Progress®
Corticon®

Deploy Corticon Server in an Application

Copyright

Last updated: Corticon 7.3

Updated: 2025/11/19

Table of Contents

1

When to deploy Corticon Server in your application

An alternative to deploying Corticon Server as a web service is to deploy Corticon Server in your application. When deployed in an application, the Corticon Server is embedded in your own application. This is sometimes referred to as deploying an *in-process* server.

This option requires that you write application code to initialize a Corticon Server, load Decision Services, and invoke them with the payload data from your application. There are two reason why you may decide to embed a Corticon Server in your application.

- **For performance** - Eliminates the network call to Corticon as a web-service. This might be done for applications with critical low-latency requirements.
- **For integration** - Allows you to create a custom wrapper for invoking your decision services. Corticon provides REST and SOAP wrappers for deploying the Corticon Server but you may have other needs such as deploying Corticon as a message client servicing requests off a message queue.

Corticon supports embedded deployment for Java and .NET applications. The Corticon Server install includes sample applications demonstrating the essentials of deploying Corticon in-process. The Corticon Server Java Doc provides full details of the Server API.

When embedding Corticon Server in your application, you can invoke Decision Services by passing payload data as JSON or XML. JSON is the fastest performing way of invoking Decision Services and recommended for performance critical applications.

2

Required application code changes

To embed Corticon Server in a Java or .NET application the application code you need to write is largely the same. The minimum requirements are:

1. Instantiate a Corticon Server.

   ```
   ICcServer corticonServer = CcServerFactory.getCcServer();
   ```

2. Load your Decision Services.

   ```
   corticonServer.addDecisionService("YourDSName","YourDS.eds", properties);
   ```

3. Invoke the Decision Service with your application data.

   ```
   corticonServer.execute (yourPayload);
   ```

The Corticon Server API provides many options to:

- Instantiate and configure a Corticon Server
- Load and configure Decision Services
- Invoke Decision Services
- Retrieve performance metrics
- and more.

For a full description of the Corticon Server API, see the Server API Javadoc.

3

Required Corticon Server configurations

Corticon Server provides many configuration options. These are described in detail in the Corticon Server guide. It's recommended that you read this guide to better understand the operation of Corticon Server. When using an embedded Corticon Server the following must be configured.

Environment Settings

`CORTICON_HOME` and `CORTICON_WORK` need to be set for both JAVA and .NET. Setting the `CORTICON_WORK` when using .NET in-process is required. For a specific example, see the .NET in-process sample's `App.config` at `<CORTICON_WORK>\Samples\Clients\In Process\C-sharp`.

Server Properties

At startup, Corticon Server will read Server configuration properties from the `brms.properties` in the Corticon work directory identified by `CORTICON_WORK_DIR`. After you have the Server instance, you can set additional properties on it by calling `setCcPropertyValue` on the instance.

Server Sandbox

When Corticon Server starts up, it checks for the existence of a sandbox directory. The Sandbox is the directory used by Corticon Server to manage its state and deployed Decision services. The location of the Sandbox is controlled by `com.corticon.ccserver.sandboxDir` settings in your `brms.properties` file.

Note: If the location specified by `com.corticon.ccserver.sandboxDir` cannot be found or is not available, the Sandbox location defaults to the current working directory as it is typically the location that initiated the call.

Server Instances

Only one instance of the Corticon Server is allowed per application. The first call to `CcServerFactory.getCcServer` instantiates the Corticon Server and returns it. All subsequent calls will return the same instance.

4

Java application requirements

To deploy Corticon in your Java application you need to include the following Corticon Server jar files in your application classpath:

- `ant_launcher.jar`
- `CcConfig.jar`
- `CcDrivers.jar` (Could be skipped if there are no database connections in the project.)
- `CcExtensions.jar`
- `CcI18nBundles.jar`
- `CcServer.jar`
- `CcThirdPartyJars.jar`

These JAR files are in:

```
<CORTICON_HOME>/Server/lib
```

In addition, your application classpath needs to include:

- `CcLicense.jar` : The license you receive from Progress for Corticon Server
- Any jar files for custom extended operators or service callouts used by your Decision Services

Sample Java Application

Corticon Server includes a sample Java application and Eclipse project demonstrating how to deploy Corticon Server in a Java application. This sample application provides a great starting point for you to learn the basics about deploying in a Java application. The sample located in:

```
<CORTICON_WORK>/Samples/Clients/In Process/Java
```

Note: This project was created for the installed Eclipse, and can be opened in Corticon Studio. If opening in an Eclipse install other than Corticon Studio you will need to define the Eclipse variable `CORTICON_HOME` for the Corticon JAR file references.

For details, see the following topics:

- Java object mapping
- Specifying version in a Java API call
- Specifying effective timestamp in a Java API call
- Invoking Corticon Server with a Java call

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.

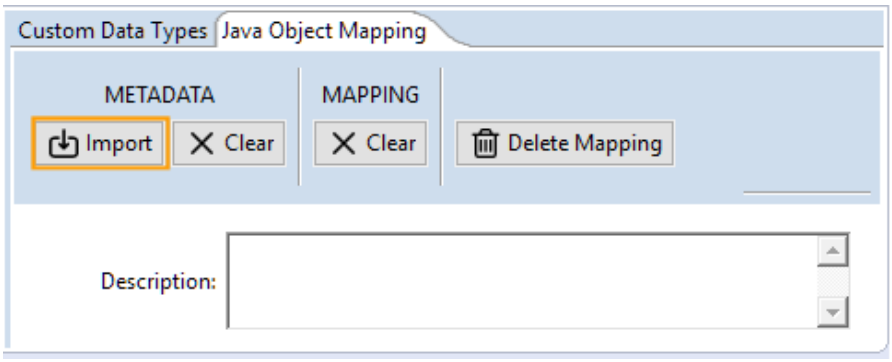
To import package metadata:

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.

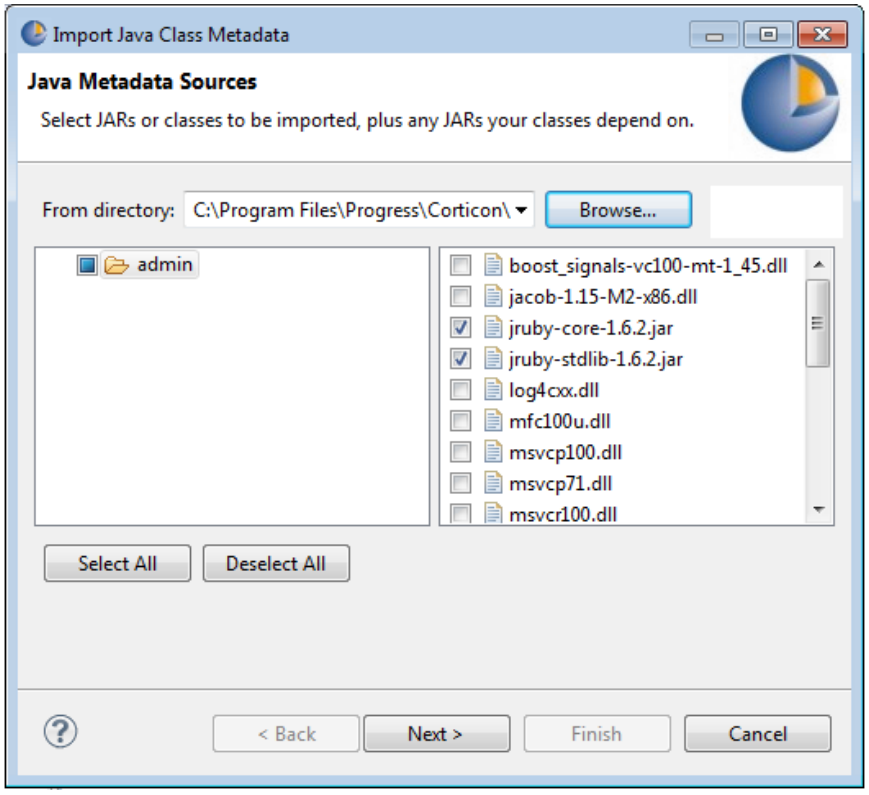
3. On the **Java Object Mapping** tab, click **Import**

Figure 1: Importing Java Class Metadata for Mapping



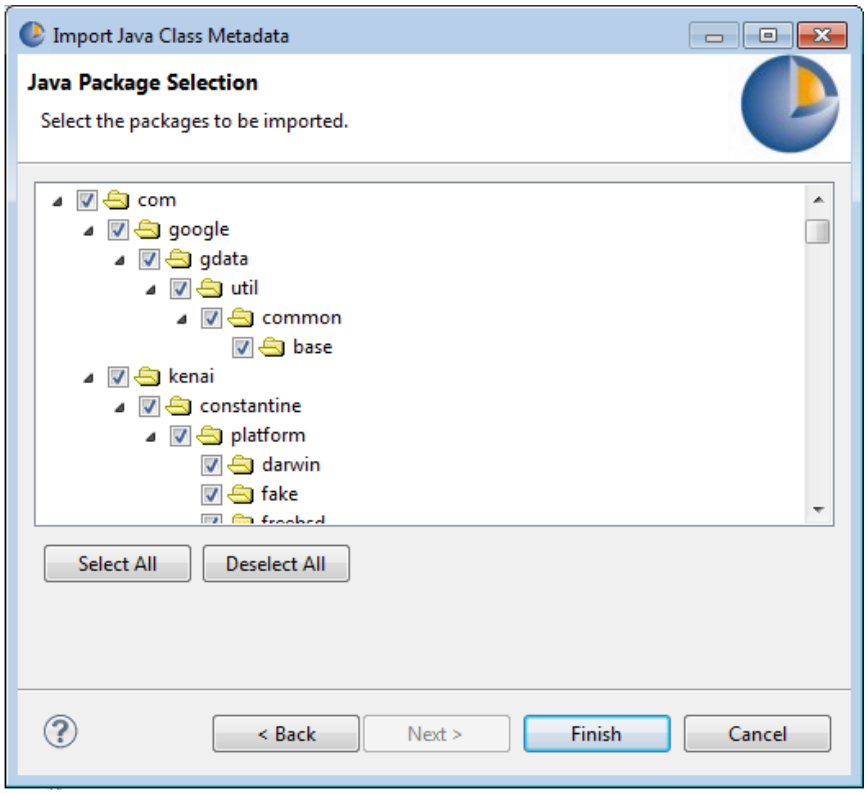
4. Use the **Browse** button to select the location of your Java Business Objects. They should be compiled `class` files or Java archives (`.jar` files).

Figure 2: Browsing to your Java Class files



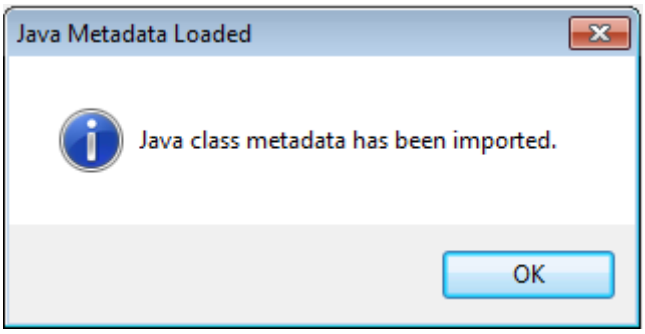
5. Select the package containing the Java business objects as shown:

Figure 3: Importing Java Class Metadata for Mapping



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

Figure 4: 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 5: 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() {}
```

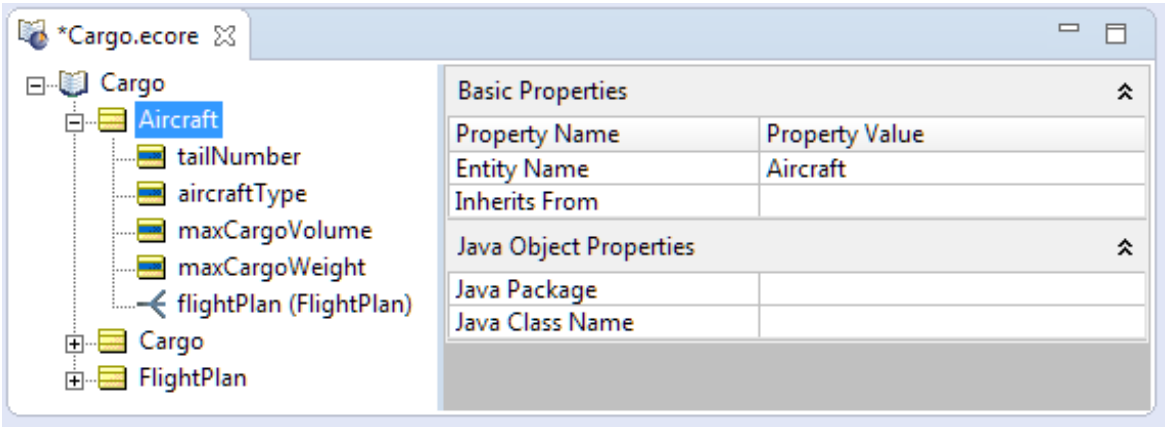
We can see in line 6 of this figure that this class is not actually named `Aircraft` – it is named `MyAircraft`. The automatic mapper attempts to locate a class by the same name as each entity. In the case of `Aircraft`, it looks for a class named `Aircraft`. Not finding one, it leaves the field empty,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

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

Table 1: Java Object Mapping Entity Properties

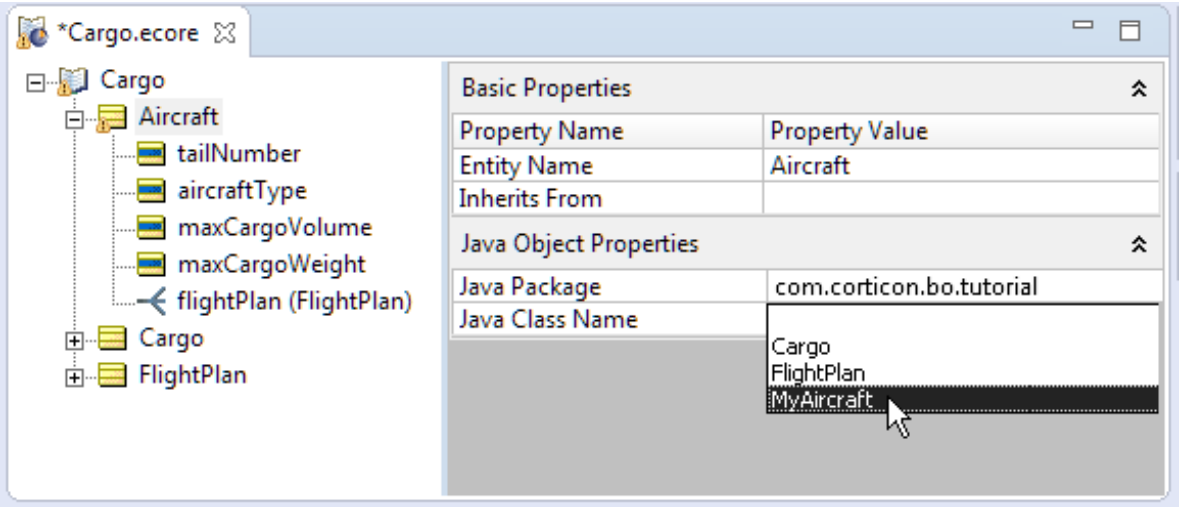
Property	Value
Java Package	Specifies the package to be used for Java class metadata mapping.
Java Class Name	Maps the entity to the specified class when no class exists with the entity name.

Figure 6: 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 7: 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 8: 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
```

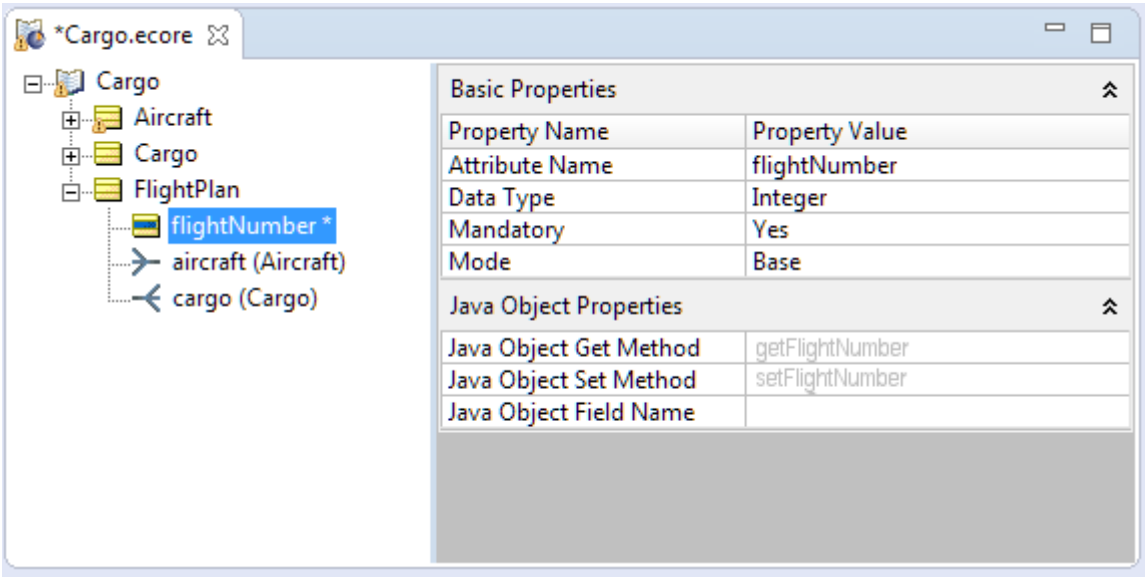
When Java Object mapping has been added to the Vocabulary, its Attribute properties are displayed.

Table 2: 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.

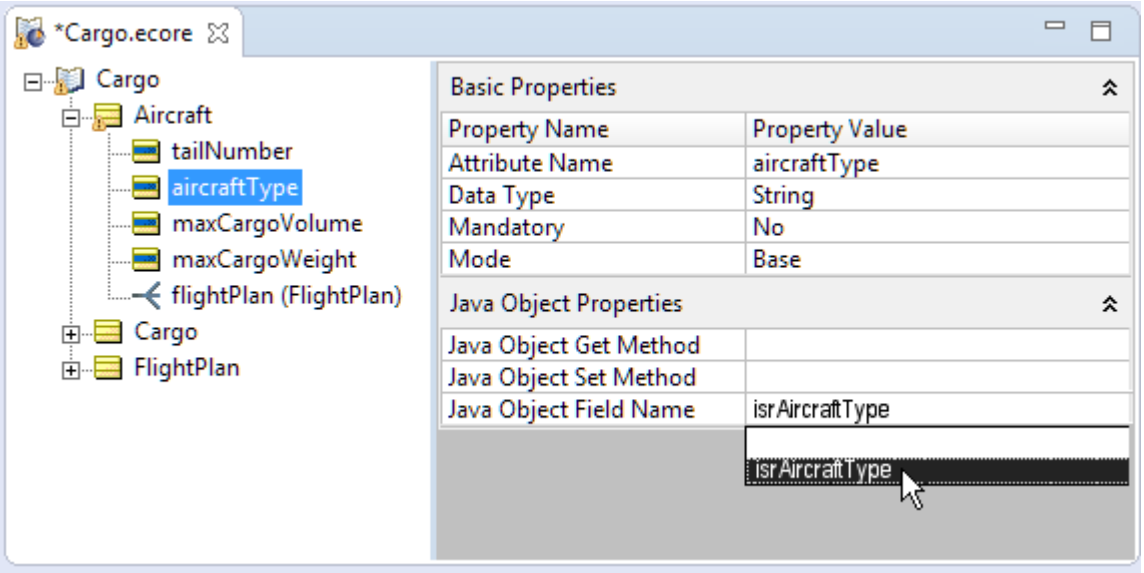
In the case of attribute `flightNumber`, the mapper finds get and set methods that conform to this naming convention, so the method names are inserted into the fields in gray type, as shown in the following figure.

Figure 9: Auto-Mapped Attribute Method Names



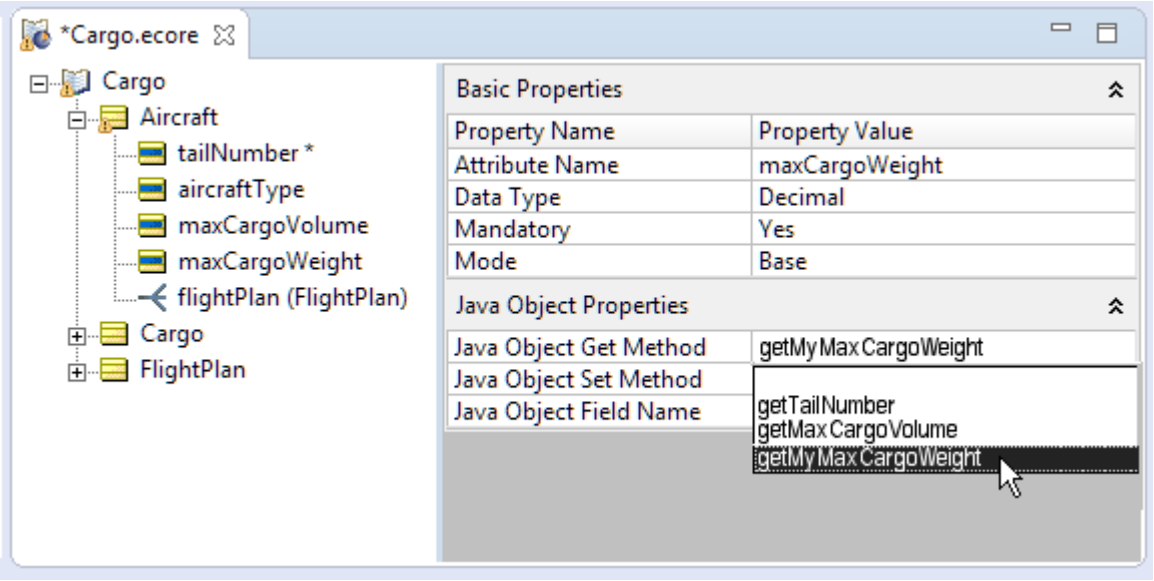
In those cases where the mapper cannot locate the corresponding methods, you will need to select them manually. Notice in the `MyAircraft` class shown in First Portion of MyAircraft Class, no get and set methods exist for `istrAircraftType` since it is a public instance variable. Therefore, we need to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

Figure 10: Manually Mapped Public Instance Variable Name



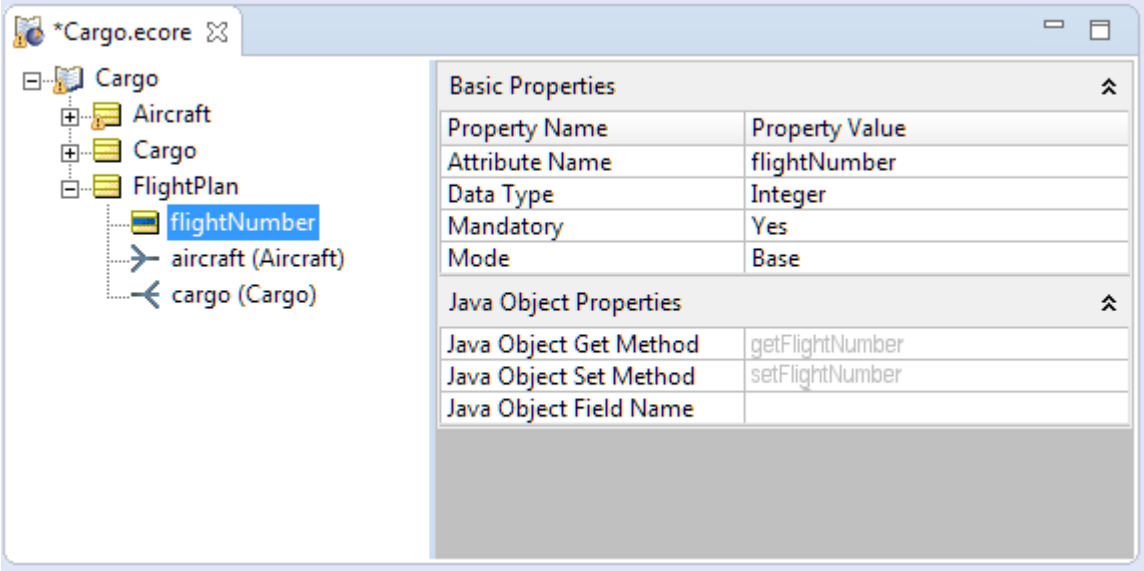
When a class property contains get and set methods, but their names do not conform to the naming convention assumed by the auto-mapper, we must select the method names from the **Java Object Get Method** and **Set Method** drop-downs, as shown in the following figure.

Figure 11: 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 12: 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 3: Java Object Mapping Association Properties

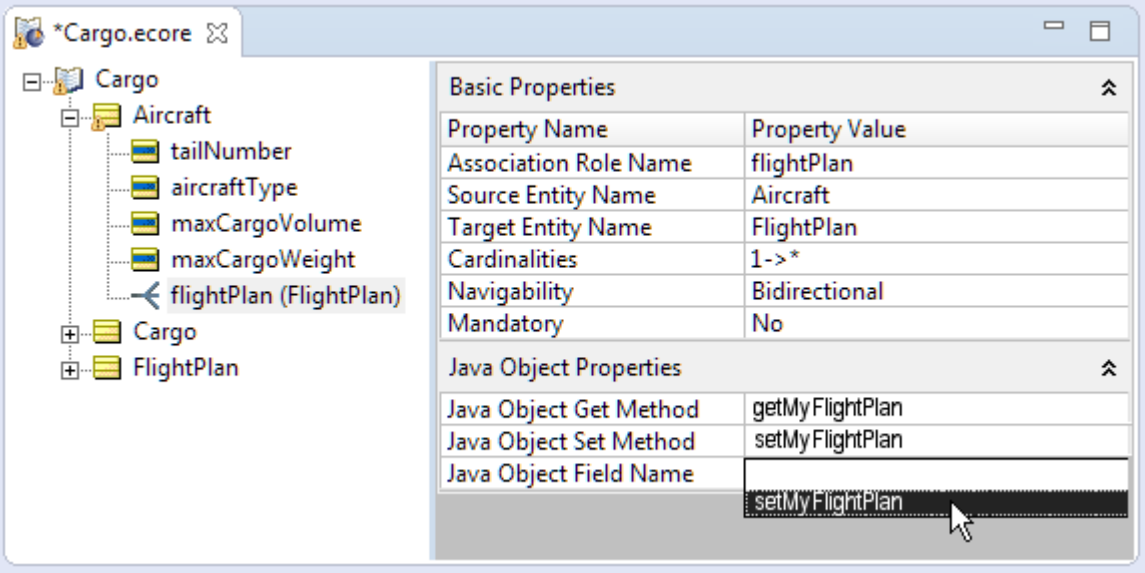
Property	Value
Java Object Get Method	Manually specifies the GET method of a class property that does not conform to naming conventions of the auto-mapper.
Java Object Set Method	Manually specifies the SET method of a class property that does not conform to naming conventions of the auto-mapper.
Java 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 13: 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 14: Manually Mapped Association Get and Set Method Names

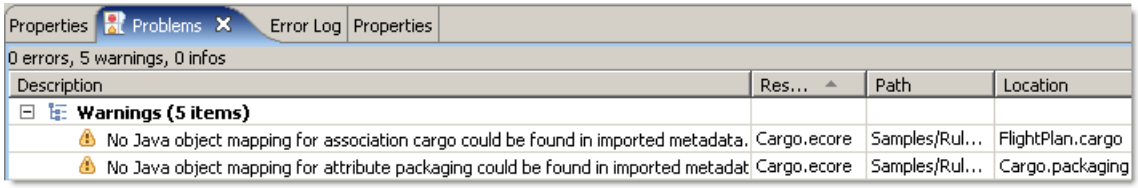


Note: Java generics - Support for type-casted collections is included in Corticon Studio. If your Java Business Objects include type-casted collections (introduced in Java 5), then Corticon will ensure these constraints are interpreted correctly in association processing.

Verify Java object mapping

If there are warning markers on any of the Vocabulary nodes in Entity mapping on page 17, Attribute mapping on page 18, or Association mapping on page 21 whose mappings have not yet been found, each warning has a corresponding message entered in the **Problems** window, as shown:

Figure 15: 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 16: Vocabulary CDT Labels must match Business Object Enumeration Labels (Types)

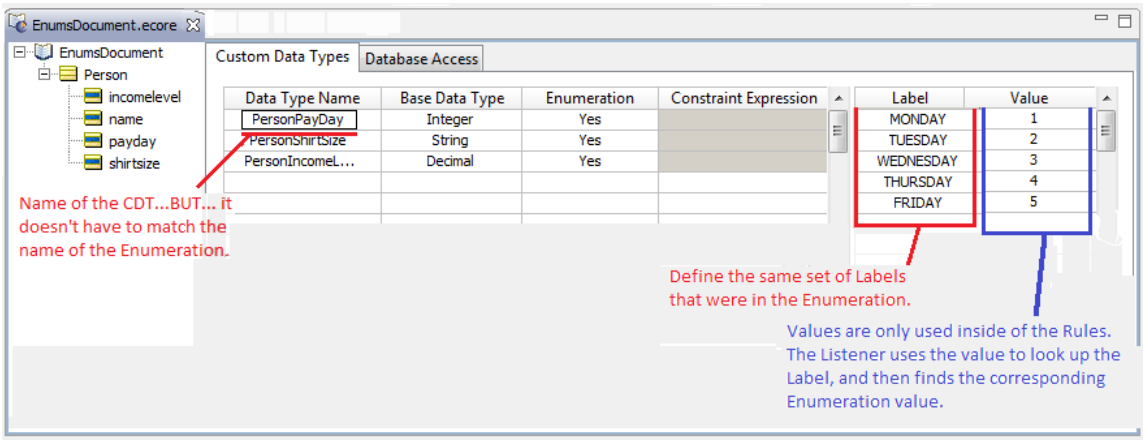
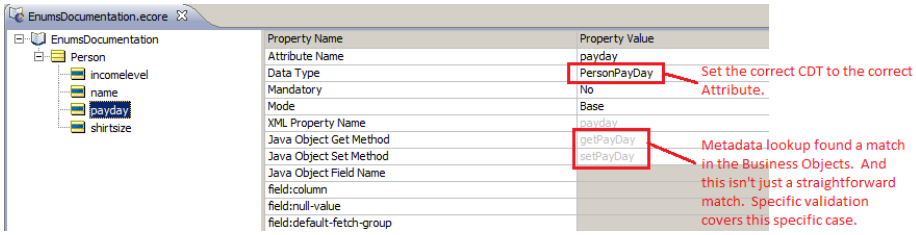


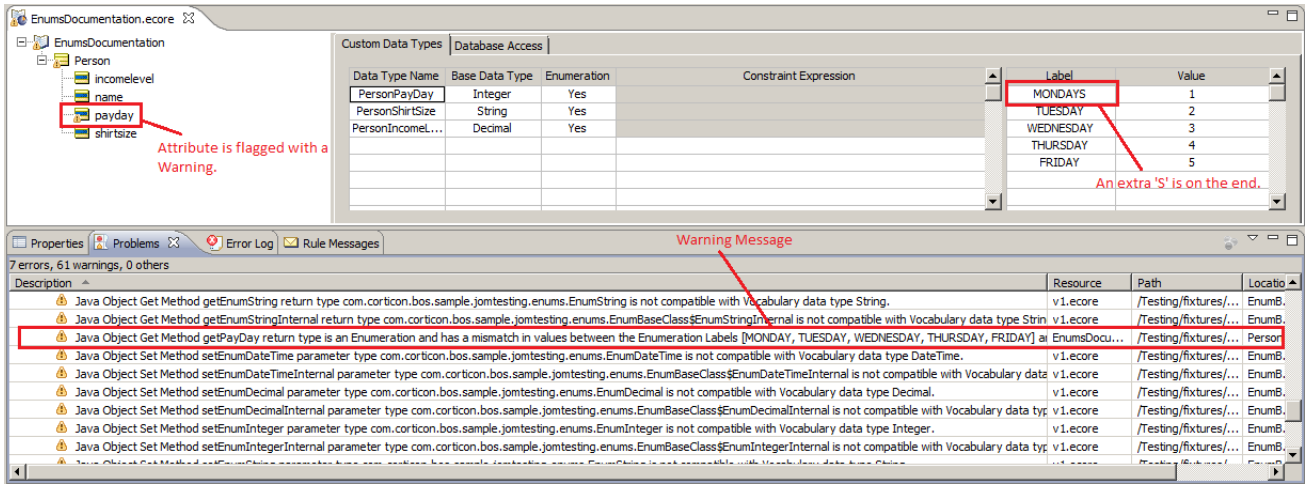
Figure 17: Vocabulary Mapper found correct Metadata based on matching enumeration labels



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 18: Vocabulary CDT Label / Object Enumeration Label Mismatch



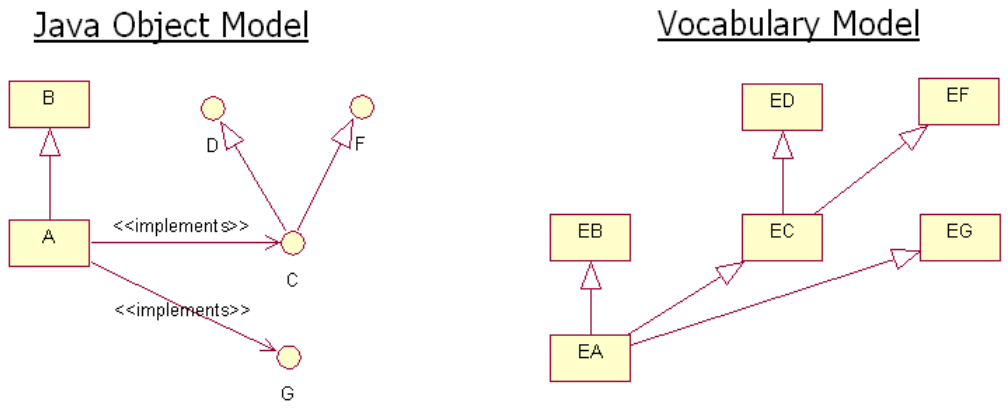
Notice the Vocabulary attribute is “flagged” with the small orange warning icon (shown in the upper left of the figure above). The associated warning message states:

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 19: 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

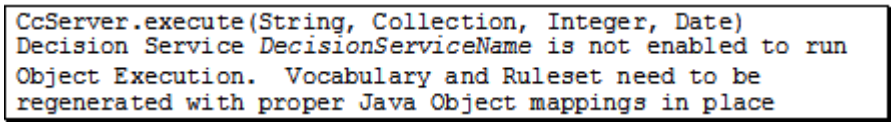
During runtime, when an attribute's value is updated by rules, the update is communicated back to the business object by way of “Listener” classes. Listener classes are compiled at deployment time when Corticon Server detects a Ruleflow using Java Object Messaging. Once compiled, these Listener classes 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 20: Server Error Message When Listeners Not Present



Specifying version in a Java API call

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

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


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

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


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

where:

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

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

Specifying effective timestamp in a Java API call

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

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

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

where:

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

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

Invoking Corticon Server with a Java call

When you deploy Corticon Server in a Java application, you call it with a Java payload -- XML, JSON, or business objects. For more information see *the Java application requirements* topic.

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

- An XML string, or an XML JDOM document, which contains the Decision Service Name as well as the payload data. The payload data must be structured according to the XSD in Corticon Studio. Defining this data payload structure to include as an argument to the execute method is the most common use of the XSD service contract.
- A JSON object which contains the Decision Service Name as well as the payload data.
- The Decision Service Name plus a collection of Java business objects which represent the WorkDocuments.
- The Decision Service Name plus a map of Java business objects which represent the WorkDocuments.

Optional arguments representing Decision Service Version and Effective Timestamp may also be included – these are described in the topic *"Ruleflow versions and effective dates" in the Rule Modeling Guide*.

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

Java Business Objects

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

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

Certain hard-coded classes are included in `CcServer.jar` so as to ensure their inclusion on the JVM's classpath whenever `CcServer.jar` is loaded. The hard-coded Java objects are intended for use when invoking the `ProcessOrder.eds` Decision Service included in the default installation.

5

.NET application requirements

.NET In-Process Deployment

To deploy Corticon within your .NET application you need to include within your application solution the following DLLs.

- `CorticonProxies.dll`
- `CorticonShared.dll`
- `JNBShare.dll`
- `JNBSharedMem_x64.dll`

These DLLs allow a .NET application to call the Corticon Server. In addition, you need to configure your application solution to include

- `jvm.dll`
- `jnbcore.jar`
- `bcel-5.1-jnbridge.jar`

Classpath must include:

- `CcConfig.jar`
- `CcI18nBundles.jar`
- `CcThirdpartyJars.jar`
- `CcDrivers.jar`
- `CcServer.jar`
- `CcLicense.jar`

Sample .NET Application

Corticon Server includes a sample .NET C# application and Microsoft Visual Studio solution demonstrating how to deploy Corticon in a .NET application. This sample application provides a great starting point for you to learn the basics about deploying in a .NET application. The sample is located in:

```
<CORTICON_WORK>/Samples/Clients/In Process/C-sharp
```

Note: This project was created for Visual Studio 2017. You also need Visual C++ Redistributable 2012 (VC++ 11.0), Visual Code C++ Redistributable2015-22, and .NET Framework 4.7 installed. To run the sample you will need a valid Progress Corticon.NET execution license.

Please refer to the sample for the following special considerations.

- Strings need to be passed in as `java.lang.JavaString`.
- Proxies do not expose any constants which are part of the `IccServer`.
- Constants can be passed in as string.