C:\Program Files\Java\jdk-21`

To run the utility, open a command prompt at `C:\Progress\Corticon 7.x\Utilities\bin` Type `corticonManagement` to display its usage:

Table 7: usage: corticonManagement

Argument	Description
`-c,--compile`	Compile a Ruleflow into a Decision Service
`-cdd,--createcdd`	Create new CDD

Argument	Description
`-e,--extractDiagnostics`	Extract diagnostic data from a log file
`-en,--encryptstring`	Encrypt supplied string
`-g --genJson`	Generate JSON from Rule Asset
`-h,--help`	Print this message
`-m,--multicompile`	Compile Ruleflows in the specified input file
`-r,--report`	Generate the report for a Corticon asset
`-s,--schema`	Generate the WSDL/XSD schema for a vocabulary or Ruleflow
`-t,--test`	Execute tests for a Ruleflow or Rulesheet

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

Compile 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 8: usage: corticonManagement --compile

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

Argument	Description
-ij,--includedjars *includedjar ...*	Required when using extensions. Explicit paths to JAR files (that are specified as dependentjars) to include in the generated EDS file, delimited by spaces.
-tle,--toplevelentities *EntityName, ...>*	Optional. List of top level Vocabulary Entity names for the Decision Service, delimited by commas.

Any values that contain spaces must be in quotes. For example:

```
-ij "C:\Program Files\myExtensions.jar" "C:\Program Files\myCallouts.jar"
```

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
   --toplevelentities Cargo,FlightPlan
```

Create a CDD file

The createcdd option create a Deployment Descriptor file (.cdd) for the specified Decision Service that can then be deployed to a Corticon Server.

Table 9: usage: corticonManagement --createcdd

Argument	Description
-a, --autoreload [*true*|*false*]	auto reload
-d, --datasource *path*.xml	path to datasource config file
-e, --edc [*null*|*R*|*RW*]	enable EDC (none or read only or read write access mode) in the decision service
-eds, --eds *path to eds folder*	path to the .eds file to be added to cdd
-el, --level2 [*true*|*false*]	enable level 2 caching
-ep, --edcproperties *path to .properties*	path to the edc runtime properties
-er, --entitiesreturn [*ALL*|*IN*]	entities return setting
-f, --file *path to cdd*	path to the cdd file that will be created with the extension .cdd
-h, --help	print this message

Argument	Description
`-m --messagestyle [HIER|FLAT|null]`	message style
`-n, --name`	name of the decision service
`-or, --onlymessages [true|false]`	response is only rulemessages
`-ri, --info [true|false]`	restrict info rulemessages
`-rv, --violation [true|false]`	restrict violation rulemessages
`-rw, --warning [true|false]`	restrict warning rulemessages
`-x, --max integer`	max pool size

A simple `-cdd` command might look like this:

```
corticonManagement --createcdd
   --file C:\outputStaging\myRules.cdd
   --eds  C:\myProject\myEDS_v1_1.eds
   --name myDecisionService
```

Extract Diagnostics

The `extractDiagnostics` option creates a text file of metrics extracted from a specified server's log for the Decision Service version.

Table 10: usage: corticonManagement --extractDiagnostics

Argument	Description
`-ds,--decisionservice Decision Service name`	Extract metrics for the specified Decision Service
`-dsv, Decision Service version`	Metrics for the specified Decision Service in *major.minor* format.
`-s --server`	Extract metrics for this server
`-i --input file`	Log file to extract metrics from
`-o --output file`	File to write in CSV format

Note: See *"Diagnose runtime performance" in the Server Guide.* for complete information on **diagnostic** data.

Encrypt a given String

The `encryptstring` option encrypts a String provided.

Table 11: usage: corticonManagement --encryptstring

Argument	Description
`-i --input`	The String that will be encrypted

Export a decision service as a JSON file

The `genJson` option lets you export a decision service as a JSON file. A valid license is required to run `gen.JSON` in Corticon Utilities.

Table 12: usage: corticonManagement --genJson

Argument	Description
`-i <.erf>`	Required. Path to the ruleflow (`.erf`) file of the decision service.
`-o <.json>`	Path to the output (`.json`) file to be generated.

The generated JSON will contain a full representation of the decision service, including all rulesheets, rules, and related attributes. The following assets must be accessible to the utility:

- Vocabulary (`.ecore`)
- Rulesheets (`.ers`)
- Nested ruleflows (`.erf`)

If the export is successful, the utility exits with code 0. If the export fails, the utility exits with a non-zero code, and error messages are output to the console and potentially logged for further troubleshooting.

Format of the Generated JSON

The generated JSON contains only the functional aspects of the decision service (rulesheets, rules, conditions, actions, vocabulary, etc.). Formatting information (such as UI-related or presentation data) is not included. The format is consistent with the JSON used by the Corticon AI Assistant.

SchemaGet the schema for the JSON representation of the ruleflow at `C:\Progress\Corticon 7.x\Utilities\Doc`.

Usability in CI/CD

The utility integrates smoothly into CI/CD processes, enabling users to export decision services in JSON format for reporting as part of their deployment pipelines.

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

Example usage:

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

Note: Multicompile uses your `brms.properties` for optional overriding of properties when compiling a decision service. For more information, see the topic *"Properties that impact Decision Service compilation" in the Corticon Deployment Guide*.

Note: For optimal performance, be sure that all the assets and projects that will be involved in the multiple compile processing have been upgraded to the same version.

Template

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

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

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

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

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

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

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

Generate a report on an asset

The `report` option generates a report on an asset (Vocabulary, Rulesheet, Ruleflow, or Ruletest), and then puts the report into a specified output folder.

Table 14: usage: corticonManagement --report

Argument	Description
`-p,--project` *`project name`*	Optional. Corticon project name to use in the report.
`-i,--input` *`file`*	Required. Explicit path to the asset.
`-dj,--dependentjars` *`dependentjar, ...`*	Optional. Explicit paths to JAR files required for this report, delimited by commas.
`-if,--image` *`image folder`*	Optional. Explicit path to the image folder that will be copied to the output folder.
`-c,--css` *`CSS file`*	Required. CSS file to use for the report
`-x, --xslt` *`XSLT file`*	Required. XSLT file to use for the report
`-o,--output` *`folder`*	Required. Explicit path to the output folder.

Any values that contain spaces must be in quotes. For example:

```
 -dj "C:\Program Files\myExtensions.jar,C:\Program Files\myCallouts.jar"
```

A complete command might look like this:

```
 corticonManagement
    --report
    --project myProject
    --input C:\myProject\myRuleflow.erf
    --dependentjars C:\myProject\myExtensions.jar,C:\myProject\myCallouts.jar
    --image C:\myReports\myPreferredImages
    --css "C:\myReports\CSS\Corticon Blue.css"
    --xslt "C:\myReports\XSLT\Detailed Ruleflow Expressions.xslt"
    --output C:\myReports
```

Generate WSDL and XSD schema files

The `schema` option generates WSDL and XSD schema files from either a Ruleflow (.erf) or Vocabulary (.ecore) file. This is the same functionality that is provided in the Studio Ruleflow and Vocabulary menu commands that export WSDL and export XSD. See How to integrate Corticon Decision Services on page 73 for more information on service contracts.

Table 15: usage: corticonManagement --schema

Argument	Description
`-dj,--dependentJars <dependentJar>`	Add jar files required by this Ruleflow.
`-h,--help`	Print this message.
`-i,--input` *`file`*	Required. The source Vocabulary (.ecore) file for a vocabulary-level schema, or Ruleflow (.erf) file for a Decision Service-level schema.
`-m,--messagestyle [`*`FLAT`*`|`*`HIER`*`]`	Optional. Specifies whether the messaging style should be flat, hierarchical. When omitted, defaults to auto-detect where the schema generator determines the best option.
`-n,--net` *`name`*	Create a .NET schema.
`-o,--output` *`folder`*	Required. Explicit path to the output folder.
`-s,--service` *`name`*	Required for a Ruleflow. The name of the Decision Service for the schema.
`-t,--type [`*`WSDL`*`|`*`XSD`*`]`	Required. Type of schema to generate.
`-u,--url` *`address`*	Required. Specifies the Server URL to set in the schema document. This URL should match the URL of the Decision Service when deployed so that clients using the schema have the correct 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 MyDS


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

Testing a Decision Service with a Ruletest

The `test` option executes multiple Ruletest (.ert) files and their test sheets, and produces an output file with the test results.

Table 16: usage: corticonManagement --test

Argument	Description

`-i,--input` *`file`*`,...`	Required. A comma-separated list of the source Ruletest `.ert` files to run.
	• If there are spaces in the filenames or paths, then the entire set needs to be in quotes. Whitespace surrounding the comma character is valid. For example:
	`--input "C:\test\file1.ert , C:\test\file2.ert"`
	• Wildcards can be used in the filename to allow greater flexibility in specifying multiple files with similar names. The wildcards (? and *) can only be used in the filename portion of the path and not any other parts of the path. Wildcards can be used in combination with the comma separated list. For example:
	`--input "C:\test\file?.ert , C:\test2\*.ert"`
	• IMPORTANT: When the `input` option resolves to multiple files, then the `sheet` option cannot be used and the `all` option must be specified.

`-o,--output` *`file`*	Required. Explicit path to the preferred output folder and existing file name (an XML file in the JUnit test output style that includes pass/fail, test suite, test file, and execution time.) The output file is never overwritten; instead, 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.
`-a,--all`	Required unless `--sheet` is stated. Runs tests for all the testsheets in the specified Ruletest in the order that they are defined in the file. Overrides any specific testsheets listed in the sheet option.
`-s,--sheet` *`sheet_names`*	Required unless `--all` is stated. Runs tests for only the one or more (in a comma-separated list) specified testsheets in the Ruletest in the order that they are listed.
`-dj,--dependentjars` *`dependentJar1`*`,...`	Comma separated list of dependent JAR paths. These are extra jar files that are needed to run the tests, such as those for extended operators, SCOs, ADC, and REST.

Argument	Description
`-ll,--loglevel` *`level`*	Sets the log level to the specified level of detail. Defaults to current server log level, typically `INFO`. Choosing `DEBUG` is verbose.
`-lp,--logpath` *`path`*	Explicit path to the folder where `CcManagement.log` will be saved. Defaults to the server's current log location, typically `[CORTICON_WORK_DIR]/logs`.

Example usage:

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

corticonManagement -t
                   -i "C:\MyTest.ert , C:\moreTests\*.ert"
                   -o  C:\MyTest_out.xml
                   -a


corticonManagement -t -a
                   -i "C:\Users\me\workspace\Generic\add*.ert,C:\test bed\base?.ert"
                   -o C:\testOutput\Output.xml
                   -ll DEBUG
```

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: If you did not apply a license during the installation of the Corticon Utilities, add a valid license at: `C:\Progress\Corticon_Utilities_Work_7.x\license\utilities`

Note: you might need to `set JAVA_HOME=C:\Program Files\Java\jdk-21`

To run the utility, open a command prompt at `C:\Progress\Corticon 7.x\Utilities\bin`, and then 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,--decisionservice        Perform actions on a decision service
 -h,--help                    print this message
 -login,--login               initiate login to web console
 -logout,--logout             initiate logout from web console
```

Lists the syntax of the commands.

corticonWebConsole -login

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

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

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

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

---

corticonWebConsole -logout

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

The logout command closes the connection to the Web Console.

corticonWebConsole -ds

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

corticonWebConsole -ds -add

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

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

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

corticonWebConsole -application

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

corticonWebConsole -application -deploy

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

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

corticonWebConsole -application -undeploy

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

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

Use 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.

For more information, see the *Corticon Server API* topics in the *Corticon Server* section, and accessible online at https://documentation.progress.com/output/Corticon/6.3/javadoc/Server/index.html .

Properties that impact Decision Service compilation

Corticon Server uses the following properties when compiling assets into a Decision Service through its packaging utilities. (Corticon Studio also uses these properties in its “Package and Deploy” wizard when compiling a Decision Service.) The following properties are settings you can apply to your Corticon Studio and Server installations by adding the properties and appropriate values as lines in its `brms.properties` file.

Note: Changing these properties can dramatically affect compilation time and the package size, especially with large Ruleflows. Compile time might be doubled for Ruleflows with more than 1500 rules when you include the Ruleflow report in the compiled EDS file. Another 10% of compile time might be added when WSDL is generated. As an alternative, there are techniques for producing WSDL and reports in Studio. (In prior releases, the default action was to automatically produce and add the WSDL and reports to the EDS.)

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. Setting the value to `true` creates the asset report, and includes it in the EDS file which will increase the EDS file size and the compilation time significantly.

Default value is `false`

```
com.corticon.server.compile.add.report=false
```

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. Setting the value to `true` creates the WSDL, and includes it in the EDS file which will increase the EDS file size and the compilation time significantly.

Default value is `false`

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

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

Default value is `-Xms256m -Xmx1g`

```
com.corticon.ccserver.compile.memorysettings=-Xms256m -Xmx1g
```

Properties that are incorporated 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. These setting are 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.

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

For information about related properties

See also:

- Properties that are incorporated into Decision Services on page 63
- *"Server execution properties"* in the *Server Guide*.
- *"Server build properties"* in the *Server Guide*.
- Properties that tune service contract output on page 77

4

How to deploy Corticon on Docker

Docker is a popular platform for building, sharing and running applications. Since the Docker container packages applications with all their libraries and dependencies, the application can run on any Docker host in any supported environment. Among the benefits of this strategy are:

- Docker containers are faster to create and quicker to start.
- Docker scales efficiently when demand for your application grows.
- Corticon Server deployed to Docker as a web application can be accessed as a REST or a SOAP service.

Note: Prepare to deploy Corticon Server to Docker by getting and running Docker for your platform at https://docs.docker.com/get-docker/. Remember to check that Docker is running every time you intend to use your repository.

Configure Corticon Docker

A Corticon Server install provides the Docker configuration file, `Dockerfile`, and the `.war` files for Corticon Server and Corticon Web Console.

For download information, see *"Download installer packages" in the Installation Guide*.

Corticon Server on Docker

1. Navigate to: `C:\Progress\Corticon 7.2\Server\Deploy\Docker`.
2. Edit `Dockerfile`. The Corticon configuration is pre-configured to deploy Corticon Server to a base Tomcat 9.0 image. You can adjust and add to the configuration to suit your installation.
3. Open the Server's `Dockerfile` in your preferred text editor:

```
###Copyright 2005-2024 Progress Software Corporation and/or its subsidiaries and
affiliates. All rights reserved.

# Specify the tomcat image version to pull from DockerHub
FROM tomcat:9.0-jre17
##This will derive the latest 9+ image of Tomcat with jre 17 version

USER root

# Open port 8080
EXPOSE 8080

################Setting Corticon Home and Work Directory ########################
##Create the directory CORTICONWORK
#RUN mkdir /usr/local/tomcat/CORTICONWORK

##SET CORTICON HOME
#ENV CATALINA_OPTS="$CATALINA_OPTS -DCORTICON_HOME=/usr/local/tomcat/CORTICONWORK"

## Similarly you can SET CORTICON WORK DIR using the -DCORTICON_WORK_DIR
#ENV CATALINA_OPTS="$CATALINA_OPTS -DCORTICON_WORK_DIR=<PATH_TO_WORK_DIR>"

############################################################

## Copy Corticon war files to work directory
COPY axis.war /usr/local/tomcat/webapps

########### Updating Corticon License ###################

##Create a Directory to copy the license
RUN mkdir /usr/local/tomcat/LicenseCorticon

##Copy the CcLicense.jar to the directory created
#COPY CcLicense.jar /usr/local/tomcat/LicenseCorticon

##Set the environment variable to update the license for the Corticon Server
#ENV CATALINA_OPTS="$CATALINA_OPTS
-DCORTICON_LICENSE=/usr/local/tomcat/LicenseCorticon/CcLicense.jar"

#################################################################
## Create Deploy and cdd folders
RUN mkdir /usr/local/Deploy
RUN mkdir /usr/local/tomcat/cdd

## To deploy Decision services, copy the eds file and cdd file to the CDD directory
#ADD Order.eds /usr/local/tomcat/cdd/
#ADD OrderProcessing.cdd /usr/local/tomcat/cdd/

# To enable Corticon Server to use the brms.properties, copy this file to the Corticon
 Work dir
#ADD brms.properties /usr/local/tomcat/
```

The predefined settings for Corticon Server on Docker are as follows:

- `FROM` statement to specify the base image to be used for this container. The docker file bundled uses a public `tomcat9` version and `jre17` version, imaged from Docker hub. You can choose any other supported image from docker hub that is supported in the Corticon release or build your own. Refer to the Progress Software web page Corticon Supported Platforms Matrix to review the currently supported Corticon Studio operating systems, and Eclipse versions.
- `USER` is set to `root` to ensure that the container has full control of the host system.
- `EXPOSE` lets you specify the network port on which Docker will listen at runtime.
- `COPY` The next step copies `axis.war`, `tomcat`.
- `RUN mkdir` sets up the required directories.
- `#ADD Order` installs the Decision Service file and the corresponding CDD. Remove the # tag to enable the actions.

Corticon Server on Web Console

1. Navigate to: `C:\Progress\Corticon 7.2\WebConsole\Deploy\Docker`.
2. Edit `Dockerfile`. The Corticon configuration is pre-configured to deploy Corticon Server to a base Tomcat 9.0 image. You can adjust and add to the configuration to suit your installation.
3. Open the Server's `Dockerfile` in your preferred text editor: Open the Server's `Dockerfile` in your preferred text editor:

```
###Copyright 2005-2024 Progress Software Corporation and/or its subsidiaries and
affiliates. All rights reserved.

# Specify the tomcat image version to pull from DockerHub
FROM tomcat:9.0-jre17
##This will derive the latest 9+ image of Tomcat with jre 17 version

USER root

# Open port 8080
EXPOSE 8080

################Setting Corticon Home and Work Directory ########################
##Create the directory CORTICONWORK
#RUN mkdir /usr/local/tomcat/CORTICONWORK

##SET CORTICON HOME
#ENV CATALINA_OPTS="$CATALINA_OPTS -DCORTICON_HOME=/usr/local/tomcat/CORTICONWORK"

## Similarly you can SET CORTICON WORK DIR using the -DCORTICON_WORK_DIR
#ENV CATALINA_OPTS="$CATALINA_OPTS -DCORTICON_WORK_DIR=<PATH_TO_WORK_DIR>"

############################################################

## Copy Corticon war files to work directory
COPY corticon.war /usr/local/tomcat/webapps


############################################################
####### Web Console (corticon.war) specific settings ######

##Create a etc directory
#RUN mkdir /usr/local/tomcat/CORTICONWORK/etc

## Copy the logback.xml to CORTICONWORK Directory
#COPY logback.xml /usr/local/tomcat/CORTICONWORK/etc

## Copy the CorticonServerConsoleConfig.groovy to CORTICONWORK Directory
#COPY CorticonServerConsoleConfig.groovy /usr/local/tomcat/CORTICONWORK/etc
```

The settings are similar to the preceding ones for Corticon Server on Docker.

Setting Corticon Home and Work Directory

You can set the Corticon Home and Corticon Work directory in the Docker file to point to a preferred location:

1. Add the lines:

```
RUN mkdir /usr/local/tomcat/CORTICONWORK
ENV CATALINA_OPTS="$CATALINA_OPTS -DCORTICON_HOME=/usr/local/tomcat/CORTICONWORK/"
ENV CATALINA_OPTS="$CATALINA_OPTS -DCORTICON_WORK_DIR=/usr/local/tomcat/CORTICONWORK/"
```

2. Adjust the `ADD` to ensure that preferences are set in a `brms.properties` file are copied to the Corticon Work Directory.

Update the Corticon license

You need to copy the license to the container and then update the environment variable.

To add or update the license:

1. Locate your valid Corticon license for the Server/Web Console version.
2. Add the lines:

```
RUN mkdir /usr/local/tomcat/LicenseCorticon
COPY CcLicense.jar /usr/local/tomcat/LicenseCorticon
ENV CATALINA_OPTS="$CATALINA_OPTS
-DCORTICON_LICENSE=/usr/local/tomcat/LicenseCorticon/CcLicense.jar"
```

   After the configuration is complete, you need to provide the files you specified, and then build the docker image.

Provide specified files

After the preceding steps, you need to copy the specified files into the Docker folder, `PROGRESS_CORTICON_X.x_DOCKER`:

- `CcLicense.jar` from `C:\Progress\Corticon_Server_Work_7.2\license\Server\`
- `brms.properties` from `C:\Progress\Corticon_Server_Work_7.2`
- `ProcessOrder.eds` and `OrderProcessing.cdd` from the `Order Processing` sample.

Build the Docker container image from the Dockerfile

1. Open a Command Prompt window as administrator to the root of the Docker installation.
2. If you are replacing an existing image, type:

```
docker container prune
```

   and then type `y` to complete the deletion.

3. Then, for the new image where `corticondemo` is your preferred name, type:

```
docker ps -a
docker build -t corticondemo .
```

For more Docker documentation and options supported with the Docker build command, reference the docker documentation at https://docs.docker.com/.

Run the Docker container

1. Open a Command Prompt window as administrator to the root of the Docker installation.
2. Type:

```
docker run -p 8080:8080 corticondemo
```

Note: Docker has many options for `run`. See https://docs.docker.com/engine/reference/commandline/run/.

The Tomcat console output shows the server startup log lines.

Confirm the Server operation

Check the Corticon server by accessing the Corticon ping REST end point on the browser:

```
http://localhost:8080/axis/corticon/server/ping
```

When the server is running, the REST/GET request returns JSON with the uptime as:

```
{"uptime" : <time in milliseconds>}
```

To check that the license has been updated properly, you can check the server properties using the REST end point

`http://localhost:8080/axis/corticon/server/getProperties`

The REST response returns JSON with the Server properties. When the license has been updated, the `licenseFilePath` attribute in the JSON has the updated path:

```
"licenseFilePath":"/usr/local/tomcat/LicenseCorticon/CcLicense.jar",
```

In the Web Console, connect to the running server in this example as:

The Server Details view shows the license and the properties.

Other platforms

The steps needed to successfully deploy docker to a Tomcat server image can be transformed to other supported application servers listed in the Corticon Supported Platforms Matrix. To deploy Corticon to the different application servers, refer to the Corticon Knowledge Base entry Corticon Server 7.x sample WAR installation for different Application Servers for settings on specific application servers.

The configurations specific to deploying a web-app to a different image can differ based on the requirements of the image. However, the Corticon-specific configurations like setting the `CORTICON_HOME` and `CORTICON_WORK_DIR` or updating the license stay the same. You still must pass the environment variables like `-DCORTICON_LICENSE` to update the license or `-DCORTICON_HOME` to use a different location for Corticon home.

The Dockerfile can be customized to fit your business needs on any supported OS/application server Docker image.

5

How to integrate Corticon Decision Services

Calling a Decision Service means making an execute call to, or invocation of, Corticon Server. The topics in this section focus on the types of payloads, calls, their components, and the tools available to help you assemble them.

The call/invocation/request (these three terms are interchangeable) consists of:

- Name and version of the Decision Service to execute
- Data or payload to be processed by the rules in the Decision Service
- Location (URL) of Corticon Server if deployed as a web service

This topic is concerned more with protocol than with content. The focus of this topic is on Decision Service name and data payload. The name and location of a Corticon Server that you want to call are discussed in the *"How to invoke Corticon Server" in the Web Services Guide*.

Service Contracts: Describing the call

A *service contract* defines the interface to a service, thus letting the consuming client applications know 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 be analyzed by consuming applications, which can then generate, populate and send compliant service requests automatically.

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. The payload must match the contract.

Web Services standards define two such service contract formats, the Web Services Description Language, or WSDL and the XML Schema 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. REST services with JSON do not have a service contract.

Note: The default process of Decision Service compilation does not, by default, include WSDL and its report in the EDS file. The selections in the Studio export Service contracts and reports that can be passed to system integrators as a `.wsdl` or `.xsd` text and a report `.html` file. If you want WSDL or reports in the Decision Service package, add the following lines to the `brms.properties` file where Corticon will do the compilation:

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

For details, see the following topics:

- Service contract options
- Generate service contracts in Corticon Studio
- Display service contracts in Corticon Web Console

Service contract options

When creating a service contract you can specify the following options.

Note: The default process of Decision Service compilation does not include WSDL and its report in the EDS file. The selections in the Studio export Service contracts and reports that can be passed to system integrators as a `.wsdl` or `.xsd` text and a report `.html` file. If you want WSDL or reports in the Decision Service package, add the following lines to the `brms.properties` file where Corticon will do the compilation:

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

Decision Service name

The Decision Service name is used to identify a deployed Decision Service. By default, the name is the same as the Ruleflow used when the Decision Service was packaged into an EDS file. When deploying a Decision Service you can override this by specifying a different name.

Once deployed, the Decision Service will always be known, referenced and invoked in runtime by its 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.

What type? WSDL or XSD?

- **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 SOAP Web Services Server. The WSDL file defines a complete interface, including SOAP binding parameters, the Decision Service name, the payload (XML data elements required inside the request message), XML data elements provided within the response message. For more information on WSDL, see https://www.w3schools.com/xml/xml_wsdl.asp

Important: The Web Services standard allows for two messaging styles between services and their callers: **RPC-style** and **Document-style**. Document-style (also called Message-style) interactions are more suitable for Decision Service invocations 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. ***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.

- **XML Schema (XSD)** - The purpose of an XML Schema is to define the legal or valid structure of an XML document. 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 https://www.w3schools.com/xml/schema_intro.asp

What level? Vocabulary or Decision Service?

- **Vocabulary** - 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. That 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 all Decision Services that are built with that Vocabulary. This master service contract is referred to as **Vocabulary Level**.
- **Decision Service** - The issue with a 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 call 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.

Which messaging style? Flat or hierarchical?

There are two structural styles the payload can take. Flat payloads have every entity instance at the top, or root, level with all associations represented by reference. Hierarchical payloads represent associations with child entity instances indented within the parent entity structure.

- **FLAT** - Entity names start with an upper-case character, associations are represented by `href` tags, and role names are in lowercase initial characters.
- **HIER** - 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.

SOAP Server URL

The URL for the SOAP node that is bound to the Corticon Server is the SOAP Server URL. It is enabled only for WSDL service contracts. The default URL, `http://localhost:8850/axis/services/Corticon` (use your appropriate host name and port number), makes a Decision Service available to the Corticon Server's application server. This Deployment property's default value can be overridden in your `brms.properties` file as `com.corticon.deployment.soapbindingurl_1`.

Platform: Java or IIS?

Editing one line in a generated WSDL can make it conform to what IIS expects. Locate the line:

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

Replace it with the line:

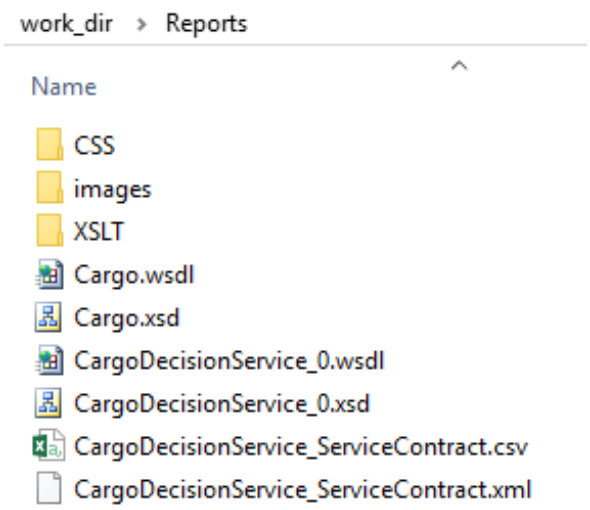
```
<CorticonRequest> <DecisionServiceName>"decisionServiceName"</DecisionService>
```

Note: Your client SOAP UI might make it a good idea to review Properties that tune service contract output on page 77 and Extended service contracts: newOrModified on page 79.

Service contract output

The output from all the service contract types for an example Vocabulary and Ruleflow are as follows:

Figure 30: Service contract output files



Note: If you run any one of the reports again, the file will be overwritten.

Generic service contract

You can create a generic service contract for a Ruleflow. This is not an enforced contract but a description of the inputs expected and outputs produced by a Decision Service generated from the Ruleflow. This is useful to document the "API" of a Decision Service or for use with the underlying mechanism such as REST/JSON that does not have an enforced contract. The generic service contract is output as a `.csv` file such that you can view it in Microsoft Excel or reporting tools that support `.csv` format.

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

Properties that tune service contract output

The following properties are settings for generating service contracts that you can apply to your Corticon installations by adding the properties and appropriate values as lines in their `brms.properties` file.

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

```
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 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
```

Controlling date and time format masks in the CSV files

You can specify date and time formats that you prefer in a generated CSV service contract by setting properties in the brms.properties file. The 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
```

Choose one or more masking properties you want to set, and then provide an appropriate mask value. See *"Formats for Date Time and DateTime properties" in the Rule Language Guide.* For example:

```
com.corticon.serviceContract.date.masking=MM-dd-yyyy
com.corticon.serviceContract.dateTime.masking=MM/dd/yyyy h:mm:ss a
com.corticon.serviceContract.time.masking=h:mm:ss a z
```

Add the lines you define to the `brms.properties` file on Corticon Studio instances and Corticon Servers (when using the `corticonManagement` utility.)

Extended service contracts: newOrModified

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 can handle the increased complexity before enabling it.

Extended datatypes when newOrModfied

If the `newOrModified` attribute is enabled, then the base XML datatypes must be extended to accommodate it. The following `complexTypes` are included in service contracts that make use of the `newOrModified` attribute.

ExtBooleanType

```
<xsd:complexType name="ExtBooleanType">
              <xsd:simpleContent>
                    <xsd:extension base="xsd:boolean">
                      <xsd:attribute name="newOrModified" type="xsd:boolean"
              use="optional" />
                    </xsd:extension>
                  </xsd:simpleContent>
              </xsd:complexType>
```

ExtStringType

```
<xsd:complexType name="ExtStringType">
                  <xsd:simpleContent>
                    <xsd:extension base="xsd:string">
                      <xsd:attribute name="newOrModified" type="xsd:boolean"
              use="optional" />
                    </xsd:extension>
                  </xsd:simpleContent>
              </xsd:complexType>
```

ExtDateTimeType

```
<xsd:complexType name="ExtDateTimeType">
                  <xsd:simpleContent>
                    <xsd:extension base="xsd:dateTime">
                      <xsd:attribute name="newOrModified" type="xsd:boolean"
              use="optional" />
                    </xsd:extension>
                  </xsd:simpleContent>
              </xsd:complexType>
```

ExtIntegerType

```
<xsd:complexType name="ExtIntegerType">
                  <xsd:simpleContent>
                    <xsd:extension base="xsd:integer">
                      <xsd:attribute name="newOrModified""type="xsd:boolean"
              use="optional" />
                    </xsd:extension>
                  </xsd:simpleContent>
              </xsd:complexType>
```

ExtDecimalType

```
<xsd:complexType name="ExtDecimalType">
                  <xsd:simpleContent>
              <xsd:extension base="xsd:decimal">
                      <xsd:attribute name="newOrModified"
              type="xsd:boolean" use="optional" />
                    </xsd:extension>
                  </xsd:simpleContent>
              </xsd:complexType>
```

Generate service contracts in Corticon Studio

Corticon Studio can produce your preferred type of service contract for the project's Vocabulary or a Ruleflow that uses the Vocabulary. The output files are generated to the Studio's `[WORK_DIR]\Reports` directory.

When you produce WSDL using this option, the WSDL is not included in the Decision Service package unless you set properties as described in the following note. The WSDL and report that you produce can be passed to system integrators as descriptive notes about the Decision Service.

Note: The default process of Decision Service compilation does not include WSDL and its report in the EDS file. The selections in the Studio export Service contracts and reports that can be passed to system integrators as a `.wsdl` or `.xsd` text and a report `.html` file. If you want WSDL or reports in the Decision Service package, add the following lines to the `brms.properties` file where Corticon will do the compilation:

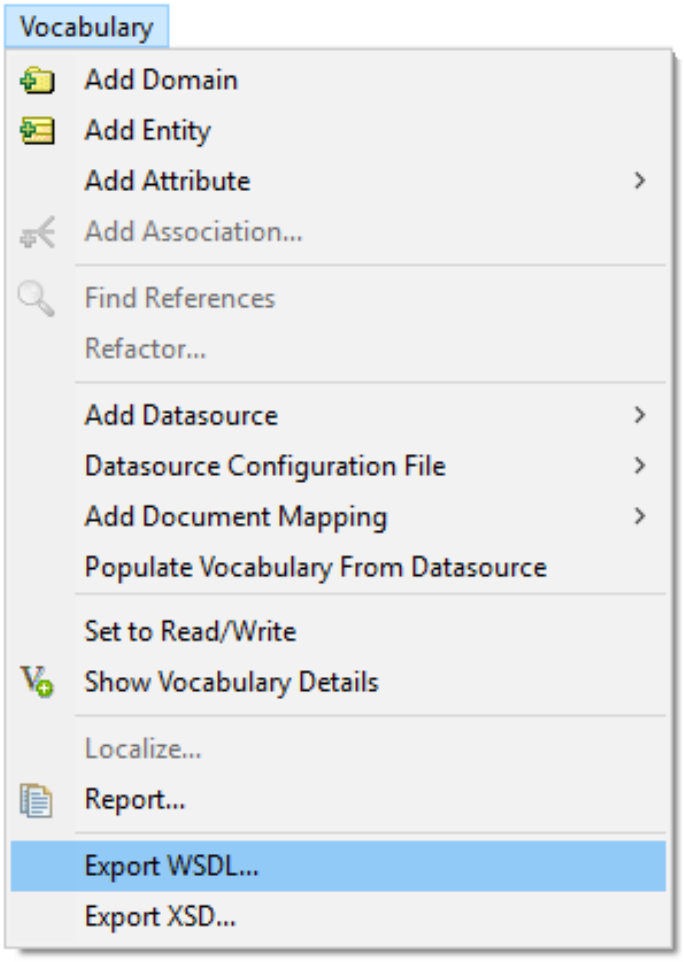
```
com.corticon.server.compile.add.wsdl=true
com.corticon.server.compile.add.report=true.
```

Generating Vocabulary level service contracts

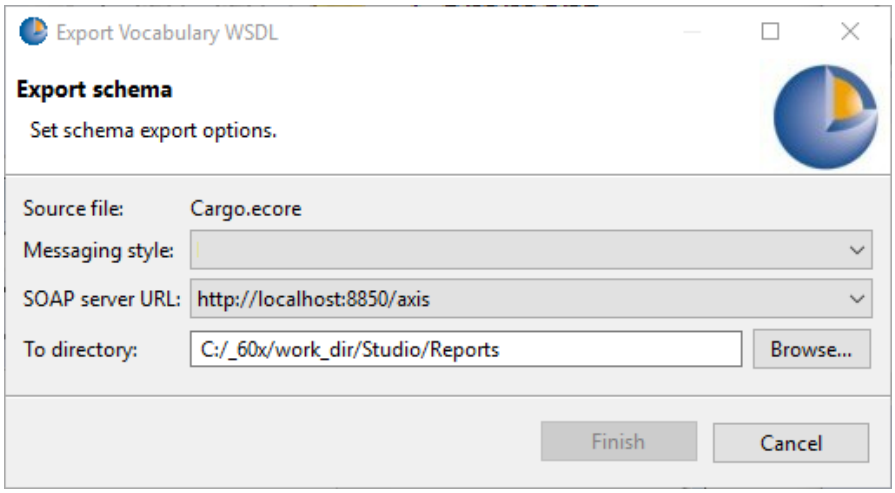
You can produce Vocabulary level service contracts as WSDL or XSD.

Vocabulary-level WSDL

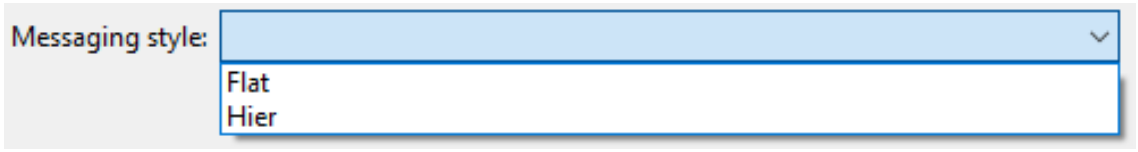
Select the Vocabulary you want. Open it in the editor. Choose the menu command **Vocabulary > Export WSDL**.



Its dialog box opens:



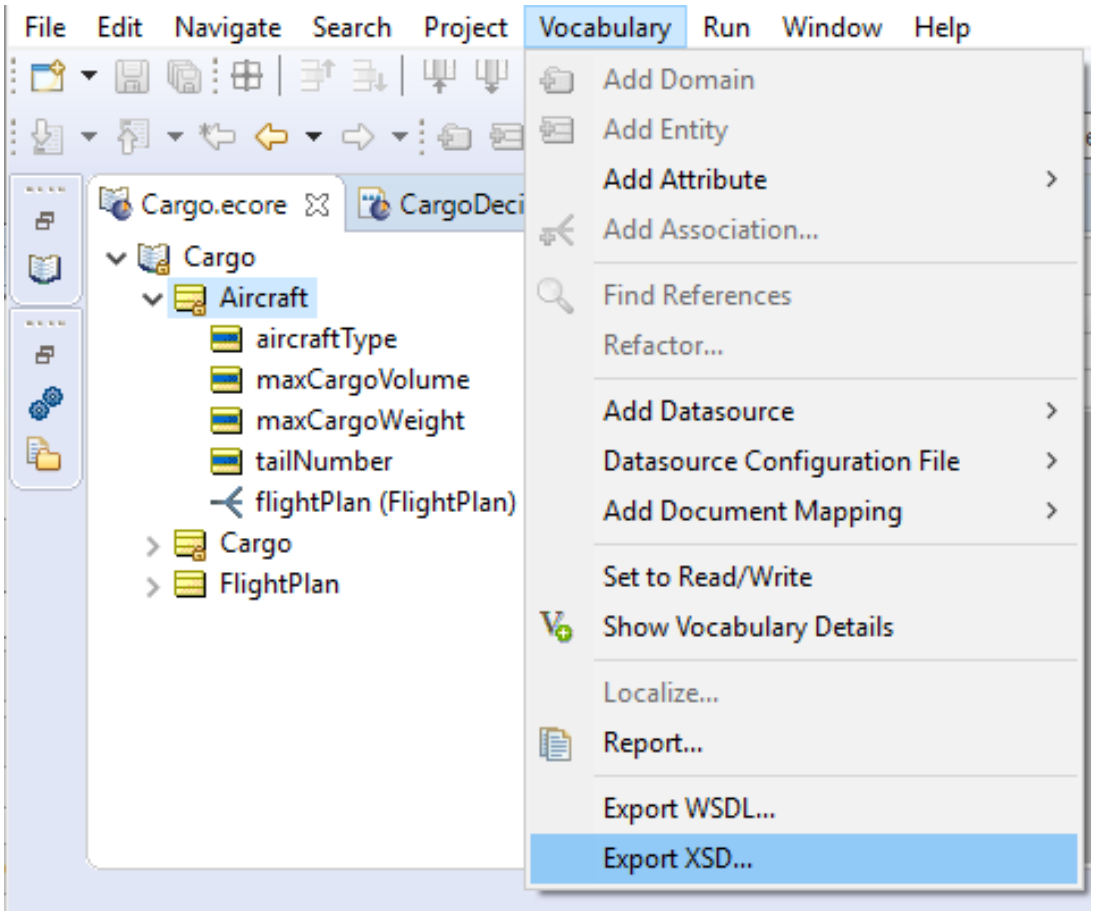
The **Messaging style** must be selected for the **Finish** button to be enabled. In the dropdown menu, choose **Flat** or **Hier**, as shown:



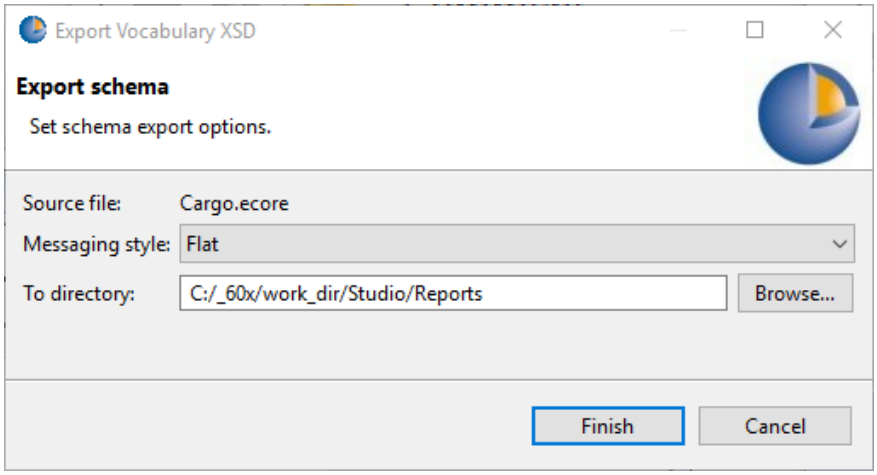
You can change the **To directory** to a preferred location. Click **Finish**. The report is generated and saved as *`VocabularyName`*`.wsdl` in the **To Directory** location.

Vocabulary-level XSD

Select the Vocabulary you want. Open it in the editor. Choose the menu command **Vocabulary > Export XSD**.



Its dialog box opens:



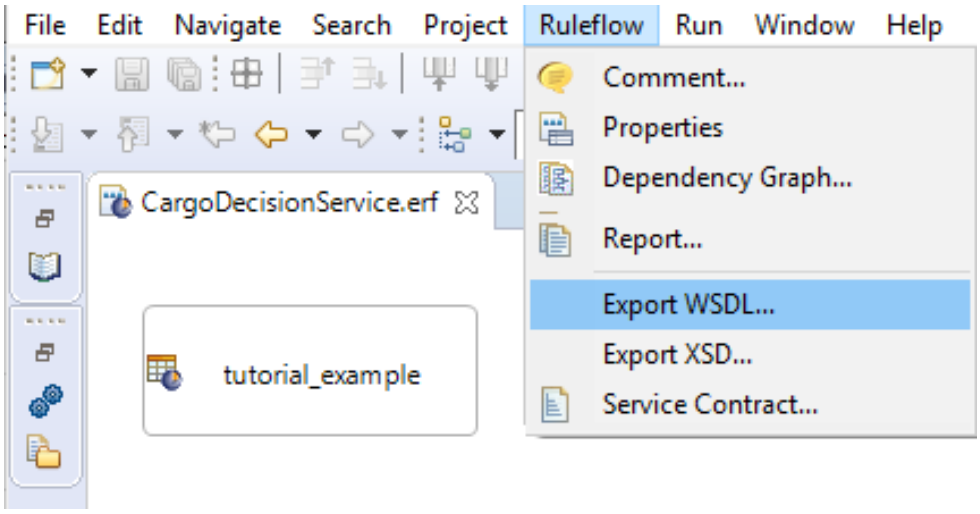
In the **Messaging style** dropdown menu, choose **Flat** or **Hier**Click **Finish**. The report is generated and saved as *`VocabularyName`*`.xsd` in the **To Directory** location.

Generating Decision Service level service contracts

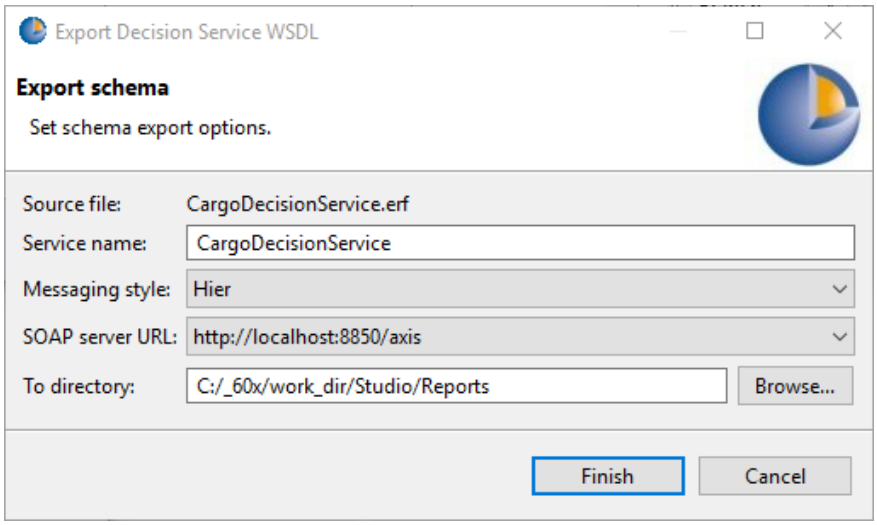
You can produce Decision Service level service contracts as WSDL or XSD.

Decision-service-level WSDL

Select the Ruleflow you want. Open in the editor. Choose the menu command **Ruleflow > Export WSDL**.



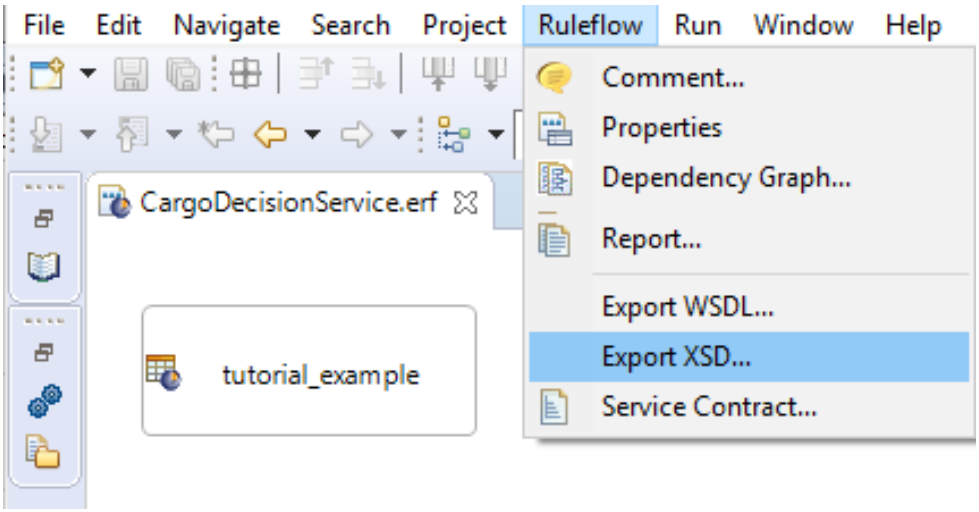
This dialog box opens:



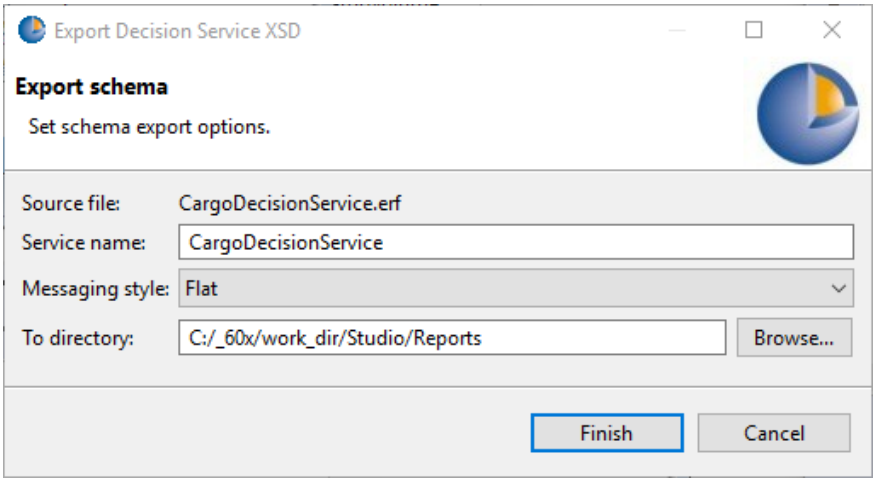
Click **Finish**. The report is generated and saved as *`RuleflowName`*`_0.wsdl` in the **To Directory** location.

Decision-service-level XSD

With the Ruleflow you want open in its editor, choose the menu command **Ruleflow > Export XSD**.



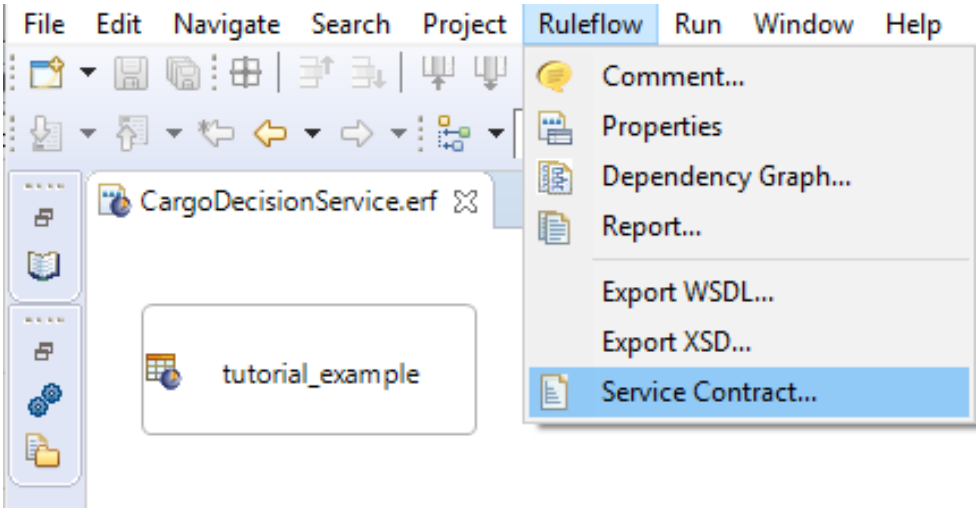
Its dialog box opens:



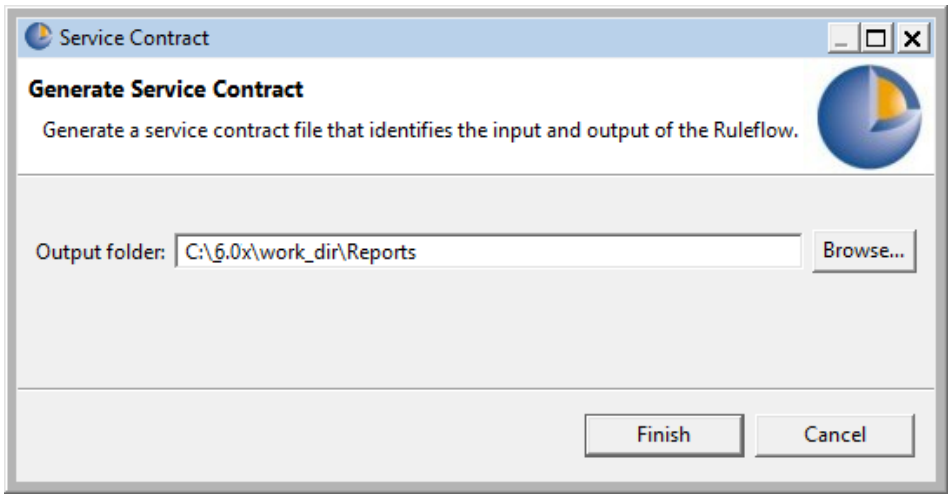
Click **Finish**. The report is generated and saved as *`RuleflowName`*`_0.xsd` in the **To Directory** location.

Decision-service-level Service Contract CSV and Excel

Open the Ruleflow you want in its editor, and then choose the menu command **Service Contract**.



Its dialog box opens:



Click **Finish**. Two reports are generated and saved: *RuleflowName*_ServiceContract.csv and *RuleflowName*_ServiceContract.xml in the **Output Folder** location. These files provide the basis for defining a JSON/REST service contract. See Generic service contract on page 76

Display service contracts in Corticon Web Console

The Corticon Web Console makes it easy to display a WSDL output for a Decision Service that produced and packaged WSDL.

Note: The default process of Decision Service compilation does not include WSDL and its report in the EDS file. The selections in the Studio export Service contracts and reports that can be passed to system integrators as a .wsdl or .xsd text and a report .html file. If you want WSDL or reports in the Decision Service package, add the following lines to the brms.properties file where Corticon will do the compilation:

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

Click on any managed or discovered Decision Service, and then click on the **/WSDL** button on the tool bar. The output you get for Cargo is illustrated:

You can open the output file in a text editor, and then save it as `Cargo_0_16.wsdl`.

6

Request and response examples

The various request and response types

Corticon accommodates many request variations from SOAP and WSDL defined XML requests to loosely defined JSON and nativeJSON. That means that various users of one decision service can be sending requests in different formats.

Note: You can set vocabulary elements to map to accept unacceptable characters by mapping them to acceptable Corticon names. For example, embedded spaces are replaced with the underscore character: `A B` maps to `A_B`.

A request must state which decision service to use. The major and minor version are optional. If you omit them, the decision service will assign the highest major and minor version that is found. If just the major version is declared, then the highest minor version of that major version is assigned.

SOAP and XML

The following examples are the output you get when you are editing a Ruletest. Click anywhere in the **Input** column, and then choose one of the **Ruletest > Export** options.

An exported SOAP header is in the form:

```
<?xml version="1.0" encoding="UTF-8"?>
  <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
         xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
  <CorticonRequest xmlns="urn:Corticon"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       decisionServiceName="InsertDecisionServiceName">
  <WorkDocuments>
```

While the corresponding XML header is simply:

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest xmlns="urn:Corticon"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     decisionServiceName="InsertDecisionServiceName">
<WorkDocuments>
```

JSON

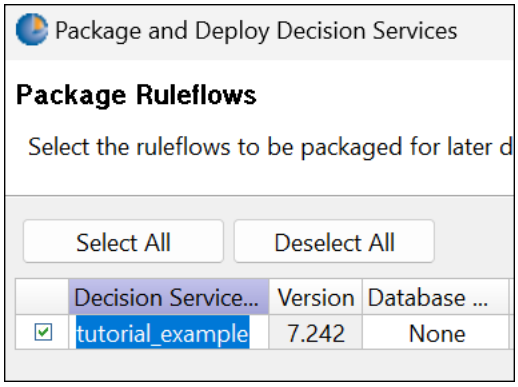
When you export a request's input as JSON it might look like this:

```
{
    "__metadataRoot": {"#locale": ""},
    "Objects": [
        {
            "volume": 10,
            "container": null,
            "weight": 1000,
            "__metadata": {
                "#type": "Cargo",
                "#id": "Cargo_id_1"
            }
        },
        {...}
        }
    ],
    "name": "InsertDecisionServiceName"
}
```

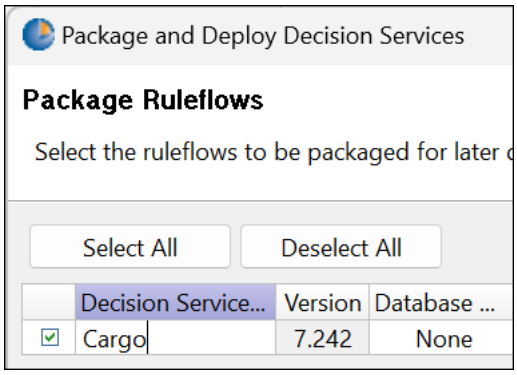
Notice that the **DecisionServiceName** says *Insert Decision Service Name*. Change it to **Cargo**.

That's not important in Ruletests, but when you try external requests it is vital. It need not be the name of the project or the Ruletest. You could name it **Freight** or **Shipping**. Just know that when you want that Decision Service, you need to use the name you assigned.

For example, using the Tutorial Ruleflow named`tutorial_example.erf`, when you start the packaging process, it looks like this:



Rename the Decision Service to `Cargo`:



When you use the Web Console, you can see how various names and versions match up for comprehensive testing.

Native JSON

Native JSON adds another level of simplicity to JSON by leaving off the metadata altogether, and expecting that the request carries the parameters for at least the decision service name. For example, where the request carries the major and minor version:

```
http://localhost/axis/corticon/execute/nativejson?
       name=Cargo&majorVersion=7&minorVersion=242
```

The request body can then be very simple:

```
{
    "volume": 10,
    "container": null,
    "weight": 1000
}
```

But if you get too simple, the algorithm that tries to compensate for the strictness in other formats can cause processing overhead. For example, if you have dozens of entities, many with an attribute `item`, you could cause delays as the algorithm discovers the right one by evaluating the other attributes. Try to keep attribute names unique, or express the entity as `__metadata` in the native JSON file.

For details, see the following topics:

- JSON and native JSON request and response messages
- SOAP and XML request and response messages

JSON and native JSON request and response messages

The following sections discuss the parameters in JSON/REST 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 JSONObjects 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 `JSONObjects` that represent Root Entities of the payload.
- `__metadataRoot` (optional) - An annotation 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 Decision Service or CcServer level properties. The following example shows the supported properties:

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

where:

- `#restrictResponseToRuleMessagesOnly`—Returns only Rule Messages that were created during this execution; no data is returned. This setting is also enabled in a Ruletest's **Execution Properties**. - Boolean
- `#locale"`—Instructs the CcServer to process this execution under a specific Locale, which might affect `Datetime` and `Decimal` values, since these values might be represented differently for a Locale. - String

The Corticon Server will translate `JSONObjects` that are inside the `JSONArray` under the name of `"Objects"`.

```
{
    "name": <string>,
    "majorVersion":  <number - optional>,
    "minorVersion":  <number - optional>,
     "Objects": [<JSONArray>
    ],
    "__metadataRoot": {
        <execution property>: “<value”,
       .
       .
    }
}
```

Metadata in requests is supported in Ruletests where it is the format of JSON output from a Ruletest when you choose **Testsheet > Data > Output > Export JSON to File**.

Native JSON

You can also use unannotated JSON payloads in a Ruletest and when calling a Decision Service. A JSON payload does not require Corticon-specific annotations to allow Corticon to map a payload to the Corticon vocabulary. That format complicated integration of Corticon from external applications by requiring passing JSON data in "Corticon’s format", not the "native format" of many applications. Eliminating the need for these annotations simplifies integrating Corticon with your applications. The annotations can still be provided but are optional.

__metadataRoot (optional)

This optional Attribute inside the main `JSONObject` can contain execution specific parameters. (Note that the initial characters are TWO underscores.) These parameters are used only for that execution, and will override a Decision Service or Server property. The following properties are supported:

- `#restrictInfoRuleMessages`
- `#restrictWarningRuleMessages`
- `#restrictViolationRuleMessages`
- `#restrictResponseToRuleMessagesOnly`
- `#locale`

- `#timezoneId`

The following example uses these properties in `__metadataRoot`:

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

Root Level Entities

All `JSONObjects` 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`.

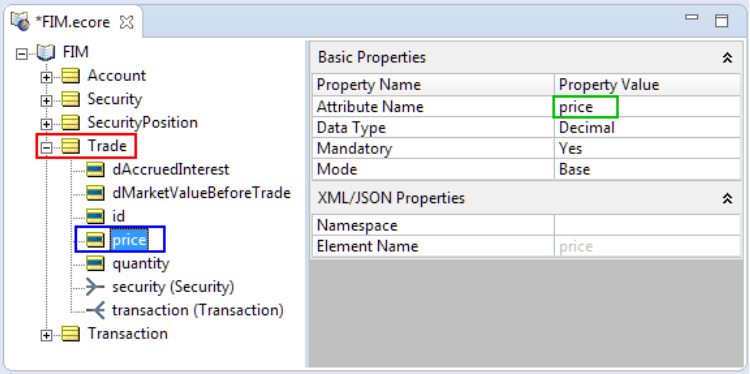
For a `Date` value, use a `String` to represent the `Date`, which will be converted into a proper `Date` object for rule processing.

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 a 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 `JSONObjects`).
- 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 `JSONObjects` 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.

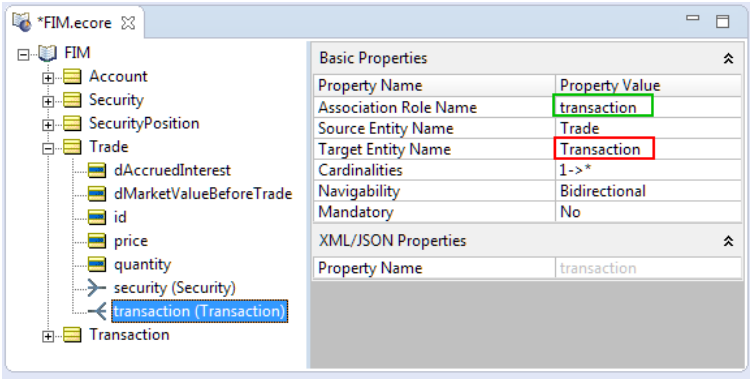
Example of Embedded Association name-value pairs:

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:

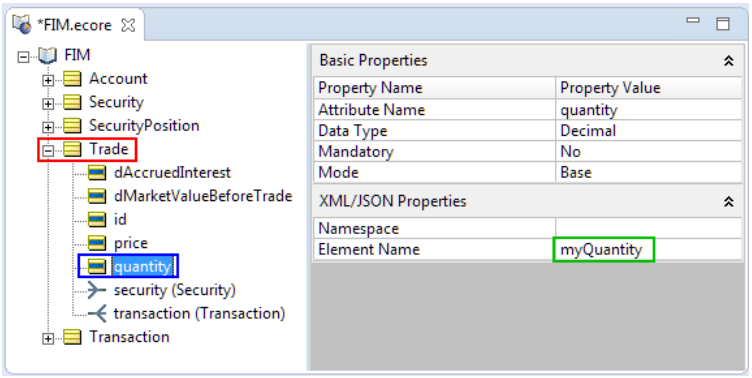
```
{
    "Objects": [{
        "dMarketValueBeforeTrade": "10216333.000000",
        "price": "950.000000",
        "transaction": [
            {
                "maxPctHiYield": "35.000000",
                "__metadata": {
                    "#id": "Transaction_id_1",
                    "#type": "Transaction"
                }
            }],
        },
        "__metadata": {
            "#id": "Trade_id_1",
            "#type": "Trade"
        }
    }],
]
```



XML Element Name overrides for Attributes and Association names

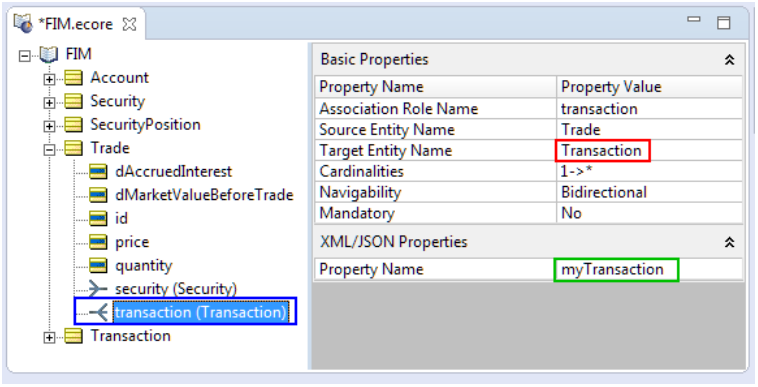
JSON Entity Attribute names are first matched against XML Name overrides, which are defined in the Vocabulary Attribute. If no XML Element Name is defined, then JSON Entity Attribute names are matched directly against the Vocabulary Attribute name.

```
{
    "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"
        }
    }],
]
```



See also *"Simplified JSON in requests" in the Web Console Guide*.

How to pass null values in a JSON request

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
```

How to control the format of associations in a JSON response

Associations in JSON responses are, by default, expressed as a `JSONArray`. You can control the format of 1-1 associations in JSON responses by setting the property:

```
com.corticon.server.execution.json.association.singleton=JSONObject
```

in the `brms.properties` on the machine where the JSON requests are processed. When this property is set, and when a Rule alters a 1-1 association through an = (equals) operator, the entity being assigned to the association will be expressed as a JSONObject, rather than an entity inside a JSONArray in the JSON response.

For example, here `Order` is represented as an array, even though there is only one order, a singleton. Note the square brackets:

```
{
 "Order":[
  {
   "OrderID":1,
   "OrderName":"XYZ"
  }
 ]
}
```

Here the same `Order` is represented as a nested object:

```
{
 "Order":{
  "OrderID":1,
  "OrderName":"XYZ"
 }
}
```

This property will not unconditionally change 1-1 Associations from a JSONArray to a JSONObject in the JSON response. Changing from a JSONArray to a JSONObject will only occur when a Rule fires to alter the contents of an Association.

Sample JSON request and response messages

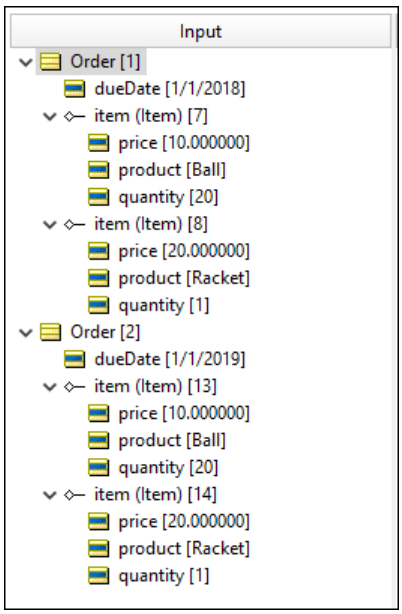
Decision service requests and the responses they generate are in two formats. This section demonstrates how two testsheets in a Corticon Studio Ruletest handle associations as detailed in *"About creating a JSON request message for a Decision Service" in this guide*:

- **Embedded hierarchical associations**—The standard way associations are described in documentation, adds the subordinate entities in an indented display.
- **Referenced flat associations**—The technique that enables shared data as described in *"How to associate one child entity with more than one parent" in the Quick Reference Guide*.

JSON Request and response with embedded hierarchical associations

The following code presents variations on the input of the `OrderProcessing` sample's `Order` Ruletest.

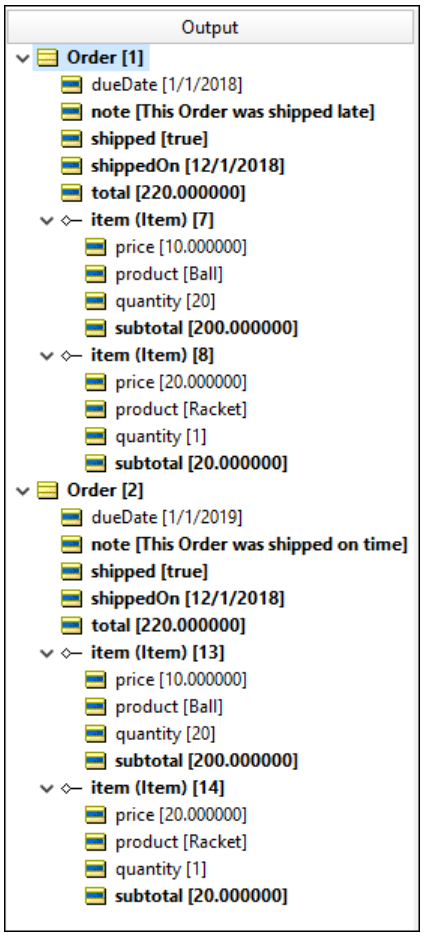
Hierarchical JSON Input—This example shows the input in its hierarchical structure using the default metadata tags. There are two orders shipping year after year with the same items, restated in each order. Here is how that looks in Studio:



The hierarchical request in JSON format with default metadata looks like this:

```
{
    "__metadataRoot": {"#locale": ""},
    "Objects": [
        {
            "item": [
                {
                    "product": "Ball",
                    "quantity": 20,
                    "price": 10,
                    "__metadata": {
                        "#type": "Item",
                        "#id": "Item_id_7"
                    }
                },
                {
                    "product": "Racket",
                    "quantity": 1,
                    "price": 20,
                    "__metadata": {
                        "#type": "Item",
                        "#id": "Item_id_8"
                    }
                }
            ],
            "dueDate": "1/1/2018",
            "__metadata": {
                "#type": "Order",
                "#id": "Order_id_1"
            }
        },
        {
            "item": [
                {
                    "product": "Ball",
                    "quantity": 20,
                    "price": 10,
                    "__metadata": {
                        "#type": "Item",
                        "#id": "Item_id_13"
                    }
                },
                {
                    "product": "Racket",
                    "quantity": 1,
                    "price": 20,
                    "__metadata": {
                        "#type": "Item",
                        "#id": "Item_id_14"
                    }
                }
            ],
            "dueDate": "1/1/2019",
            "__metadata": {
                "#type": "Order",
                "#id": "Order_id_2"
            }
        }
    ]
}
```

Hierarchical JSON Output—This example shows the output in its hierarchical structure. Here is how that looks in Studio:



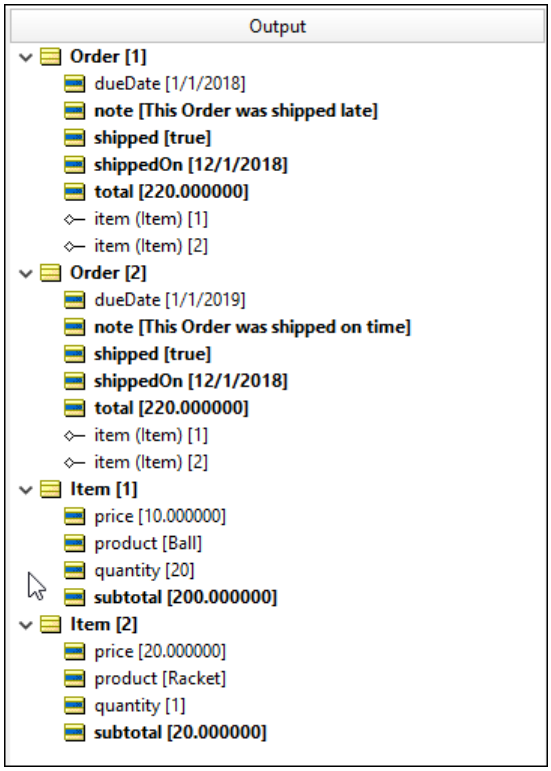
The hierarchical JSON output of only Objects (no Rule Statements) looks like this:


```
{
    "__metadataRoot": {"#locale": ""},

    "Objects": [
        {
            "note": "This Order was shipped late",
            "total": 220,
            "item": [
                {
                    "product": "Ball",
                    "quantity": 20,
                    "price": 10,
                    "subtotal": 200,
                    "__metadata": {
                        "#type": "Item",
                        "#id": "Item_id_7"
                    }
                },
                {
                    "product": "Racket",
                    "quantity": 1,
                    "price": 20,
                    "subtotal": 20,
                    "__metadata": {
                        "#type": "Item",
                        "#id": "Item_id_8"
                    }
                }
            ],
            "shipped": true,
            "dueDate": "1/1/2018",
            "__metadata": {
                "#type": "Order",
                "#id": "Order_id_1"
            },
            "shippedOn": "12/1/2018"
        },
        {
            "note": "This Order was shipped on time",
            "total": 220,
            "item": [
                {
                    "product": "Ball",
                    "quantity": 20,
                    "price": 10,
                    "subtotal": 200,
                    "__metadata": {
                        "#type": "Item",
                        "#id": "Item_id_13"
                    }
                },
                {
                    "product": "Racket",
                    "quantity": 1,
                    "price": 20,
                    "subtotal": 20,
                    "__metadata": {
                        "#type": "Item",
                        "#id": "Item_id_14"
                    }
                }
            ],
            "shipped": true,
            "dueDate": "1/1/2019",
            "__metadata": {
                "#type": "Order",
                "#id": "Order_id_2"
            },
            "shippedOn": "12/1/2018"
        }
    ]
}
```

JSON Request and response with referenced flat associations

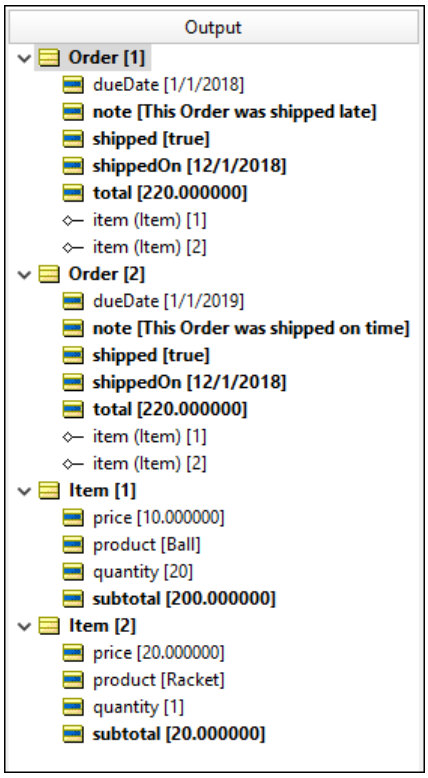
Referenced style JSON Input—This example shows the input in its flat structure using the default metadata tags. There are two orders shipping year after year with the same items, referenced in each order. See *"How to associate one child entity with more than one parent" in the Data Integration Guide*. For more information, see *"About creating a JSON request message for a Decision Service" in this Deployment Guide*. Here is how that input looks in Studio:



The referenced request in JSON format with default metadata looks like this:

```
{
    "__metadataRoot": {"#locale": ""},
    "Objects": [
        {
            "item": [
                {"__metadata": {"#ref_id": "Item_id_1"}},
                {"__metadata": {"#ref_id": "Item_id_2"}}
            ],
            "dueDate": "1/1/2018",
            "__metadata": {
                "#type": "Order",
                "#id": "Order_id_1"
            }
        },
        {
            "item": [
                {"__metadata": {"#ref_id": "Item_id_1"}},
                {"__metadata": {"#ref_id": "Item_id_2"}}
            ],
            "dueDate": "1/1/2019",
            "__metadata": {
                "#type": "Order",
                "#id": "Order_id_2"
            }
        },
        {
            "product": "Ball",
            "quantity": 20,
            "price": 10,
            "__metadata": {
                "#type": "Item",
                "#id": "Item_id_1"
            }
        },
        {
            "product": "Racket",
            "quantity": 1,
            "price": 20,
            "__metadata": {
                "#type": "Item",
                "#id": "Item_id_2"
            }
        }
    ]
}
```

JSON Output of referenced shared data —This example shows the output in its flat structure. Here is how that looks in Studio:



The JSON output of a shared data request (with only Objects) looks like this:

```
{
    "__metadataRoot": {"#locale": ""},

    "Objects": [
        {
            "note": "This Order was shipped late",
            "total": 220,
            "item": [
                {"__metadata": {"#ref_id": "Item_id_1"}},
                {"__metadata": {"#ref_id": "Item_id_2"}}
            ],
            "shipped": true,
            "dueDate": "1/1/2018",
            "__metadata": {
                "#type": "Order",
                "#id": "Order_id_1"
            },
            "shippedOn": "12/1/2018"
        },
        {
            "note": "This Order was shipped on time",
            "total": 220,
            "item": [
                {"__metadata": {"#ref_id": "Item_id_1"}},
                {"__metadata": {"#ref_id": "Item_id_2"}}
            ],
            "shipped": true,
            "dueDate": "1/1/2019",
            "__metadata": {
                "#type": "Order",
                "#id": "Order_id_2"
            },
            "shippedOn": "12/1/2018"
        },
        {
            "product": "Ball",
            "quantity": 20,
            "price": 10,
            "subtotal": 200,
            "__metadata": {
                "#type": "Item",
                "#id": "Item_id_1"
            }
        },
        {
            "product": "Racket",
            "quantity": 1,
            "price": 20,
            "subtotal": 20,
            "__metadata": {
                "#type": "Item",
                "#id": "Item_id_2"
            }
        }
    ]
}
```

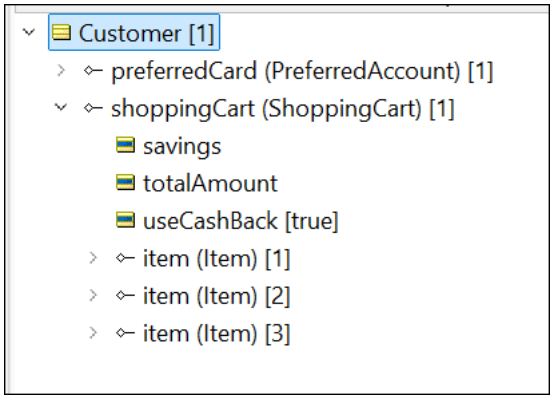
Testing a JSON request

Formats of requests

You can specify JSON requests in either of two formats:

- **Detailed with `__metadata` annotations**—The default format allows detailed, unambiguous requests. This format is especially useful when associations are involved, or when the entities have attributes that are used in many entities.
- **Native with no annotations**—Many applications produce requests in the minimal format, presenting compact and readable files for requests. The format requires that Corticon parse the incoming data to identify the root entities and the attributes of the associated entities.

Either format is accepted as tester input. The following example uses the Advanced Tutorial's Ruleflow, as shown:



Sample request with full `__metadata` annotations

```
{
    "__metadataRoot": {"#locale": ""},
    "Objects": [{
        "preferredCard": {
            "cumulativeCashBack": 9.24,
            "__metadata": {
                "#type": "PreferredAccount",
                "#id": "PreferredAccount_id_1"
            },
            "cardNumber": "12"
        },
        "ShoppingCart": {
            "totalAmount": null,
            "Item": [
                {
                    "price": 55,
                    "name": "Filet Mignon",
                    "__metadata": {
                        "#type": "Item",
                        "#id": "Item_id_1"
                    },
                    "department": null,
                    "barCode": "39-280-12345"
                },
                {
                    "price": 14.99,
                    "name": "Beach Towel",
                    "__metadata": {
                        "#type": "Item",
                        "#id": "Item_id_2"
                    },
                    "department": null,
                    "barCode": "32-300-23456"
                },
                {
                    "price": 12.5,
                    "name": "Ginger Ale Case",
                    "__metadata": {
                        "#type": "Item",
                        "#id": "Item_id_3"
                    },
                    "department": null,
                    "barCode": "32-285-34567"
                }
            ],
            "useCashBack": true,
            "savings": null,
            "__metadata": {
                "#type": "ShoppingCart",
                "#id": "ShoppingCart_id_1"
            }
        },
        "__metadata": {
            "#type": "Customer",
            "#id": "Customer_id_1"
        },
        "Name": ""
    }]
}
```

SOAP and XML request and response messages

Notice that the **decisionServiceName** says *Insert Decision service Name*.

```
decisionServiceName="InsertDecisionServiceName"
```

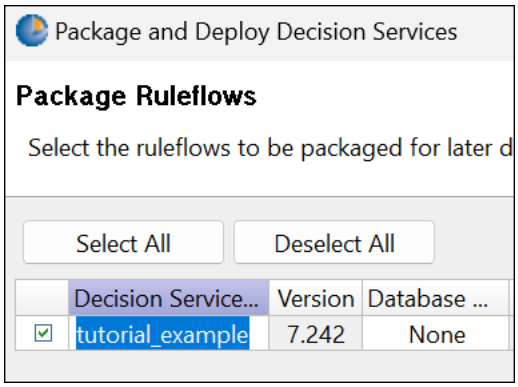
Change it to Cargo:

```
decisionServiceName="Cargo">
```

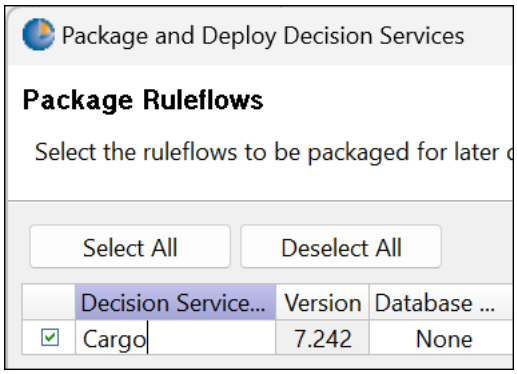
That will be the decision service when it is deployed.

Packaging the Decision Service

Open the Tutorial Ruleflow named`tutorial_example.erf`, and then start the packaging process by choosing the **Project** menu item **Package and Deploy Decision Services**:



Rename it to `Cargo`:



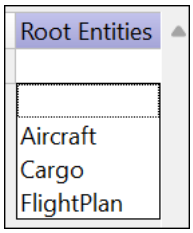
The names do not need to match in the Ruletest, but the are integral part of the Decision Service you deploy.

The name is not required to be the project name or the source file name. You could name the Decision Service `Freight`, and then package it as `Freight`. Two different Decision Services.

When you use the Web Console, you can see how various names and versions match up for comprehensive testing.

Root Entities

Another aspect of creating decision services is easing the burden in the algorithm that determines the root of the request data.



This has two effects:

1. If the Vocabulary has a large number of entities, and especially if there are large number of attributes that are common, the algorithm that determines that right match can be more efficient
2. When the request is in native JSON, the absence of entity IDs and even metadata can make the match arbitrary and possibly cause errors.

Sample XML CorticonRequest content

A sample `CorticonRequest` payload is produced in the Corticon Studio when you choose Export XML to Fileshown 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>
```

How to pass null values in an XML request

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

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

XML Payloads with null values created by Rules

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

Assume that the incoming payload is...

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

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

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

```
Person
  Name : Jack
```

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>
```

7

Decision Service versioning and effective dating

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

This section 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 versions and effective dates" in the Rule Modeling Guide.*

For details, see the following topics:

- How to deploy Decision Services with identical Decision Service names
- How to invoke a Decision Service by version number
- How to invoke a Decision Service by date
- Summary of major version and effective timestamp behavior

How to deploy Decision Services with identical Decision Service names

Typically, each Decision Service deployed to a Corticon Server has a unique Decision Service name that lets the rules engine handle the request when applications and clients invoke a Decision Service by its name.

However, the Decision Service Versioning and Effective Dating feature makes an exception to this rule. Decision Services with identical Decision Service Names can be deployed on the same Corticon Server provided that each has a different Major version number, or that each has the same Major yet a different Minor version number.

For example, you could deploy `Cargo_v0_16.eds`,`Cargo_v1.1.eds`,`Cargo_v1.2.eds`.

Note: An alternate approach is to rename the Decision Service when you compile it so that you have, say, `Cargo_v0_16.eds`and `Freight_v0_16.eds`.

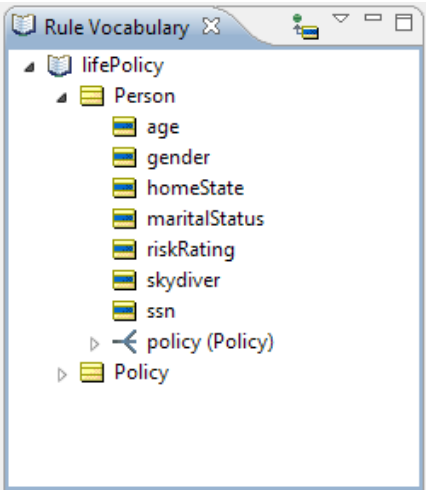
How to invoke a Decision Service by version number

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

How to create samples of versioned Ruleflows

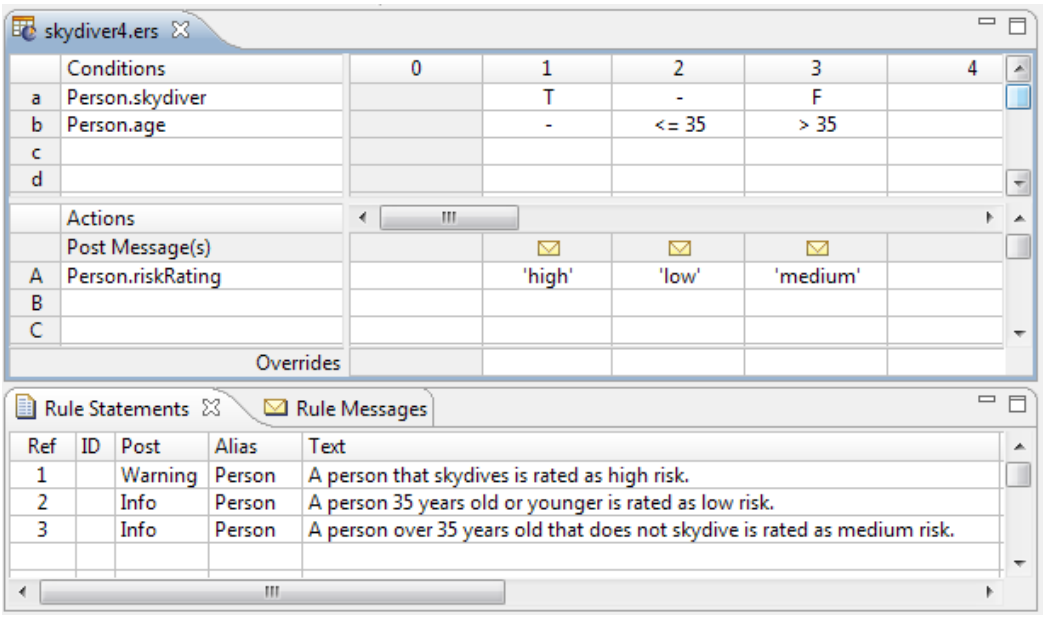
The Ruleflows in this section are based on Rulesheet variations of a single rule. Notice that the only difference between the three Rulesheets is the threshold for the age-dependent rules (columns 2 and 3 in each Rulesheet). The age threshold is 35, 45, and 55 for Version 1, 2 and 3, respectively. This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

Figure 31: Sample Vocabulary for demonstrating versioning



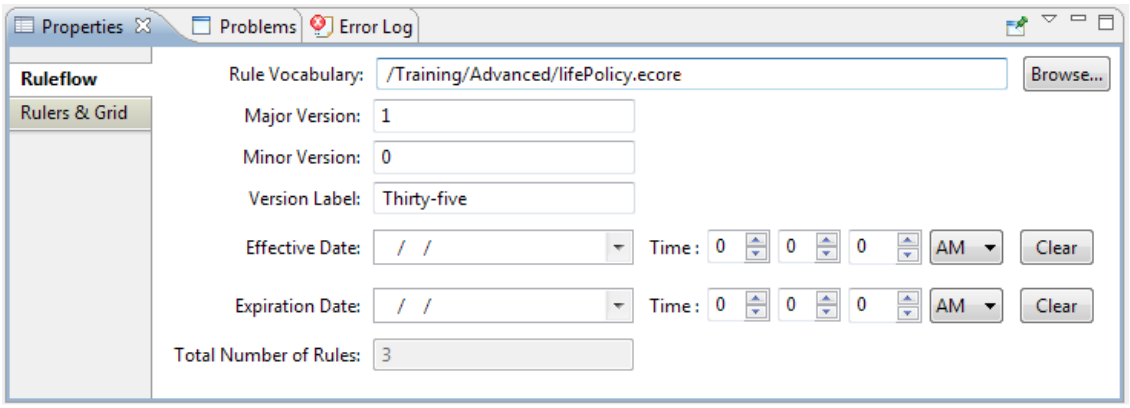
There is more than one Ruleflow with the same name and differing versions, so first a **File > New Folder**placed a `Version1` folder in the project. Then Rulesheet was created for defining the policy risk rating that considers age 35 as a decision point, as shown:

Figure 32: Rulesheet skydiver4.ers in folder Version1



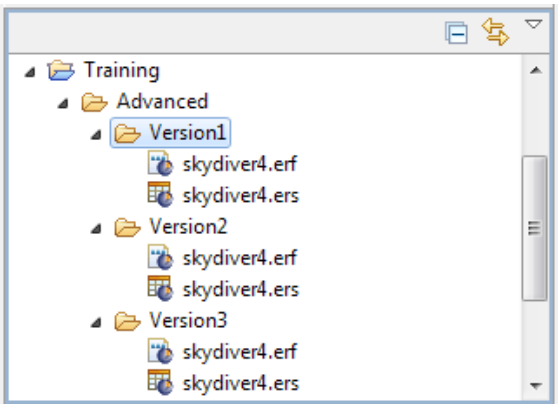
A new Ruleflow was created that added the Version1 `skydiver4.ers` Rulesheet to it. Then, the Major version was set to `1` and the Minor version to `0`. The label `Thirty-five` was entered to express the version in natural language.

Figure 33: Ruleflow in folder Version1 and set as Version 1.0



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

Figure 34: 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 you can deploy both versioned and effective-dated Decision Services at the same time.

The Rulesheets and Ruleflows in the copied folders were edited as shown, first for Version2:

Figure 35: Rulesheet skydiver4.ers in folder Version2

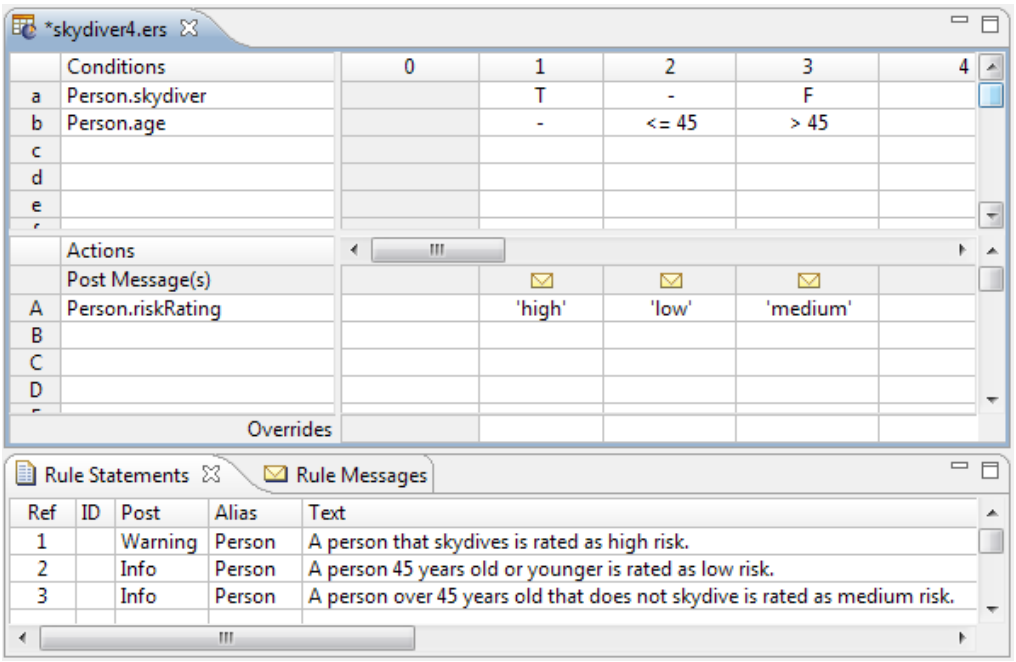
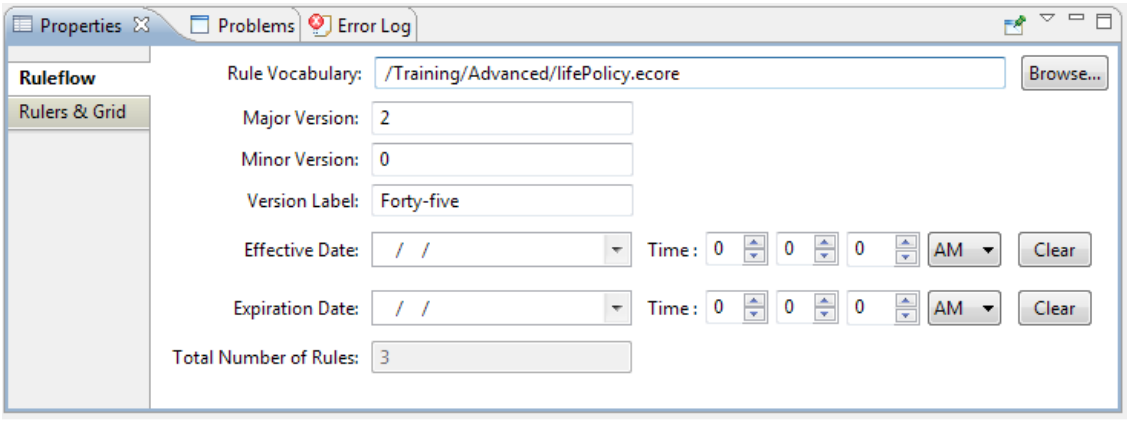


Figure 36: Ruleflow in folder Version2



And then for Version 3:

Figure 37: Rulesheet skydiver4.ers in folder Version3

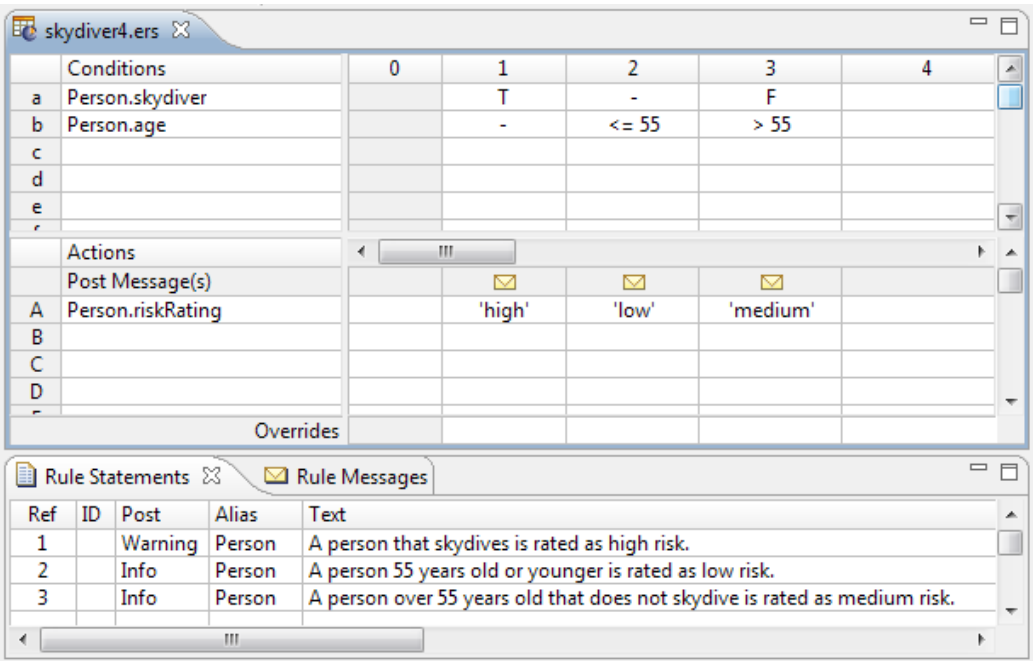
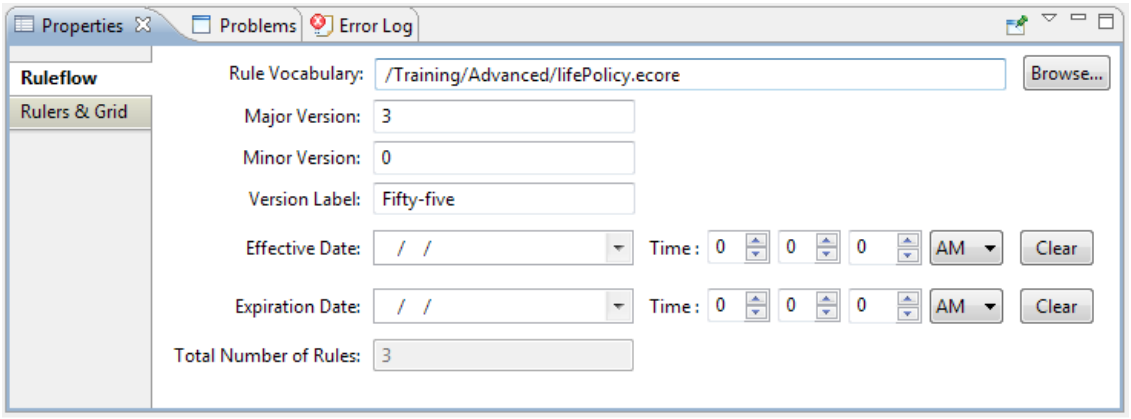


Figure 38: Ruleflow in folder Version3



How to specify 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 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.

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.

Execute the request - Use your preferred SOAP API to execute the request. The Web Console provides a straightforward way to test executions. 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 was defined as version `1.0` of the Decision Service. This should be no surprise—you specifically requested version `1.0` in the request message. Corticon Server has honored the 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.

```
<?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 the request to use version `2.0` of the Decision Service. The age threshold of `45` is the hint that version `2.0` was executed.

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
```

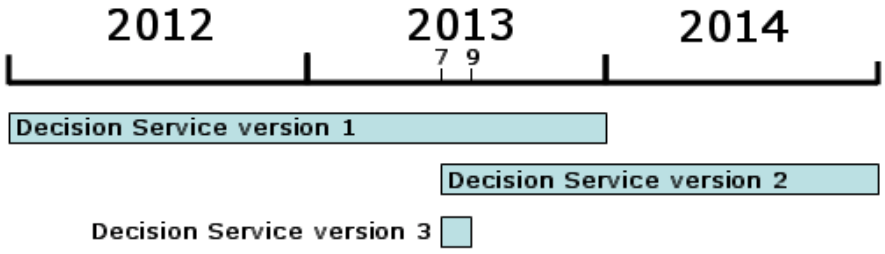
When no Version Number or EffectiveTimestamp is specified, the Server executes against v2.1 (if its Effective/Expiration range is valid). However, when Major Version 1 is passed in without an EffectiveTimestamp specified, the Server executes against v1.2 (if its Effective/Expiration range is valid).

How to invoke a Decision Service by date

When multiple Major versions of a Decision Service also contain different Effective and Expiration Dates, you can also instruct Corticon Server to execute a 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 39: DS Effective and Expiration Date Timeline



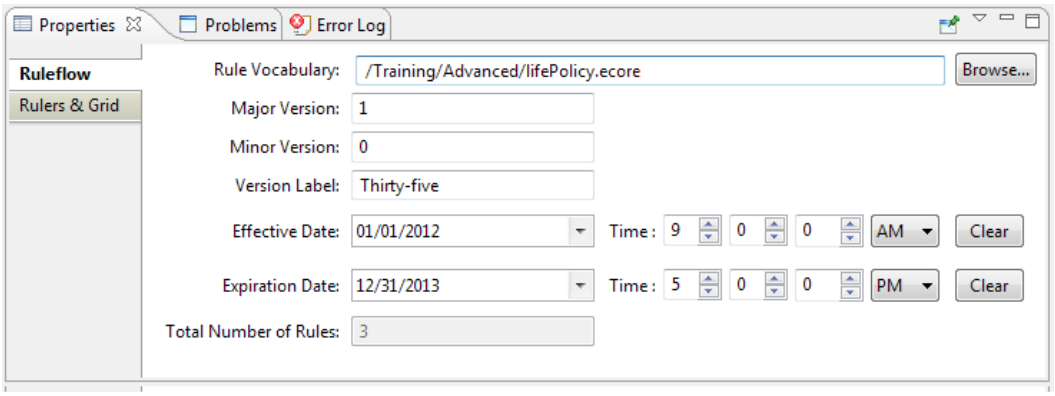
As illustrated, the three deployed Decision Services have Effective and Expiration dates that overlap in several date ranges: Version 1 and Version 2 overlap from July 1, 2013 through December 31, 2013. And Version 3 overlaps with both 1 and 2 in July-August 2013. To understand how Corticon Server resolves these overlaps, let's 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.

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

Figure 40: Ruleflow in folder Version1 with dateTime set



Now proceed to edit the other two Ruleflows as shown:

Figure 41: Ruleflow in folder Version2 with dateTime set

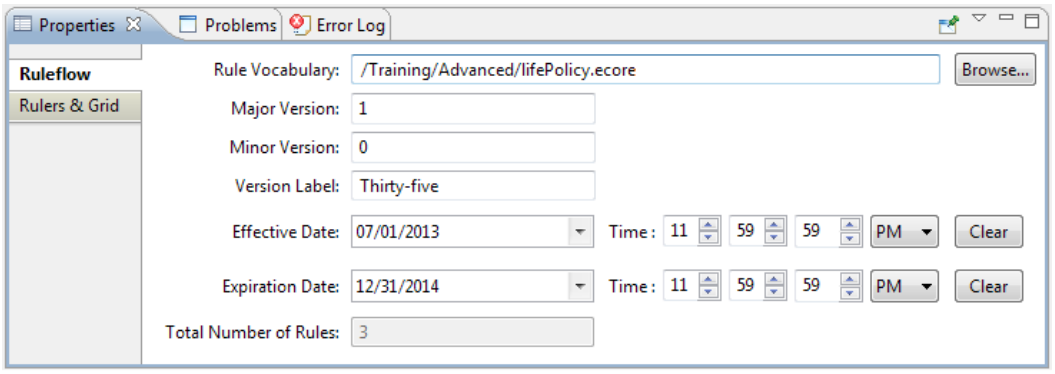
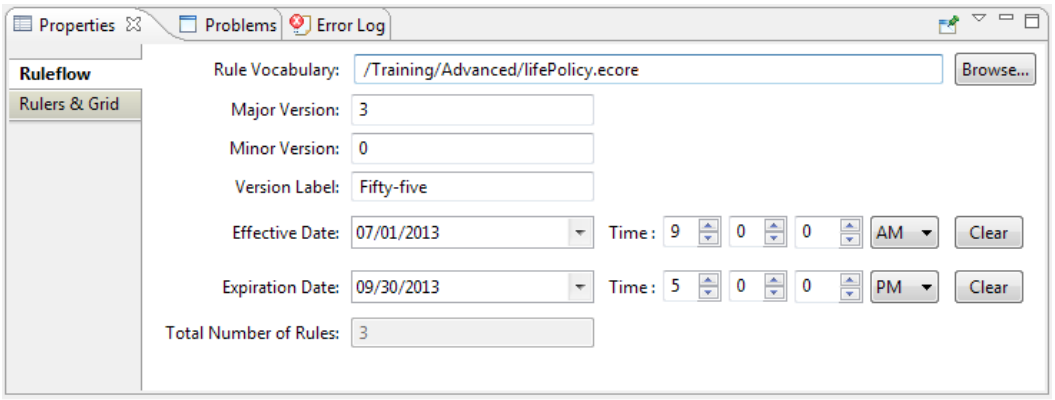


Figure 42: Ruleflow in folder Version3 with dateTime set



How to specify 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 from Corticon Studio. Refer to the topic How to integrate Corticon Decision Services on page 73 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>
```

Execute the request - Use your preferred SOAP API to execute the request.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.

How to specify 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 you saw in the overlap example).

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

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

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

Summary of major version and effective timestamp behavior

Request Specifies Major Version?	Request Specifies Minor Version?	Request Specifies Timestamp?	Server Behavior
No	No	No	Execute the highest Major.Minor version Production Decision Service that is in effect based on the invocation timestamp

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

8

Enable Server handling of locales languages and time zones

By default, Corticon Server will parse input data and return output using the locale of the system where it is running. When deploying Decision Services that will be called by users or services running in different locales, you need to address issues with locale-dependent data formats and localized messages. When calling a Corticon Server in a different locale, you need to pass with the input payload the `locale` property. This property instructs Corticon how to parse locale dependent data such as dates and numbers in the input payload and how to format data returned. In addition to formatting of data returned, the `locale` property will cause rule messages to be returned in the specified locale if translations for those messages are defined in the Decision Service. When calling a Corticon Server in a different time zone, set the `timezone` property in the input payload so that the server can transform the payload's time calculations to the time zone of the server, run the rules, and return the output formatted for the submitter's specified time zone.

Note: Locale can be set in Studio for running Ruletest Testsheets in Studio and against a server. See *"How to set the locale for a Testsheet" in the Quick Reference Guide.* Also see *"How to localize Corticon Studio" in the Quick Reference Guide*.

For details, see the following topics:

- How to handle requests and replies across locales
- Examples of cross-locale processing
- Example of cross-locale literal dates
- Example of requests that cross time zones

How to handle 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 a Java standard identifier, such as en-US for **English-United States**.

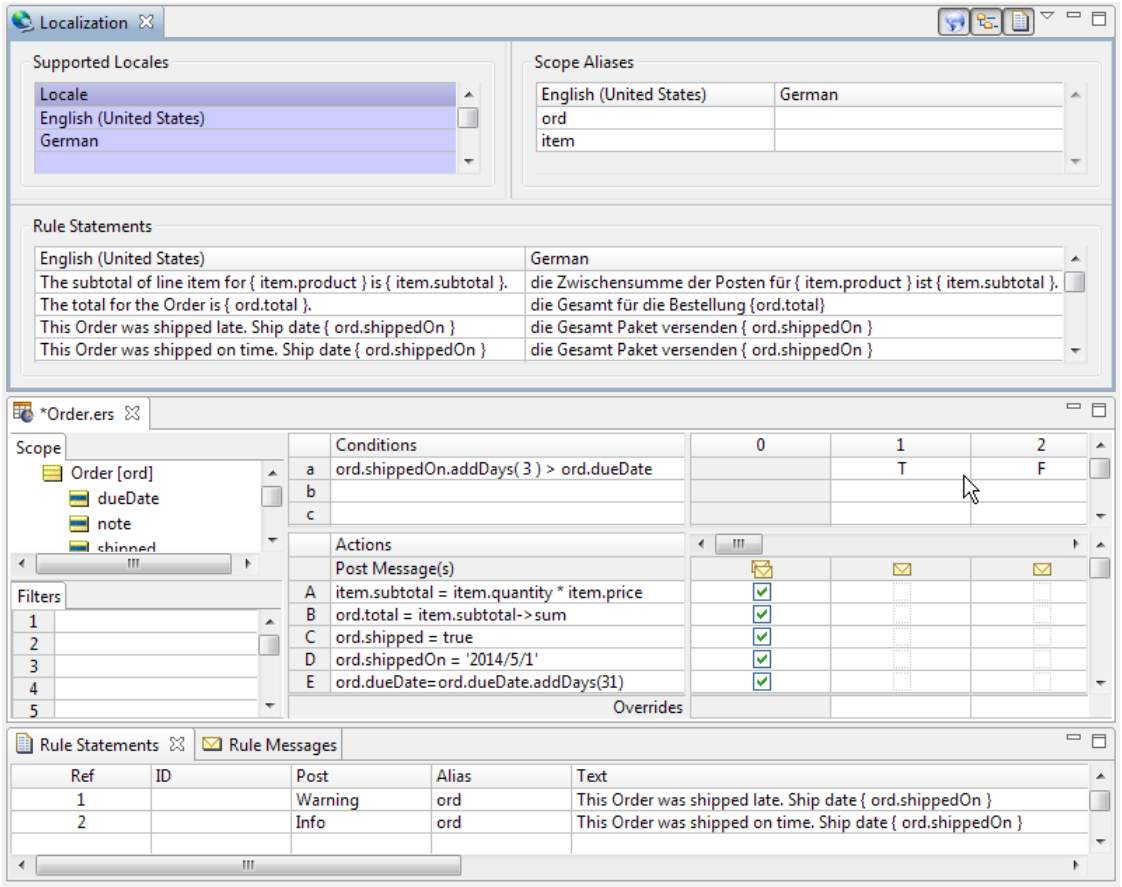
When the message's locale is specified, it is used at rule execution time regardless of the Server's default locale. If the Rulesheet has a matching locale, those rule statement messages are used. When rule processing is complete, the output response maps the results to the formats of the requestor's 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 the `locale` 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

If you have Decision Service that is called from English and German locales you would want to define translations for your rule statements as shown here:



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.

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 if 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>
    ...
  </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: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 Paket versenden 05/01/14 12:00:00 AM</ns1:text>
          <ns1:entityReference href="#Order_id_1" />
        </ns1:Message>
      </ns1:Messages>
    </ns1:CorticonResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

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

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

```
      </ns1:ExecutionProperties>
     ...
```

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

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

Example of cross-locale literal dates

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

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

Note: Attributes in a response that were not specified in its request message will have the standard `date` and `dateTime` formats for the locale.

To use literal names for input dates echoed in the response:

1. Stop the server.
2. Locate and edit the `brms.properties` text file.
3. Add (or update) the line `com.corticon.ccserver.ensureComplianceWithServiceContract.lenientDateTimeFormat=true`
4. Save the edited file.
5. Start the server.

The following request from `de-DE` is like the one in the previous topic except that is submits literal month names, in this case `Sep` and `Okt`:

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 decisionServiceName="Order_localeAware">
  <ExecutionProperties>
    <ExecutionProperty name="PROPERTY_EXECUTION_LOCALE" value="de-DE" />
  </ExecutionProperties>
  <WorkDocuments>
    <Order id="Order_id_1">
```

```
      <dueDate>Sep 25, 2014</dueDate>
      <total xsi:nil="1" />
      <myItems id="Item_id_1">
        <price>10,250000</price>
        <product>Ball</product>
        <quantity>20</quantity>
      </myItems>
    </Order>
    <Order id="Order_id_2">
      <dueDate>Okt 9, 2014</dueDate>
      <myItems id="Item_id_4">
        <price>0,050000</price>
        <product>Pencil</product>
        <quantity>100</quantity>
      </myItems>
    </Order>
  </WorkDocuments>
</CorticonRequest>
```

The response handles not only the decimal delimiter and German rule statements, it also adds a month to the dates so it calculates and then replies with `Okt` and `Nov`:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:CorticonResponse xmlns:ns1="urn:Corticon" xmlns="urn:Corticon"
decisionServiceName="Order_localeAware">
      <ns1:ExecutionProperties>
        <ns1:ExecutionProperty name="PROPERTY_EXECUTION_LOCALE"
                               value="de-DE" />
      </ns1:ExecutionProperties>
      <ns1:WorkDocuments>
        <ns1:Order id="Order_id_1">
          <ns1:dueDate>Okt 26, 2014</ns1:dueDate>
          <ns1:shipped>true</ns1:shipped>
          <ns1:shippedOn>05/01/14 12:00:00 AM</ns1:shippedOn>
          <ns1:total>205,000000</ns1:total>
          <ns1:myItems id="Item_id_1">
            <ns1:price>10,250000</ns1:price>
            <ns1:product>Ball</ns1:product>
            <ns1:quantity>20</ns1:quantity>
            <ns1:subtotal>205,000000</ns1:subtotal>
          </ns1:myItems>
        </ns1:Order>
       ...
      </ns1: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 Gesamt Paket versenden 05/01/14 12:00:00 AM</ns1:text>
          <ns1:entityReference href="#Order_id_2" />
        </ns1:Message>
      </ns1:Messages>
    </ns1:CorticonResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Similarly, the following `fr-FR` request is similar to the one in the previous topic except that is submits literal month names, in this case `avril` and `juillet`:

Note: Case is important.

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 decisionServiceName="Order_localeAware">
  <ExecutionProperties>
    <ExecutionProperty name="PROPERTY_EXECUTION_LOCALE"
                       value="fr-FR" />
  </ExecutionProperties>
  <WorkDocuments>
    <Order id="Order_id_1">
      <dueDate>avril 25, 2014</dueDate>
      <total xsi:nil="1" />
      <myItems id="Item_id_1">
        <price>10,250000</price>
        <product>Ball</product>
        <quantity>20</quantity>
      </myItems>
    </Order>
   ...
  </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: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 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>
  ...
  </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:WorkDocuments>
    <ns1:Messages version="1.10">
      <ns1:Message>
        <ns1:severity>Info</ns1:severity>
        <ns1:text>The subtotal of line item for Pencil is 5.000000.</ns1:text>
        <ns1:entityReference href="#Item_id_4" />
      </ns1:Message>
      <ns1:Message>
        <ns1:severity>Info</ns1:severity>
        <ns1:text>The subtotal of line item for Ball is 205.000000.</ns1:text>
        <ns1:entityReference href="#Item_id_1" />
      </ns1:Message>
      ...

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

Example of requests that cross time zones

To call a Decision Service in another time zone, you need to specify the `timezone` property so dates and times are correctly handled.

Note: Time zone 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 time zone 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 time zone 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>
```

9

Sample client applications

Corticon Server installations include sample applications demonstrating how to 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 get started calling Decision Services. The source samples are heavily commented. The samples are in a Corticon Server installation's `[CORTICON_WORK_DIR]\Samples\Clients` directory.

To get started calling Corticon as a Web Service, see one of these samples:

- C-sharp
- - `C-sharp\RESTClient`
  - `C-sharp\SOAPClient`
- Java
  - `Java\RESTClient`
  - `Java\SOAPClient`
- JavaScript
  - `JavaScript\RESTClient`
- Python
  - `Python\RESTClient`
  - `In Process\C-sharp`
  - `In Process\Java`
- For more examples of the Corticon API in use in:
  - `REST\CcServerRestTest.java`
  - `SOAP\CcServerApiTest.java`

To get started calling Corticon in-process, see the documentation guides:

- *Deploy Corticon Server in an Application*
- *Tutorial - Deploying a Progress Corticon Decision Service in Process for Java*.