

Corticon Server:

Deploying Web Services with .NET

Copyright

© 2017 Progress Software Corporation and/or one of its subsidiaries or affiliates. All rights reserved.

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

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, Deliver More Than Expected, Icenium, Kendo UI, Making Software Work Together, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Analytics360, AppServer, BusinessEdge, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustCode, JustDecompile, JustMock, JustTrace, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Sitefinity, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

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

Updated: 2017/04/24

Table of Contents

Chapter 1: Conceptual overview of the .NET server.....	7
What is a web service on .NET server?.....	7
What is a Decision Service on .NET server?.....	8
What is the Corticon Server for .NET?.....	8
What is a .NET web services consumer?.....	8
Chapter 2: Getting started with Corticon Server for .NET.....	9
Testing the configuration.....	10
Testing the installed Corticon Server for .NET.....	10
Using a .NET Server installation to set up an in-process server.....	10
Testing as in-process	11
Testing a remote server on IIS.....	13
Chapter 3: Corticon Server for .NET files and API tools.....	17
Setting up Corticon Server for .NET use cases.....	18
The Corticon Server for .NET home and work directories.....	18
The Corticon Server for .NET Sandbox.....	19
Chapter 4: Deploying a Ruleflow to the Corticon Server for .NET.....	21
Creating a Ruleflow for .NET server.....	22
Creating and installing a .NET server Deployment Descriptor file.....	22
Using the .NET Server's Deployment Console Decision Services.....	22
Installing the Deployment Descriptor file on .NET server.....	25
Hot re-deploying .NET server Deployment Descriptor files and Ruleflows.....	26
Chapter 5: Consuming a Decision Service on .NET server.....	27
Integrating and testing a Decision Service on .NET server.....	28
Path 1: Using Corticon Studio as a SOAP client to consume a Decision Service.....	29
Configuring Studio to send a SOAP Message to IIS.....	29
Creating a .NET server test in Corticon Studio	29
Executing the remote .NET server test.....	32
Path 2: Using bundled C# sample code to consume a Decision Service.....	33
Creating the WSDL and proxy files.....	33
Path 3: Using SOAP client to consume a Decision Service.....	35
Web services messaging styles.....	35
Creating a service contract using the Deployment Console.....	35

Creating a request message for a decision service.....	36
Sending a request message to the server.....	37
Path 4: Using JSON/RESTful client to consume a Decision Service on .NET server.....	37
Running the sample JSON Request on .NET server.....	37
Path 5: Using bundled JSON sample code to consume a Decision Service.....	40
Limits of the .NET server default evaluation license.....	40
Troubleshooting .NET server.....	41
 Chapter 6: Using .NET Business Objects as payload for Decision Services.....	43
 Chapter 7: Compiling a Decision Service into an Assembly DLL.....	55
 Chapter 8: Support for Windows Communication Framework (WCF)....	57
Creating WSDL and proxy files.....	57
 Appendix A: Updating your Corticon license JAR for .NET.....	61
 Appendix B: Access to Corticon knowledge resources.....	63

Conceptual overview of the .NET server

This guide describes concepts and procedures for running the Corticon Server for .NET as a web services server, deploying Ruleflows to the Server, exposing the Ruleflows as Decision Services and testing them with document-style SOAP requests. There are other installation, deployment and integration options available beyond the SOAP/Web Services method described here, including Java-centric options using Java objects and APIs. More detailed information on all available methods is contained in the *Server Integration & Deployment Guide*.

For details, see the following topics:

- [What is a web service on .NET server?](#)
- [What is a Decision Service on .NET server?](#)
- [What is the Corticon Server for .NET?](#)
- [What is a .NET web services consumer?](#)

What is a web service on .NET server?

From the business perspective: A Web Service is a software asset that automates a task and can be shared, combined, used, and reused by different people or systems within or among organizations.

From the information systems perspective: A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. [From <http://www.w3c.org>.]

What is a Decision Service on .NET server?

A Decision Service automates a discrete decision-making task. It is implemented as a set of business rules and exposed as a web service (or Java component or .NET library). By definition, the rules within a Decision Service are complete and unambiguous; for a given set of inputs, the Decision Service addresses every logical possibility uniquely, ensuring “decision integrity”.

A Ruleflow is built in Corticon Studio. Once deployed to the Corticon Server for .NET, it becomes a Decision Service.

What is the Corticon Server for .NET?

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

The Corticon Server for .NET is a high-performance, scalable and reliable system resource that manages pools of Decision Services and executes their rules against incoming requests. The Corticon Server for .NET can be easily configured as a web services server, which exposes the Decision Services as true web services.

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

- The **Corticon Server for deploying web services with .NET** -- the product documented here -- facilitates deployment on Windows .NET framework and Microsoft Internet Information Services (IIS) that are packaged in the supported operating systems. The .NET server has its own installer and documentation. See *Deploying Web Service with .NET* for more information.
- The **Corticon Server for deploying web services with Java** is supported on various application servers, databases, and client web browsers. After installation on a supported Windows platform, that server installation's deployment artifacts can be redeployed on various UNIX and Linux web service platforms. See the Progress Software web page [Progress Corticon 5.6 - Supported Platforms Matrix](#) for more information.

What is a .NET web services consumer?

A Web Services Consumer is a software application that makes a request to, and receives a response from, a web service. Most modern application development environments provide native capabilities to consume web services, as do most modern Business Process Management Systems.

Getting started with Corticon Server for .NET

Installing Corticon Server for .NET

To install Corticon Server for .NET

1. Be sure you have an appropriate Corticon license and that your target machine meets the system requirements. Refer to the Progress Software web page [Progress Corticon 5.6 - Supported Platforms Matrix](#) for information on supported .NET Framework and IIS. See "System requirements" in the *Corticon Installation Guide* for more information.
2. Download and install Corticon Server for .NET on your designated Windows server machine. See the *Corticon Installation Guide* for details.
3. Access the appropriate Corticon Knowledgebase article for your platform:
 - [Steps to set up IIS 7.5 on Windows Server 2008 or Windows 7 for Corticon Server for .NET 5.x](#). These instructions are also appropriate for Windows 10.
 - [Steps to set up IIS 8.0 and 8.5 on Windows Server 2012 R1 and R2 for Corticon Server for .NET 5.x](#). These instructions are also appropriate for Windows Server 2016.

For details, see the following topics:

- [Testing the configuration](#)
- [Testing the installed Corticon Server for .NET](#)

Testing the configuration

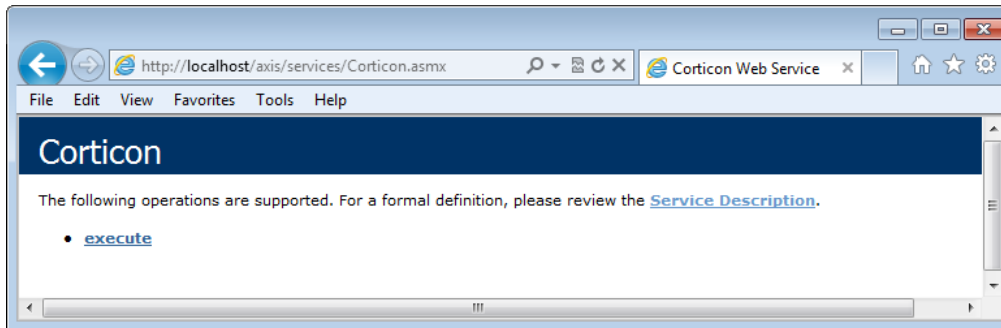
With Corticon Server for .NET installed in IIS, it is a good practice to test the remote server setup to ensure it is running and listening. At this point, no Decision Services have been deployed, so Corticon Server for .NET is not yet ready to process transactions.

In a browser, access the following URLs (assuming that IIS is running on its default port 80):

- `http://localhost/axis/services/CorticonAdmin.asmx`
- `http://localhost/axis/services/Corticon.asmx`
- `http://localhost/axis/services/CorticonExecute.asmx`

Selecting any of these URLs displays a web page in the following format:

Figure 1: Testing the Corticon Web Service



Note: If you do not see appropriate test info, choose **Refresh** in your browser to clear any cached references.

Testing the installed Corticon Server for .NET

With Corticon Server installed in the .NET environment, it is useful to test the installation to ensure Corticon Server is running and listening. At this point, no Decision Services have been deployed, so Corticon Server is not ready to process transactions. However, the Corticon Server API set contains administrative methods that interrogate it and return status information. Several tools are provided to help you perform this test.

Using a .NET Server installation to set up an in-process server

If you choose to manage Corticon Server for .NET in-process via your client application or via a custom container, you are taking responsibility for many of the tasks that are normally performed by a web or application server. But by doing it in your own code, you can optimize your environment and eliminate unneeded overhead. This can result in much smaller footprint installations and faster performance.

Because Corticon Server is a set of classes, it can easily be deployed in-process in an application. When deployed in-process, the following tasks are the responsibility of the client application:

- Management of application settings, ensuring the base set of Corticon Server classes is properly referenced.
- Lifecycle management, including server startup/shutdown
- Security (if needed)

Corticon Server can also be installed into a custom container within any application. It has a small footprint and thus can be installed into client applications including browser-based applications, laptops and mobile devices.

For step-by-step instructions on using the Installer to gain access to Corticon Server's core files, see *“Running the Server and Web Console installer wizard”* in the *Corticon Installation Guide*.

Installation in-process or in a custom container involves these basic steps:

1. Place the following Corticon Server directories and their contents in a directory that is accessible by the application container.
 - /bin
 - /lib
 - /conf
2. Configure the application to reference all DLL files located in the /bin directory.
3. Write code that:
 - Initializes Corticon Server
 - Sets the following three environment variables:
 - CORTICON_HOME - The explicit path that is the root for /bin, /lib, and /conf.
 - CORTICON_WORK_DIR - The explicit path to the working directory
 - CORTICON_LICENSE - The explicit path to the CcLicense.jar file.
 - Deploys the Decision Services into the Corticon Server
 - Requests a decision by marshaling the data payload and then invoking the relevant Corticon Decision Service
 - Processes the response from the Decision Service.

Sample code is provided that demonstrates an in-process deployment of Corticon Server for .NET. This code is packaged as the executable `Corticon-API-Inprocess-Test.exe` in the `[CORTICON_HOME]\Server .NET\samples\bin` directory.

Testing as in-process

Sample code is provided that demonstrates an in-process deployment of Corticon Server for .NET. This code is packaged as the executable `Corticon-API-Inprocess-Test.exe` in the `[CORTICON_HOME]\Server .NET\samples\bin` directory.

The API in-process test opens a Windows console and displays the API menu, as shown below:

Figure 2: Top Portion of the .NET Server in-process API console

```

C:\_56x_install_dir\home\Server\Server.NET\samples\bin\Corticon-API-Inprocess-Test.exe
AppSettings: Corticon Home is ..
AppSettings: Corticon Work Dir is C:\_56x_install_dir\work_dir\Server
Setting Corticon Home to C:\_56x_install_dir\home\Server\Server.NET\samples
Setting Corticon Work Dir to C:\_56x_install_dir\work_dir\Server
CorticonConfiguration.setClasspath() - Start
CorticonConfiguration.setClasspath() java.class.path = C:\_56x_install_dir\home\Server\Server.NET\samples\lib\ant-launcher.jar;C:\_56x_install_dir\home\Server\Server.NET\samples\lib\CcExtensions.jar;C:\_56x_install_dir\home\Server\Server.NET\samples\lib\CcLicense.jar;C:\_56x_install_dir\home\Server\Server.NET\samples\lib\CcTools.jar;C:\_56x_install_dir\home\Server\Server.NET\samples\lib\CorticonLibrary.jar;C:\_56x_install_dir\home\Server\Server.NET\samples\lib\tools.jar;
lstrLogPath C:\_56x_install_dir\work_dir\Server\logs
lstrLogLevel INFO
Starting Progress Corticon Server : 5.6.0.0 -b7168
Progress Corticon Server sandbox location : C:\_56x_install_dir\work_dir\Server\CcServerSandbox

```

The menu displayed in the Windows console is too large to fit on a single printed page, so it has been divided into two screenshots here. In the upper portion of the Windows console, shown in the figure above, the class loading process is visible. Once all classes are loaded, Corticon Server for .NET starts up in the IIS.

Figure 3: Lower Portion of the .NET Server in-process API console

```

C:\_56x_install_dir\home\Server\Server.NET\samples\bin\Corticon-API-Inprocess-Test.exe
Transactions:
-1 - Exit Server Api Test

101 - Add a Decision Service <3 parameters>
102 - Add a Decision Service <6 parameters>
103 - Add a Decision Service <9 parameters>

110 - Load CcServer with .edd file
111 - Load CcServer files from directory

112 - Reload Decision Service
113 - Reload Decision Service <by specific Decision Service Major Version>
114 - Reload Decision Service <by specific Decision Service Major and Minor Version>

115 - Remove Decision Service
116 - Remove Decision Service <by specific Decision Service Major Version>
117 - Remove Decision Service <by specific Decision Service Major and Minor Version>

118 - Clear All Non-Cdd Decision Services

120 - Get Decision Service Names
121 - Get CcServer current info

130 - Execute using a JDOM Document <CorticonRequest Document>
131 - Execute using a XML String <CorticonRequest String>

132 - Execute using a hard-coded set of Business Objects <Collection>
133 - Execute using a hard-coded set of Business Objects <Collection> <by specific Decision Service Major Version>
134 - Execute using a hard-coded set of Business Objects <Collection> <by specific Decision Service Major and Minor Version>
135 - Execute using a hard-coded set of Business Objects <Collection> <by specific execution Date>
136 - Execute using a hard-coded set of Business Objects <Collection> <by specific execution Date and Decision Service Major Version>

137 - Execute using a hard-coded set of Business Objects <HashMap>
138 - Execute using a hard-coded set of Business Objects <HashMap> <by specific Decision Service Major Version>
139 - Execute using a hard-coded set of Business Objects <HashMap> <by specific Decision Service Major and Minor Version>
140 - Execute using a hard-coded set of Business Objects <HashMap> <by specific execution Date>
141 - Execute using a hard-coded set of Business Objects <HashMap> <by specific execution Date and Decision Service Major Version>

150 - Precompile a Ruleflow into a .eds file
151 - Precompile a Ruleflow into a Database Access optimized .eds file

100 - Switch menu to Common Functions
200 - Switch menu to Decision Service Functions
300 - Switch menu to Monitoring Functions
400 - Switch menu to CcServer Functions

Enter transaction number:

```

In the lower portion of the Windows console, shown in the figure above, we see the available API methods of the **Common Functions** (the 100 series) listed by number. You can list the commands in the other series by entering their series number:

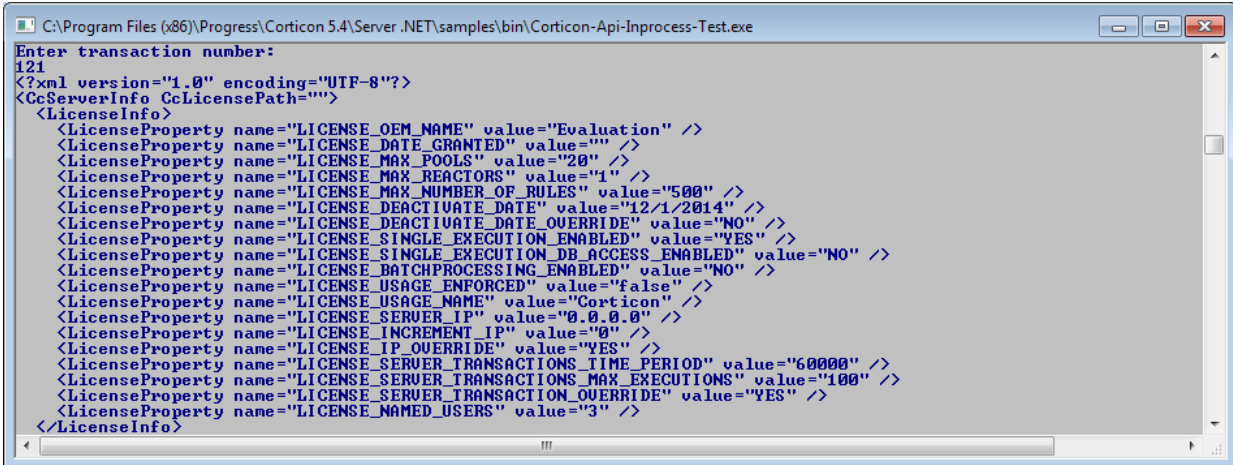
- Enter 200 to list the Decision Service Functions command set
- Enter 300 to list the Monitoring Functions command set
- Enter 400 to list the CcServer Functions command set
- Enter 100 to again list the Common Functions command set

Note: After you enter a transaction, the result is displayed followed a restating of the current command set. You might need to scroll back a bit to see your results.

Since we have not deployed any Ruleflows yet, we will need to use an administrative method to test if Corticon Server is loaded in-process correctly. A good administrative method to call is option #121, **Get CcServer Info**. This choice corresponds directly to the corresponding API method `getCcServerInfo()`.

To try this, enter 121 in the command window. The `CcServerApiTest` class makes a call to the Corticon Server running in-process. It asks for a list of configuration parameters and returns them to the Windows console. The results of the call are shown in the following figure:

Figure 4: .NET Server in-process API console response to command 121



```

C:\Program Files (x86)\Progress\Corticon 5.4\Server .NET\samples\bin\Corticon-Api-Inprocess-Test.exe
Enter transaction number:
121
<?xml version="1.0" encoding="UTF-8"?>
<CcServerInfo CcLicensePath="">
  <LicenseInfo>
    <LicenseProperty name="LICENSE_OEM_NAME" value="Evaluation" />
    <LicenseProperty name="LICENSE_DATE_GRANTED" value="" />
    <LicenseProperty name="LICENSE_MAX_POOLS" value="20" />
    <LicenseProperty name="LICENSE_MAX_REACTORS" value="4" />
    <LicenseProperty name="LICENSE_MAX_NUMBER_OF_RULES" value="500" />
    <LicenseProperty name="LICENSE_DEACTIVATE_DATE" value="12/1/2014" />
    <LicenseProperty name="LICENSE_DEACTIVATE_DATE_OVERRIDE" value="NO" />
    <LicenseProperty name="LICENSE_SINGLE_EXECUTION_ENABLED" value="YES" />
    <LicenseProperty name="LICENSE_SINGLE_EXECUTION_DB_ACCESS_ENABLED" value="NO" />
    <LicenseProperty name="LICENSE_BATCHPROCESSING_ENABLED" value="NO" />
    <LicenseProperty name="LICENSE_USAGE_ENFORCED" value="false" />
    <LicenseProperty name="LICENSE_USAGE_NAME" value="Corticon" />
    <LicenseProperty name="LICENSE_SERVER_IP" value="0.0.0.0" />
    <LicenseProperty name="LICENSE_INCREMENT_IP" value="0" />
    <LicenseProperty name="LICENSE_IP_OVERRIDE" value="YES" />
    <LicenseProperty name="LICENSE_SERVER_TRANSACTIONS_TIME_PERIOD" value="60000" />
    <LicenseProperty name="LICENSE_SERVER_TRANSACTIONS_MAX_EXECUTIONS" value="100" />
    <LicenseProperty name="LICENSE_SERVER_TRANSACTION_OVERRIDE" value="YES" />
    <LicenseProperty name="LICENSE_NAMED_USERS" value="3" />
  </LicenseInfo>
</CcServerInfo>

```

We haven't loaded any Decision Services, so Corticon Server is basically replying with an empty status message. But the important thing is that we have verified that Corticon Server for .NET is running correctly in-process and is listening for, and responding to, calls. At this stage in the deployment, this is all we want to verify.

There is also a sample application test for the in-process Corticon Server. This code is packaged as the executable `Corticon-Api-Example.exe` in the `[CORTICON_HOME]\Server .NET\samples\bin` directory.

Testing a remote server on IIS

To test that Corticon Server deployed as a SOAP service is running correctly, all you need is a SOAP client or the sample batch file provided and described below.

Testing the installation here assumes you have already set up IIS, and installed Corticon Server for .NET as a Web Service. Be sure that you have created an application from the axis directory, and that it is bound to application pools appropriately.

Because a SOAP service is listening for SOAP calls, we need a way to invoke an API method via a SOAP message then send that message to Corticon Server using a SOAP client. In the sample code supplied in the default installation, Corticon provides an easy way to send API calls through a SOAP message.

Sample code is provided that demonstrates a remote deployment of Corticon Server for .NET on IIS. This code is packaged as the executable `Corticon-Api-Remote-Test.exe` in the `Server\bin` directory of your Corticon Server for .NET installation directory.

When executed, it opens a Windows console and displays the API menu, as shown below:

Figure 5: Top Portion of the .NET Server remote API console

```

C:\_56x_install_dir\home\Server\Server.NET\samples\bin\Corticon-Api-Remote-Test.exe
AppSettings: Corticon Home is ..
AppSettings: Corticon Work Dir is C:\_56x_install_dir\work_dir\Server
Setting Corticon Home to C:\_56x_install_dir\home\Server\Server.NET\samples
Setting Corticon Work Dir to C:\_56x_install_dir\work_dir\Server
CorticonConfiguration.setClasspath() - Start
Server.NET\samples\lib\ant-launcher.jar;C:\_56x_install_dir\home\Server\Server.NET\samples\lib\CcExtensions.jar;C:\_56x_install_dir\home\Server\Server.NET\samples\lib\CcTools.jar;C:\_56x_install_dir\home\Server\Server.NET\samples\lib\CorticonLibrary.jar;C:\_56x_install_dir\home\Server\Server.NET\samples\lib\tools.jar;
lstrLogPath C:/_56x_install_dir/work_dir/Server/logs
lstrLogLevel INFO

-----
Axis Servlet Location : http://localhost:80
Execute Servlet (Message Style) : http://localhost:80/axis/services/Corticon
Execute Servlet (RPC Style) : http://localhost:80/axis/services/CorticonExecute
Admin Servlet : http://localhost:80/axis/services/CorticonAdmin
Heartbeat : http://localhost:80/axis/services/CorticonHeartbeat
Output Directory : C:/_56x_install_dir/work_dir/Server/output

-----
Current Apache Axis Location: http://localhost:80

```

The menu displayed in the Windows console is too large to fit on a single printed page, so it has been divided into two screenshots here. In the upper portion of the Windows console, shown in the figure above, the classpath definition process is visible. Once all classes are loaded, the Corticon Server for .NET starts up in the IIS, which is needed by our simple SOAP client class.

Figure 6: Lower Portion of the .NET Server remote API console

```

C:\_56x_install_dir\home\Server\Server.NET\samples\bin\Corticon-Api-Remote-Test.exe
Transactions:
-1 - Exit Server Api Test

0 - Change Connection Parameters

101 - Add a Decision Service (3 parameters)
102 - Add a Decision Service (6 parameters)
103 - Add a Decision Service (9 parameters)

110 - Load CcServer with .cdd file
111 - Load CcServer files from directory

112 - Reload Decision Service
113 - Reload Decision Service <by specific Decision Service Major Version>
114 - Reload Decision Service <by specific Decision Service Major and Minor Version>

115 - Remove Decision Service
116 - Remove Decision Service <by specific Decision Service Major Version>
117 - Remove Decision Service <by specific Decision Service Major and Minor Version>

118 - Clear All Non-Cdd Decision Services

120 - Get Decision Service Names
121 - Get CcServer current info

130 - Execute SOAP Document Style (CorticonRequest Document)
131 - Execute SOAP RPC Style (CorticonRequest String)

150 - Precompile a Ruleflow into a .eds file
151 - Precompile a Ruleflow into a Database Access optimized .eds file

200 - Switch menu to Common Functions
300 - Switch menu to Monitoring Functions
400 - Switch menu to CcServer Functions

Enter transaction number:

```

In the lower portion of the Windows console, shown in the figure above, we see the available API methods of the **Common Functions** (the 100 series) listed by number. You can list the commands in the other series by entering their series number:

- Enter 200 to list the Decision Service Functions command set
- Enter 300 to list the Monitoring Functions command set
- Enter 400 to list the CcServer Functions command set

- Enter 100 to again list the Common Functions command set

Note: After you enter a transaction, the result is displayed followed a restating of the current command set. You might need to scroll back a bit to see your results.

Since we have not deployed any Ruleflows yet, we will use an administrative method to test if Corticon Server is correctly installed as a SOAP service inside our web server. A good administrative method to call is transaction #121, **Get CcServer current info**. This choice corresponds directly to the API method `getCcServerInfo()`.

To try this, confirm that IIS is running, and then enter 121 in the command window. The `CcServerAxisTest` class makes a call to the Corticon Server SOAP Servlet. It asks for a list of configuration parameters and returns them to the Windows console. The results of the call are shown in the following figure:

Figure 7: .NET Server remote API console response to command 121

```

Enter transaction number:
121
<?xml version="1.0" encoding="UTF-8"?>
<CcServerInfo CcLicensePath="">
  <LicenseInfo>
    <LicenseProperty name="LICENSE_OEM_NAME" value="Evaluation" />
    <LicenseProperty name="LICENSE_DATE_GRANTED" value="" />
    <LicenseProperty name="LICENSE_MAX_POOLS" value="10" />
    <LicenseProperty name="LICENSE_MAX_REACTORS" value="5" />
    <LicenseProperty name="LICENSE_MAX_NUMBER_OF_RULES" value="200" />
    <LicenseProperty name="LICENSE_DEACTIVATE_DATE" value="6/1/2013" />
    <LicenseProperty name="LICENSE_DEACTIVATE_DATE_OVERRIDE" value="NO" />
    <LicenseProperty name="LICENSE_SINGLE_EXECUTION_ENABLED" value="YES" />
    <LicenseProperty name="LICENSE_SINGLE_EXECUTION_DB_ACCESS_ENABLED" value="NO" />
    <LicenseProperty name="LICENSE_BATCHPROCESSING_ENABLED" value="YES" />
    <LicenseProperty name="LICENSE_USAGE_ENFORCED" value="false" />
    <LicenseProperty name="LICENSE_USAGE_NAME" value="Corticon" />
    <LicenseProperty name="LICENSE_SERVER_IP" value="0.0.0.0" />
    <LicenseProperty name="LICENSE_INCREMENT_IP" value="0" />
    <LicenseProperty name="LICENSE_IP_OVERRIDE" value="YES" />
    <LicenseProperty name="LICENSE_SERVER_TRANSACTIONS_TIME_PERIOD" value="60000" />
    <LicenseProperty name="LICENSE_SERVER_TRANSACTIONS_MAX_EXECUTIONS" value="100" />
    <LicenseProperty name="LICENSE_SERVER_TRANSACTION_OVERRIDE" value="YES" />
  </LicenseInfo>
  <CDDs />
  <NonCDDs />
  <RegisteredTrackingAttributes />
  <ServiceStartups />
  <LoggingStartups />
</CcServerInfo>

Transaction completed.

```

The response verifies that our Corticon Server is running correctly as a SOAP Servlet and is listening for, and responding to, calls. At this stage in the deployment, this is all we want to verify.

Corticon Server for .NET files and API tools

Corticon Server for deploying web services with .NET facilitates deployment on Windows .NET framework and Microsoft Internet Information Services (IIS) that are packaged in the supported operating systems. This guide points out features for various deployment technologies and strategies.

First we'll deploy a Ruleflow to the .NET server as a Decision Service, then we'll try out consuming that Decision Service with various manual, SOAP/XML, and JSON/RESTful techniques.

After that we'll explore some advanced topics that enable you to:

- Use Java Object Messaging (JOM) for payloads
- Compile a Decision Service into an Assembly.dll to increase performance
- Take a look at Windows Communication Framework (WCF)

But before exploring these features, you should get acquainted with some of the .NET server files and API tools.

For details, see the following topics:

- [Setting up Corticon Server for .NET use cases](#)
- [The Corticon Server for .NET home and work directories](#)
- [The Corticon Server for .NET Sandbox](#)

Setting up Corticon Server for .NET use cases

In most production deployments, Corticon Server JARs are bundled and given an interface class or classes. The interface class is often called a “helper” or “wrapper” class because its purpose is to receive the client application's invocation, translate it (if necessary) into a call which uses Corticon Server's native API, and then forwards the call to Corticon Server's classes. The type of interface class depends on the container where you intend to deploy the Corticon Server.

Corticon Studio makes in-process calls to the same Corticon Server classes (although packaged differently-- see the following topic) when Ruletests are executed. This ensures that Ruleflows behave exactly the same way when executed in Studio Ruletests as they do when executed by Corticon Server, no matter how Corticon Server is installed.

The Corticon Server for .NET home and work directories

As a Corticon installation completes, it tailors two properties that define its global environment. These variables are used throughout the product to determine the relative location of other files.

Corticon environment

The installer establishes a common environment configuration file, `\bin\corticon_env.bat`, at the program installation location. That file defines the Progress Corticon runtime environment so that most scripts simply call it to set common global environment settings, such as `CORTICON_HOME` and `CORTICON_WORK_DIR` (and, in some cases, simply `CORTICON_WORK`.)

CORTICON_HOME

The explicit path of the installation home directory -- either the default location, `C:\Program Files\Progress\Corticon 5.6`, or the preferred location you specified -- is assigned to `[CORTICON_HOME]`.

CORTICON_WORK_DIR

The explicit path of the work directory -- either the default location, `C:\Users\{username}\Progress\CorticonWork 5.6`, or the preferred location you specified -- is assigned to `[CORTICON_WORK_DIR]`.

Note: The Corticon **Start** menu provides a **Corticon Command Prompt** command that calls `corticon_env.bat`, adds several `[CORTICON_HOME]` script paths to the `PATH` so that you can launch scripts by name from several locations -- `\bin`, `\Server\bin`, `\Server\pas\bin`, `\Studio\bin`, and, `\Studio\eclipse` -- and then relocates the prompt to the root of the Corticon work directory.

It is a good practice to use global environment settings

Many file paths and locations are determined by the `CORTICON_HOME` and `CORTICON_WORK_DIR` variables. Be sure to call `corticon_env.bat`, and then use these variables in your scripts and wrapper classes so that they are portable to deployments that might have different install paths.

Note: While you could change these locations with the assurance that well-behaved scripts will follow your renamed path or location, you might also encounter unexpected behaviors from any that do not. Also, issues might arise when running update, upgrade, and uninstall utilities.

The Corticon Server for .NET Sandbox

When Corticon Server starts up, it checks for the existence of a “sandbox” directory. This Sandbox is a directory structure used by Corticon Server to manage its state and deployment code.

The location of the Sandbox is controlled by `com.corticon.ccserver.sandboxDir` settings in your `brms.properties` file located at the root of `[CORTICON_WORK_DIR]`. For more information, see *Server properties* described in the *Integration and Deployment Guide*.

This configuration setting is defined by the `CORTICON_WORK_DIR` variable, in this case:

```
com.corticon.ccserver.sandboxDir=%CORTICON_WORK_DIR%/%CORTICON_SETTING%/CcServerSandbox
```

In a default Windows installation, the result for this is

`C:\Users\{username}\Progress\CorticonWork_5.6\SER\CcServerSandbox`. In other words, in the `SER` subdirectory of the `CORTICON_WORK_DIR`. This directory is created (as well as peer directories, logs and output) during the first launch of Corticon Server.

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.

Deploying a Ruleflow to the Corticon Server for .NET

Just because Corticon Server for .NET has been installed does not mean it is ready to process transactions. It must still be loaded with one or more Ruleflows. Once a Ruleflow has been loaded, or deployed, to the Corticon Server we call it a Decision Service because it is a service ready and able to make decisions for any external application or process (client) that requests the service properly.

Loading the Corticon Server with Ruleflows can be accomplished in two ways:

- **Deployment Descriptor files** - This is the easiest method and the one we will use in this guide because it is also the method typically used in production web service deployments.
- **.NET APIs** - This method requires more knowledge of the Server for .NET API set, and is not discussed in this guide. For more information, see the topics in *"Packaging and deploying Decision Services" in the Integration and Deployment Guide*.

Both methods are described more thoroughly in the *Integration & Deployment Guide*.

For details, see the following topics:

- [Creating a Ruleflow for .NET server](#)
- [Creating and installing a .NET server Deployment Descriptor file](#)

Creating a Ruleflow for .NET server

It is assumed you have already created a Ruleflow suitable for deployment. If you have completed the *Corticon Tutorial: Basic Rule Modeling*, then you created a sample Ruleflow that is ready for deployment to the Server for .NET. We will use that Ruleflow here.

You can take a shortcut to simply access the reference example Ruleflow, `tutorial_example.erf`, in the completed tutorial located in the server's `[CORTICON_WORK_DIR]\Samples\RuleProjects\Tutorial\Tutorial-Done`

Creating and installing a .NET server Deployment Descriptor file

A Deployment Descriptor file tells the Corticon Server for .NET which Ruleflows to load and how to handle transaction requests for those Ruleflows. A Deployment Descriptor file has the suffix `.cdd`, and we will often simply refer to it as a `.cdd` file.

Important: The `.cdd` file “points” at the Ruleflow via a path name – it is important that this path **not** contain space characters. For example, a Ruleflow stored in `My Documents` cannot be referenced by a Deployment Descriptor file because its path contains a space. Even though the default storage location for your Ruleflow files is inside a Corticon Studio installation's `[CORTICON_WORK_DIR]\Samples\RuleProjects\Tutorial\Tutorial-Done` (which contains a space), we avoid the problem by substituting `../../../../` as a relative reference to the directory structure.

Deployment Descriptors are easily created using the Deployment Console, which is installed by the Server installer.

Using the .NET Server's Deployment Console Decision Services

To start the Corticon Deployment Console for .NET, choose the Windows **Start** menu command **All Programs > Progress > Corticon 5.6 > Corticon .NET Deployment Console** to launch the executable file `Server.NET\samples\bin\DeploymentConsole.exe`.

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

Figure 8: Left Portion of Deployment Console, with Deployment Descriptor File Settings Numbered

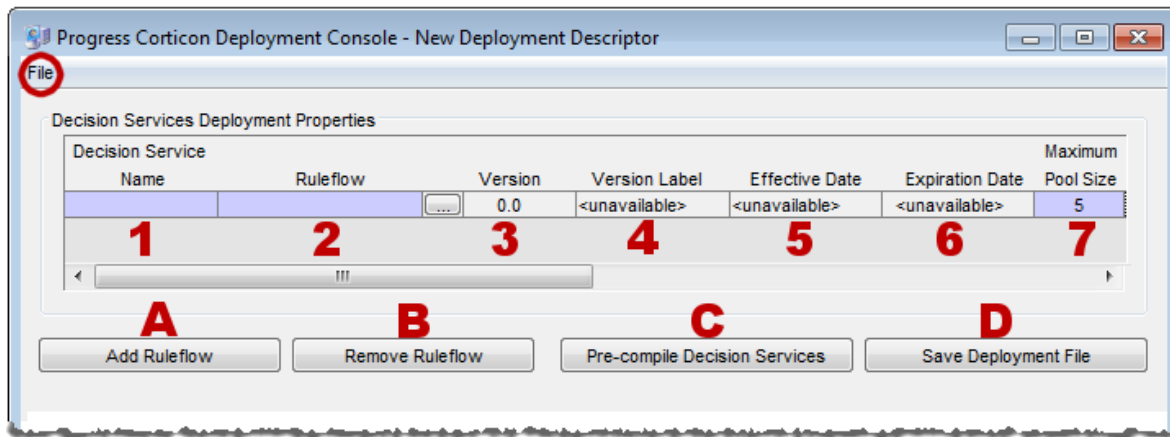
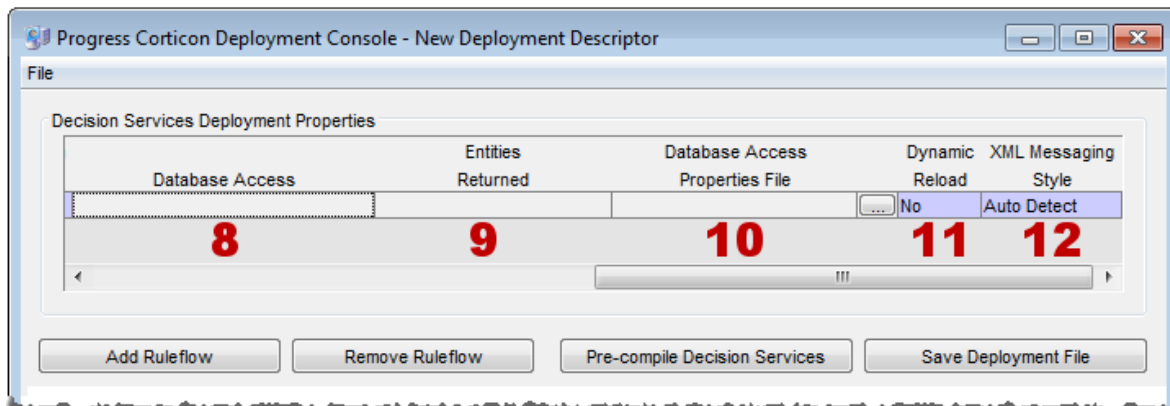


Figure 9: Right Portion of Deployment Console, with Deployment Descriptor File Settings Numbered



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

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

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

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

1. **Decision Service Name** - A unique identifier or label for the Decision Service. It is used when invoking the Decision Service, either via an API call or a SOAP request message. See [Invoking Corticon Server](#) for usage details.
2. **Ruleflow** - All Ruleflows listed in this section are part of this Deployment Descriptor file. Deployment properties are specified on each Ruleflow. Each row represents one Ruleflow. Use the button to navigate to a Ruleflow file and select it for inclusion in this Deployment Descriptor file. Note that Ruleflow *absolute* pathnames are shown in this section, but *relative* pathnames are included in the actual .cdd file.

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

3. **Version** - the version number assigned to the Ruleflow in the **Ruleflow > Properties** window of Corticon Studio. Note that this entry is editable only in Corticon Studio and not in the Deployment Console. A discussion of how Corticon Server processes this information is found in the topics *"Decision Service Versioning and Effective Dating" of the Integration and Deployment Guide*. Also see the *Quick Reference Guide* for a brief description of the Ruleflow Properties window and the *Rule Modeling Guide* for details on using the Ruleflow versioning feature. It is displayed in the Deployment Console simply as a convenience to the Ruleflow deployer.
4. **Version Label** - the version label assigned to the Ruleflow in the **Ruleflow > Properties** window of Corticon Studio. Note that this entry is editable only in Corticon Studio and not in the Deployment Console. See the *Quick Reference Guide* for a brief description of the Ruleflow Properties window and the purpose of the Ruleflow versioning feature.
5. **Effective Date** - The effective date assigned to the Ruleflow in the **Ruleflow > Properties** window of Corticon Studio. Note that this entry is editable only in Corticon Studio and not in the Deployment Console. A discussion of how Corticon Server processes this information is found in the topics *"Decision Service Versioning and Effective Dating" of the Integration and Deployment Guide*. Also see the *Quick Reference Guide* for a brief description of the Ruleflow Properties window and the purpose of the Ruleflow effective dating feature.
6. **Expiration Date** - The expiration date assigned to the Ruleflow in the **Ruleflow > Properties** window of Corticon Studio. Note that this entry is editable only in Corticon Studio and not in the Deployment Console. A discussion of how Corticon Server processes this information is found in the topics *"Decision Service Versioning and Effective Dating" of the Integration and Deployment Guide*. Also see the *Quick Reference Guide* for a brief description of the Ruleflow Properties window and the purpose of the Ruleflow expiration dating feature.
7. **Maximum Pool Size** - Specifies how many execution threads for this Decision Service will be added to the Execution Queue. This parameter is an issue only when Allocation is turned on. If you are evaluating Corticon, your license requires that you set the parameter to 1. See *'Multi-threading, concurrency reactors, and server pools' in "Inside Corticon Server" section of the Integration and Deployment Guide* for more information.

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

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

8. **Database Access** - *Active if your Corticon license enables EDC* - Controls whether the deployed Rule Set has direct access to a database, and if so, whether it will be read-only or read-write access.
9. **Entities Returned** - *Active if your Corticon license enables EDC* - Determines whether the Corticon Server response message should include all data used by the rules including data retrieved from a database (**All Instances**), or only data provided in the request and created by the rules themselves (**Incoming/New Instances**).
10. **Database Access Properties File** - *Active if your Corticon license enables EDC* - The path and filename of the database access properties file (that was typically created in Corticon Studio) to be used by Corticon Server during runtime database access. Use the adjacent



button to navigate to a database access properties file.

11. **Dynamic Reload** - When **Yes**, the `ServerMaintenanceThread` will detect if the Ruleflow or `.eds` file has been updated; if so, the Decision Service will be updated into memory and -- for any subsequent calls to that Decision Service -- that execution Thread will execute against the newly updated Rules. When **No**, the `CcServerMaintenanceThread` will ignore any changes to the Ruleflow or `.eds` file. The changes will not be read into memory, and all execution Threads will execute against the existing Rules that are in memory for that Decision Service.
12. **XML Messaging Style** - Determines whether request messages for this Decision Service should contain a flat (**Flat**) or hierarchical (**Hier**) payload structure. The [Decision Service Contract Structures](#) section of the Integration chapter provides samples of each. If set to **Auto Detect**, then Corticon Server will accept either style and respond in the same way.

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

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

Installing the Deployment Descriptor file on .NET server

Once Corticon Server for .NET has been installed and deployed on IIS, the following sequence occurs:

1. IIS server starts.
2. Corticon Server for .NET starts as a web service in IIS.
3. Corticon Server looks for Deployment Descriptor files in the `<IISRoot>\axis\cdd` directory.
4. Corticon Server for .NET loads into memory the Ruleflow(s) referenced by the Deployment Descriptor files, and creates Reactors for each according to their minimum pool size settings. At this stage, we say that the Ruleflows have become Decision Services because they are now callable by external applications and clients.

In order for the Corticon Server for .NET to find Deployment Descriptor files when it looks in step 3, we must ensure that the `.cdd` files are stored in the default location, the `[CORTICON_WORK_DIR]\cdd` directory. When creating `.cdd` files, save them to this directory so they become accessible to the deployed Corticon Server for .NET.

Note: This location is configurable, but be aware that Deployment Descriptor files usually contain paths relative to where they were created; as such, copying or moving them to a different location can make the file behave incorrectly. For more information, see the topic *"Packaging and deploying Decision Services" in the Integration and Deployment Guide*.

Now, when the startup sequence reaches step 3 above, the server knows where all Ruleflows are located because `.cdd` files contain their pathnames.

Hot re-deploying .NET server Deployment Descriptor files and Ruleflows

Changes to a Deployment Descriptor file or any of the Ruleflows it references do **not** require restarting IIS. A maintenance thread in the Corticon Server for .NET watches for additions, deletions, and changes and updates appropriately. A Ruleflow can be modified in Studio even while it is also simultaneously deployed as a Decision Service and involved in a transaction - Server can be configured to update the Decision Service dynamically for the very next transaction.

Having selected **No** for the **Dynamic Reload** setting earlier, our `tutorial_example` Decision Service will not update automatically when the `.erf` file is changed. To enable this automatic refresh, choose **Yes** for the **Dynamic Reload** setting.

Note: When using .NET Server on IIS 7.5 and EDC-enabled Decision Services, redeploying a Decision Service to use a different database is not supported. To change the database or a deployed Decision Service you need to; undeploy the Decision Service, restart the .NET server (or server host), and then redeploy the Decision Service with the preferred database.

Consuming a Decision Service on .NET server

So far:

1. We have installed Corticon Server for .NET files onto a workstation or server .
2. We have configured Corticon Server for .NET as a web service onto IIS.
3. We have used the **Deployment Console** to generate a Deployment Descriptor file for our sample Ruleflow.
4. We have installed the Deployment Descriptor file in the location where Corticon Server for .NET looks when it starts.

Now we are ready to consume this Decision Service by sending a real XML/SOAP “request” message and inspecting the “response” message it returns.

For details, see the following topics:

- [Integrating and testing a Decision Service on .NET server](#)
- [Path 1: Using Corticon Studio as a SOAP client to consume a Decision Service](#)
- [Path 2: Using bundled C# sample code to consume a Decision Service](#)
- [Path 3: Using SOAP client to consume a Decision Service](#)
- [Path 4: Using JSON/RESTful client to consume a Decision Service on .NET server](#)
- [Path 5: Using bundled JSON sample code to consume a Decision Service](#)
- [Limits of the .NET server default evaluation license](#)
- [Troubleshooting .NET server](#)

Integrating and testing a Decision Service on .NET server

In order to use a Decision Service in a process or application, it is necessary to understand the Decision Service's service contract, also known as its interface. A service contract describes in precise terms the kind of input a Decision Service is expecting, and the kind of output it returns following processing. In other words, a service contract describes how to *integrate* with a Decision Service.

When an external process or application sends a request message to a Decision Service that complies with its service contract, the Decision Service receives the request, processes the included data, and sends a response message. When a Decision Service is used in this manner, we say that the external application or process has successfully “consumed” the Decision Service.

This guide describes four paths for consuming a Decision Service:

- [Path 1](#)

Use Progress Corticon as a SOAP client to send and receive SOAP messages to a Decision Service running on a remote Corticon Server - This is different from testing Ruleflows in Corticon “locally.” This path is the easiest method to use and requires the least amount of technical knowledge to successfully complete. If you have already installed Corticon Studio, then you have all necessary components to complete this path. If not but want to follow this path, we recommend completing the *Corticon Installation Guide* and the *Corticon Studio Tutorial: Basic Rule Modeling* before continuing on this path.

- [Path 2](#)

Manually integrate and test a Decision Service - In this path, we will use bundled sample code (a command file) to send a request message built in Corticon Studio's Tester, and display the results. This path requires more technical knowledge and confidence to complete, but illustrates some aspects of the software which may be interesting to a more technical audience. If you have already installed Studio, then you have all necessary components to complete this path. If not but want to follow this path, we recommend completing the *Corticon Installation Guide* and the *Corticon Studio Tutorial: Basic Rule Modeling* before continuing on this path.

- [Path 3](#)

Use a commercially available SOAP client to integrate with and test a Decision Service - This SOAP client will read a web-services-standard service contract, generate a request message from it, send it to the Corticon Server and display the response message.

- [Path 4](#)

Use JSON/RESTful client to consume a Decision Service on .NET server - This RESTful client will read a web-services-standard service contract (discussed below), generate a request message from it, send it to the Corticon Server and display the response message.

- [Path 5](#)

Use bundled JSON/REST sample code to consume a Decision Service - A sample of .NET code is provided that you can tailor to execute Decision Services with REST/JSON.

Path 1: Using Corticon Studio as a SOAP client to consume a Decision Service

In this path, we will use Corticon Studio as a SOAP client to execute Decision Services running on a remote Corticon Server.

Configuring Studio to send a SOAP Message to IIS

Corticon Studio is configured by default to query a `localhost` web server on port 8850. Because we are using IIS, we'll change the port used by Studio to send Test messages.

Note: Instead of `localhost`, you can use the static IP or DNS-resolvable name of the host -- a good idea as it emulates actual deployment.

To configure the port:

1. Navigate to the directory `[CORTICON_WORK_DIR]`.
2. Edit the file `brms.properties` in that location.

Note: If you specified a preferred name and location of the override properties file in Studio preferences, edit that file as it the one that will be the last loaded.

3. Add the following line to the file so that your IIS points to its server port:

```
com.corticon.deployment.soapbindingurl_2=http://localhost:80/axis
```

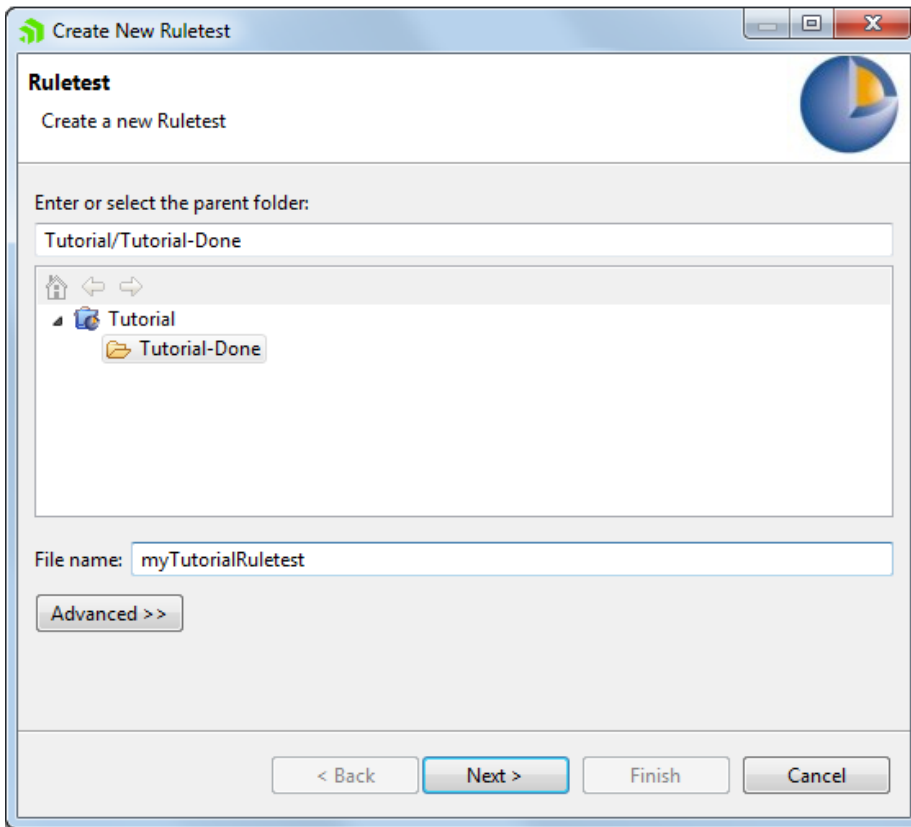
4. Save the edited file.
5. Restart Corticon Studio.

The edited value is added to the list of Remote Servers.

Creating a .NET server test in Corticon Studio

Important: Even though we are using Corticon Studio to test, we will use its *remote* testing feature, which executes a Decision Service running on a Corticon Server (remotely), not a Ruleflow open in Corticon Studio (locally). To keep this distinction clear, we are **not** going to open `tutorial_example.erf` in Corticon Studio – it is not necessary since we are really testing the Decision Service sample `Cargo` .NET sample deployed and running as an application on a Microsoft IIS server.

1. In Corticon Studio, create a new Ruletest by choosing the menu command **File > New > Ruletest**.



2. Select the **parent folder** for the new Ruletest.
3. Enter a file name for the new Ruletest.
4. Click **Next** to open the **Select Test Subject** panel.
5. Click the **Run against Server** tab to open its panel, as shown:

Select Test Subject

Run in Studio | Run against Server

Server URL:
http://localhost:8850/axis

Credentials are required if authentication is enabled on Server:
Username: Password:

Decision Services:
Refresh

Name	Major	Minor	Effective Start Date	Effective Stop Date
AllocateTrade	1	14		
Candidates	1	14		
Cargo	0	16		
Freight	1	9		
Freight	2	1		

Optional Overrides

Major Version: 0
Minor Version: 16
Effective Target Date: / / Time: 0:00 AM Clear

OK Cancel

- For the **Server URL**, enter a URL; for example, an IIS server installed (and running) on its default port installed on the same machine as the Studio: `http://localhost:80/axis`. Your entry is validated when you click **Refresh**, and persisted in your Studio. Once you have persisted URLs, click on the right side of the Server URL area to open the dropdown menu to make your selection.

Note: Only a few Server URLs are persisted this way. If you have a larger list that you want to edit, see "Specifying server URLs for access to test subjects" in the *Quick Reference Guide*.

Click **Refresh** to populate the list of deployed Decision Services on that server.

- Click on an appropriate Decision Service for this Ruletest:

Name	Major	Minor	Effective Start Date	Effective Stop Date
AllocateTrade	1	14		
Candidates	1	14		
Cargo	1	0		
ProcessOrder	1	10		

- You can click **OK** at this point if you do not want to apply the optional overrides.
- When the selected Decision Service was deployed with a date range defined, it is active from the effective date through the expiration date. You can apply overrides to change the test Decision Service's version or

to simulate the Ruletest's call as occurring at a specific point in time. Specify your preferred values -- major version + effective target date -- as illustrated:



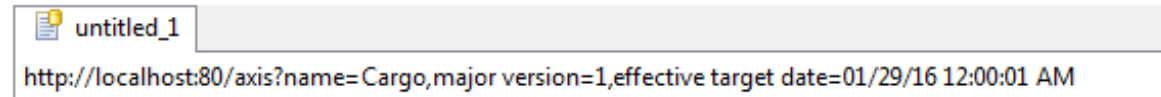
Optional Overrides

Major Version:

Minor Version:

Effective Target Date: Time: : :

10. Click **OK**. The dialog closes. The details of the remote server and Decision Service specifications are displayed at the top of the Testsheet:




untitled_1
http://localhost:80/axis?name=Cargo,major version=1,effective target date=01/29/16 12:00:01 AM

11. Run the Ruletest.

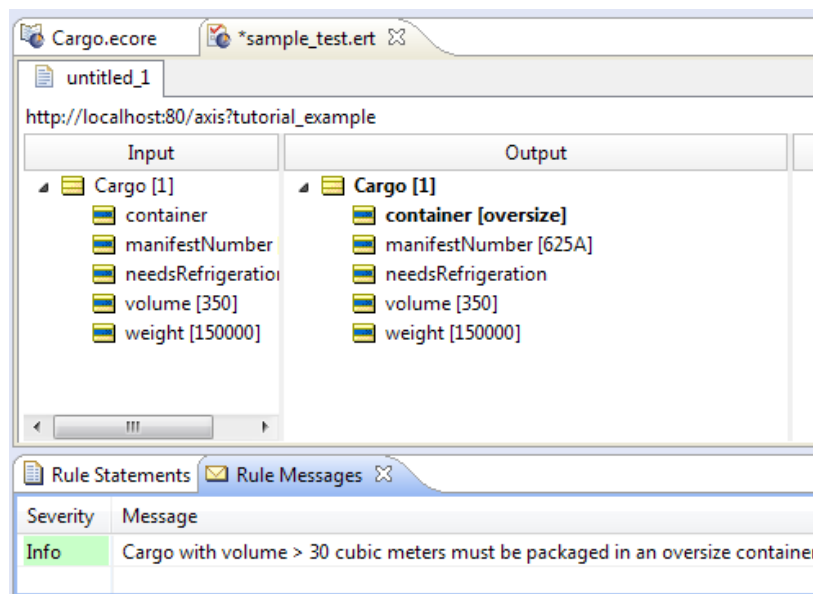
The test executes against the specified Decision Service on the selected .NET web server using the overrides you entered.

Executing the remote .NET server test

Execute the Test by selecting **Ruletest > Testsheet > Run Test** from the Corticon Studio menubar or  from the toolbar.

We should see an Output pane similar to the following:

Figure 10: Response from Remote Decision Service



The Output pane of the Testsheet shown above displays the response message returned by the Corticon Server. This confirms that our Decision Service has processed the data contained in the request and sent back a response containing new data (the `container` attribute and the message).

Path 2: Using bundled C# sample code to consume a Decision Service

To use this path, you should have solid .NET programming skills and familiarity with the .NET Framework SDK environment. This guide does not provide in-depth explanations of working within the .NET environment.

Sample web service client code is provided in `[CORTICON_HOME]Server.NET\samples\webservice-client`. This sample includes the following files:

- `Cargo_FlightPlan.wsdl` - WSDL generated by the Deployment Console
- `CargoDecisionProxy_ASPNET.cs` - C# web service proxy generated by `wsdl.exe`
- `CallCargoService.cs` - C# code demonstrating how to call the web service
- `GenProxy.bat` - Code to generate the decision service proxy from the WSDL

Creating the WSDL and proxy files

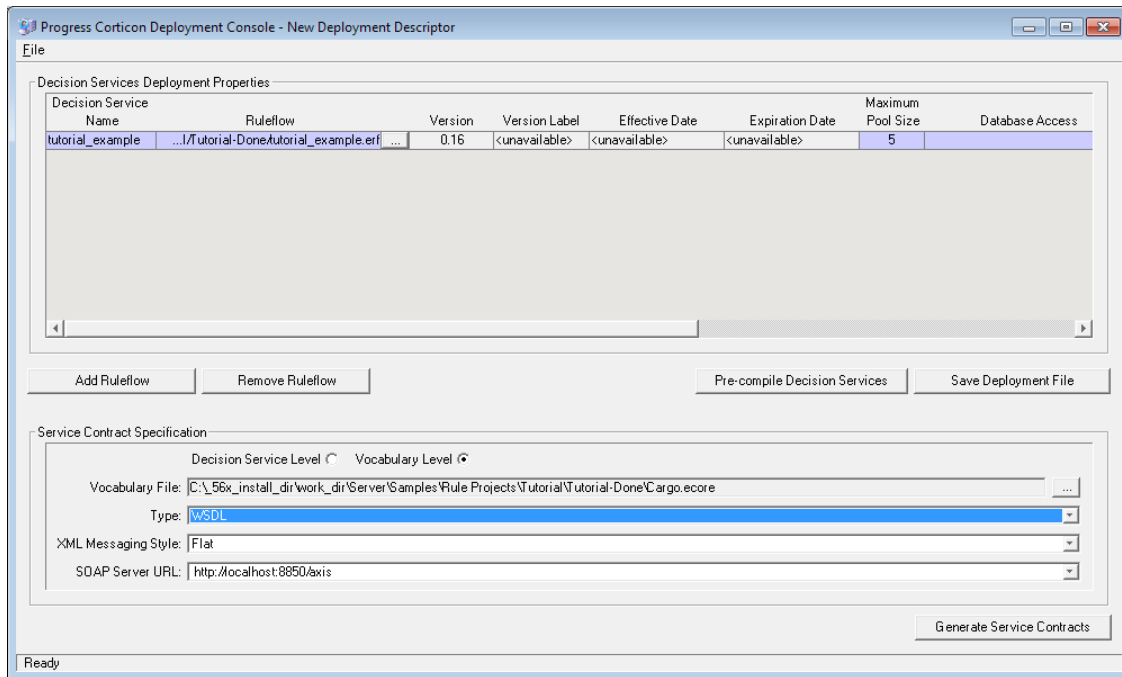
The WSDL and the proxy files are created as follows:

1. If your .NET Server and Studio are colocated, you have the Tutorial Ruleflow in the server's `[CORTICON_WORK_DIR]\Samples\Rule Projects\Tutorial\Tutorial-Done`.
If your .NET Server and Studio are on separate machines, copy and stage that file so that it can be accessed on the .NET server machine.
2. Launch the Deployment Console on the Corticon Server .NET machine by choosing the **Start** menu command **All Programs > Progress > Corticon 5.6 > Corticon .NET Deployment Console**
3. Click the ... button to the right of the Ruleflow on the one empty line listed, and locate the `tutorial_example.erf` file.

- In the lower section, click the **Type** dropdown, and then choose WSDL.

The window should now look like this:

Figure 11: Creating a new WSDL using the Deployment Console



- Click **Generate Service Contracts** to save the service contract file, which is named `Cargo_Cargo.wsdl`. It may be convenient to generate this file into a separate directory. Here, we use directory `[CORTICON_WORK_DIR]`.

Note: To generate a web service proxy, you need `wsdl.exe`. When you run `wsdl.exe Cargo_Cargo.wsdl`, the file `CargoDecisionService.cs` is created. Place that file in the .NET Server's `[CORTICON_HOME]`. Refer to the `GenProxy.bat` file located at `[CORTICON_HOME]\Server.NET\samples\wcf-client` for the WSDL options, typically `/namespace:` and `/out:.`

- Write C# client code to call the web service. We provide a sample in `CallCargoService.cs`, which sets values of attributes used in the rules.
- Compile `CargoDecisionService.cs` and `CallCargoService.cs` using the `csc *.cs` command. Generally, the compile process needs to occur in your .NET Framework root directory, so you may need to move both C# files to that directory prior to compilation. In our case, the .NET Framework is installed at `C:\WINDOWS\Microsoft.NET\Framework\v4.0.30319`

This generates an executable file named `CallCargoService-webservice.exe`. Store the file in your `[CORTICON_WORK_DIR]`.

- If you have not already done so, deploy the `tutorial_example` Decision Service to Corticon Server for .NET on IIS. Follow the instructions for **Creating and Installing a Deployment Descriptor File**.

- Run `CallCargoService-webservice.exe` to execute the call to Corticon Server. You will see the following output:

Figure 12: Invoking Corticon Server for .NET via C# Sample Code

```
Calling service tutorial_example
-----
Response
-----
workDocumentsType
Cargo weighing > 20000 kilos and a volume <= 30 cubic meters
must be packaged in a heavyweight container.
```

Path 3: Using SOAP client to consume a Decision Service

Web Services Service Contracts

Many commercial SOAP and web services development tools have the ability to import an XSD or WSDL service contract and generate a compliant request message directly from it. This path assumes you have access to such a tool and want to use it to consume a Decision Service.

The Corticon Deployment Console can produce both XSD and WSDL documents. The *Server Integration & Deployment Guide* contains much more information about these documents, including detailed descriptions of their structure and elements. However, if you have chosen this path, we assume you are already familiar enough with service contracts to be able to use them correctly once generated.

Web services messaging styles

There are also two types of messaging styles commonly used in web services:

1. RPC-style, which is a simpler, less-capable messaging style generally used to send smaller messages and receive single variable answers. Some of the administrative methods in Corticon Server's SOAP API use RPC-style messaging.
2. Document-style, which is more complex, but allows for richer content, both in request and response messages. Corticon Server for .NET's `execute` method is most commonly invoked through document-style messaging because of its ability to work with a complex data payload.

Important: Any SOAP client or SOAP-capable application used to consume a Decision Service deployed to the Corticon Server must use document-style messaging. See the *Integration & Deployment Guide* for complete details on proper structure of a compliant request message.

Creating a service contract using the Deployment Console

Launch the **Deployment Console** as before and follow the instructions below to generate a service contract. All **Deployment Console** options below are also described in more detail in the *Server Integration & Deployment Guide*.

1. **Decision Service Level / Vocabulary Level.** These radio buttons determine whether one service contract is generated per listed Ruleflow, or if a single “master” service contract is generated from the entire Vocabulary. A Decision Service-level service contract is usable only for a specific Decision Service, whereas a Vocabulary-level service contract can be used for all Decision Services that were built using that Vocabulary. Choose the option that is most compatible with your SOAP tool.
2. **Vocabulary File.** If generating a Vocabulary-level service contract, enter the Vocabulary file name (`.ecore`) here. If generating a Decision Service-level contract, this field is read-only and shows the Vocabulary associated with the currently highlighted Ruleflow row above.
3. **Type.** This is the service contract type: WSDL, XML Schema, or Java classes. Note, no output is produced when Java classes is selected because there is no standard method for describing service contracts in the Java world.
4. **Output directory.** The location where you want the Deployment Console to save this service contract.
5. **XML Messaging Style.** Enabled only for Vocabulary-level service contracts. Describes the message style, flat or hierarchical, in which the WSDL will be structured.
6. **SOAP Server URL.** URL for the SOAP node that is bound to the Corticon Server. Enabled for WSDL service contracts only. The default URLs <http://localhost:8850/axis/services/Corticon> and <https://localhost:8851/axis/services/Corticon> make a Decision Service available to the default Corticon Server installation performed earlier. Note: These URLs can be changed and additional URLs can be added to the drop-down list.
7. **Generate Service Contracts.** Use this button to generate either the WSDL or XML Schema service contracts into the output directory. If you select Decision Service-level contracts, one service contract per Ruleflow listed at top will be created. If you select Vocabulary-level, only one contract is created per Vocabulary file.

Creating a request message for a decision service

Once your SOAP development tool has imported the WSDL or XSD service contract, it should be able to generate an instance of a request message that complies with the service contract. It should also provide you with a way of entering sample data to be included in the request message when it is sent to the Decision Service.

Important:

Most commercial SOAP development tools accurately read service contracts generated by the Deployment Console, ensuring well-formed request messages are composed.

One occasional problem, however, involves the Decision Service Name, which was entered in field 3 of the Deployment Console's Deployment Descriptor section. Even though all service contracts list `decisionServiceName` as a mandatory element, many SOAP tools do not automatically insert the Decision Service Name attribute into the request message's `decisionServiceName` element. Be sure to check this before sending the request message. If the request message is sent without a `decisionServiceName`, the Corticon Server will not know which Decision Service is being requested, and will return an error message.

Enter all required data into the request message. The `tutorial_example.erf` example requires the following data:

Vocabulary Term	Possible Values
<code>Cargo.weight</code>	A number less than or equal to 200,000
<code>Cargo.volume</code>	Any real number

Sending a request message to the server

Make sure IIS is running and your Deployment Descriptor file is installed in the correct location as described earlier. Now, use your SOAP tool to send the request message to the Server for .NET.

Your SOAP tool should display the response from the Server for .NET. Are the results what you expected? If not, or if the response contains an error, proceed to the Troubleshooting section of this guide.

Path 4: Using JSON/RESTful client to consume a Decision Service on .NET server

You can create Corticon requests in JavaScript Object Notation (JSON), a text format that you can use as an alternative to XML. A JSON RESTful interface is one that follows the REST architectural style and uses JSON as its data representation format. Specifically, a standardized `JSONObject` with name-value pairs of `"Objects": <JSONArray>` can be passed in to Corticon Server's `ICcServer.execute (...)` to process the request and return a JSON-formatted reply.

Running the sample JSON Request on .NET server

A Corticon Server installation provides a JSON sample (similar to the SOAP `.xml` sample) and a test script that runs the sample.

The sample, located at `[CORTICON_WORK_DIR]\Samples\Rule Projects\OrderProcessing\OrderProcessingPayload.json`, is as follows:

```
{
  "Objects": [
    {
      "total": null,
      "myItems": [
        {
          "product": "Ball",
          "price": "10.000000",
          "quantity": "20",
          "subtotal": null,
          "__metadata": {
            "#id": "Item_id_1",
            "#type": "Item"
          }
        },
        {
          "product": "Racket",
          "price": "20.000000",
          "quantity": "1",
          "subtotal": null,
          "__metadata": {
            "#id": "Item_id_2",
            "#type": "Item"
          }
        },
        {
          "product": "Wrist Band",
          "price": "5.250000",
          "quantity": "2",
          "subtotal": null,
          "__metadata": {
            "#id": "Item_id_3",
            "#type": "Item"
          }
        }
      ]
    }
  ]
}
```

```
        "#type": "Item"
      }
    ],
    "shipped": null,
    "shippedOn": null,
    "__metadata": {
      "#id": "Order_id_1",
      "#type": "Order"
    },
    "dueDate": "1/1/2008 12:00:00 AM",
    "note": null
  }}}}
```

To run the JSON sample:

1. Start Corticon Server .NET.
2. Open a command prompt window at [CORTICON_HOME]\Server .NET\samples\bin.
3. Launch Corticon-API-Rest_Test.exe. The command transaction list is displayed:

```
-----
--- Current Apache Axis Location: http://localhost:8850/axis
-----
```

Transactions:

```
-1 - Exit REST API Test
```

```
-----
0 - Change Connection Parameters
-----
```

```
142 - Execute JSON REST request
```

```
143 - Execute JSON REST request (by specific Decision Service Major Version)
```

```
144 - Execute JSON REST request (by specific Decision Service Major and Minor Version)
```

```
145 - Execute JSON REST request (by specific execution Date)
```

```
146 - Execute JSON REST request (by specific execution Date and Decision Service Major
Version)
-----
```

```
Enter transaction number
```

4. Enter 142.
5. When prompted for **Input JSON File Path**, enter the path to the sample:

```
C:\Users\{user}\Progress\CorticonWork_5.6\Samples\Rule
Projects\OrderProcessing\OrderProcessingPayload.json
```

6. When prompted for **Input Decision Service Name**, enter (or copy) the name of the Decision Service that is the sample's target:

```
ProcessOrder
```

The request is processed, and its output is placed at [CORTICON_WORK_DIR]\output with a name formatted as OutputCRString_{epochTime}.json where {epochTime} is the number of seconds that have elapsed since 1/1/1970. The input file is also placed there. The output for the sample is as follows:

```
{
  "Messages": {
    "Message": [
      {
        "entityReference": "Item_id_3",
        "text": "The subtotal of line item for Wrist Band is 10.500000.",
        "severity": "Info",
```

```

    "__metadata": {"#type": "#RuleMessage"}
  },
  {
    "entityReference": "Item_id_2",
    "text": "The subtotal of line item for Racket is 20.000000.",
    "severity": "Info",
    "__metadata": {"#type": "#RuleMessage"}
  },
  {
    "entityReference": "Item_id_1",
    "text": "The subtotal of line item for Ball is 200.000000.",
    "severity": "Info",
    "__metadata": {"#type": "#RuleMessage"}
  },
  {
    "entityReference": "Order_id_1",
    "text": "The total for the Order is 230.500000.",
    "severity": "Info",
    "__metadata": {"#type": "#RuleMessage"}
  },
  {
    "entityReference": "Order_id_1",
    "text": "This Order was shipped late. Ship date 12/1/2008 12:00:00 AM",
    "severity": "Warning",
    "__metadata": {"#type": "#RuleMessage"}
  }
],
 "__metadata": {"#type": "#RuleMessages"},
 "version": "0.0"
},
"Objects": [{
  "total": 230.5,
  "myItems": [
    {
      "product": "Ball",
      "price": "10.000000",
      "quantity": "20",
      "subtotal": 200,
      "__metadata": {
        "#id": "Item_id_1",
        "#type": "Item"
      }
    },
    {
      "product": "Racket",
      "price": "20.000000",
      "quantity": "1",
      "subtotal": 20,
      "__metadata": {
        "#id": "Item_id_2",
        "#type": "Item"
      }
    },
    {
      "product": "Wrist Band",
      "price": "5.250000",
      "quantity": "2",
      "subtotal": 10.5,
      "__metadata": {
        "#id": "Item_id_3",
        "#type": "Item"
      }
    }
  ]
},
 "shipped": true,
 "shippedOn": "12/1/2008 12:00:00 AM",
 "__metadata": {
   "#id": "Order_id_1",

```

```
        "#type": "Order"
    },
    "dueDate": "1/1/2008 12:00:00 AM",
    "note": "This Order was shipped late"
  }
}
```

Path 5: Using bundled JSON sample code to consume a Decision Service

Sample C# code that you can use to create a C# project, or create your own sample client to execute a JSON request using the REST API supported by Corticon .NET.

Note: To use this path, you should have solid .NET and C# programming skill, as well as familiarity with the .NET Framework SDK environment.

This sample includes the following files:

- `CorticonServerRestTest.cs` - Commented C# code to help you create a .NET REST client to execute against the REST Service in Corticon Server for .NET running in IIS 7.5.
- `Json60` - Reference DLL for JSON used in `CorticonserverRestTest.cs` for .Net framework.
- `app.config` - Application configuration file to load assembly files for a C# project.

Limits of the .NET server default evaluation license

The license included in the default Corticon Server for .NET installation has preset limits on some Corticon Server and Decision Service parameters. These limits are:

- **Number of Decision Services** – Up to five Decision Services may be deployed at any given time. This means the sum total of all Decision Services loaded via `.cdd` files, Web Console, or APIs cannot exceed five.
- **Pool Size** – No Decision Service may have a maximum pool size setting of greater than five. Pool size is measured on a Decision Service-by-Decision Service basis, so you may have up to 5 Decision Services deployed (compliant with the Decision Service limitation above), each with up to five Reactors in its pool, without violating the default license restrictions.
- **Number of Rules** – All rules in all deployed Ruleflows (in other words, all deployed Decision Services) must not exceed 200. A rule generally consists of a single Condition/Action Column or a single Action row in Column 0. Filter expressions do not count because they only modify other rules.

The Corticon Server log can capture errors and exceptions caused by expired or “under-strength” licenses. These log messages are detailed in the *Using Corticon Server logs section of the Integration and Deployment Guide*.

If you are a Progress Corticon customer, you should have access to an unlimited license that will lift these restrictions. If you are an evaluator, and discover that these limitations are preventing or inhibiting your evaluation, contact Progress Corticon support or your Progress representative for a license with expanded capabilities.

Troubleshooting .NET server

When you have problems on the .NET Server, refer to *"Using Corticon Server logs"* and *"Troubleshooting"* topics in the *Integration and Deployment Guide*.

Using .NET Business Objects as payload for Decision Services

Introduction

Microsoft .NET Classes provide applications with reusable, portable code. Classes are logical sections of an application. For instance, the call to a database and retrieval of table data is part of a data class. These classes can be used in other sections of the application, or can be used it in an entirely different software design.

Class properties that allow other areas of code to interface with the class are usually created with "get" and "set" keywords.

Corticon Server for .NET can communicate with .NET classes through a Java Object Messaging (JOM) interface. This document illustrates how a client program can instantiate a .NET class and then use that object instance to communicate with Corticon Server for .NET. In other words, this enables Corticon Server for .NET to natively bind to the application's .NET class definitions during rules execution.

To use the JOM interface from any C# client application, it is necessary to create Java stubs that match the C# object model. These stubs will enable Corticon Studio Vocabulary Editor to import the class metadata for mapping purposes; in other words, to pull in the names of the `get` and `set` methods needed. The stubs are also used to generate import statements in the compiled rules.

At deployment time, the Corticon Server for .NET will dynamically transform the Rule asset from Java byte code into .NET Common Intermediate Language (CIL), the object-oriented assembly language for .NET runtime environments. When this is done, the Java code stub references inside the stub library (a `.jar` file) are translated into references to the actual C# class file library (a `.dll` file). Finally, the translated Rule asset (CIL) will directly call get/set methods on the C# object during Rule asset execution.

IKVM.NET

To create Java Stubs from .NET class files, Corticon supplies an open source utility, IKVM.NET, which is an essential component of the Corticon Server for .NET runtime architecture enabling it to seamlessly operate in .NET environments. IKVM.NET is included in your Corticon Server for .NET installation.

Note: For more information about IKVM.NET, see their web site, <http://www.ikvm.net>.

IKVM.NET is an implementation of Java for Mono and the Microsoft .NET Framework. It includes the following components:

- A Java Virtual Machine implemented in .NET
- A .NET implementation of the Java class libraries
- Tools that enable Java and .NET interoperability

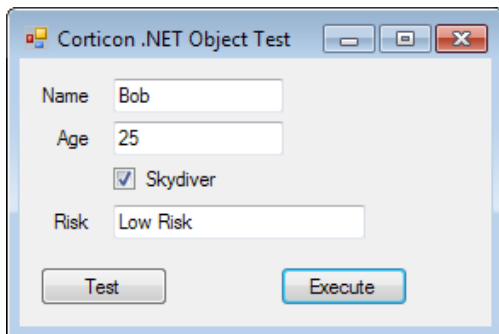
Running the sample Java Object Messaging (JOM) client application

The sample JOM Client uses C# objects to communicate with Corticon Server for .NET (in- process) via the object messaging interface.

Run this sample program by launching [CORTICON_HOME]\Server .NET\samples\bin\JomClient.exe)

When the application launches, the window lets you enter parameter values for executing this decision service:

- Name
- Age
- Is this person a skydiver?



Enter values, and then click **Execute** to see the decision.

You might wonder how this application was created and how it works. Let's examine its building blocks. When you understand the solution architecture, it will be easy to create your own examples and solutions.

Examining the sample

The sample `JomClient.exe` uses a Rulesheet. Open [CORTICON_NET_WORK_DIR]\Samples\Rule Projects\JOM\Rules\jom.RiskRating.ers:

The screenshot shows the JOM editor with two tabs: `jom_RiskRating.ers` and `jom.ecore`. The `jom_RiskRating.ers` tab displays a table of conditions and actions.

Conditions	0	1	2	3
a Applicant.isSkydiver		T	-	F
b Applicant.age		-	< 35	>= 35
c				
d				

Actions			
Post Message(s)		✉	✉
A Applicant.riskRating		'High Risk'	'Low Risk'
B			'Medium Risk'
C			

Below the actions table is an 'Overrides' section.

The `jom.ecore` tab shows 'Rule Statements' and 'Rule Messages'.

Ref	ID	Post	Alias	Text
1		Info	Applicant	Applicants who skydive are High Risk
2		Info	Applicant	Applicants under 35 are Low Risk
3		Info	Applicant	Applicants 35 and over who don't skydive are Medium Risk

Open the Vocabulary [CORTICON_NET_WORK_DIR]\Samples\Rule Projects\JOM\Vocab\jom.ecore. When your vocabulary view is set to show details, the Applicant Entity shows that it is bound to the Java Package `cli.CorticonLibrary`, and its Java Class name refers to the `C_Applicant` class.

The screenshot shows the JOM editor with two tabs: `jom_RiskRating.ers` and `jom.ecore`. The `jom.ecore` tab displays the details of the `Applicant` entity.

Property Name	Property Value
Entity Name	Applicant
Entity Identity	
Inherits From	
XML Namespace	
XML Element Name	Applicant
Java Package	cli.CorticonLibrary
Java Class Name	C_Applicant
Datastore Persistent	Yes
Table Name	
Datastore Caching	No

Each Attribute within the Entity refers to a **Java Object Get Method** and a **Java Object Set Method** method, as shown:

The screenshot shows the JOM editor with two tabs: `jom_RiskRating.ers` and `jom.ecore`. The `jom.ecore` tab displays the details of the `age` attribute.

Property Name	Property Value
Attribute Name	age
Data Type	Integer
Mandatory	No
Mode	Base
XML Namespace	
XML Element Name	age
Java Object Get Method	get_IntAge
Java Object Set Method	set_IntAge
Java Object Field Name	
Column Name	

So how do we get the `CorticonLibrary` that contains these Java classes exposed in the Corticon Vocabulary editor? Let's see.

Invoking Corticon Server for .NET

`JomClient.exe` is written in C#. Logic that invokes the in-process server is located for the sample at [CORTICON_NET_HOME]\samples\api-client\jom\JomClientForm.cs, as shown:

```
private void btnExecute_Click(object sender, EventArgs e)
{
    C_Applicant lC_Applicant = new C_Applicant();
    lC_Applicant.StrName = txtName.Text;
    lC_Applicant.IntAge = Convert.ToInt32(txtAge.Text);
    lC_Applicant.BlnSkydiver = chkSkydiver.Checked;

    ArrayList llistObjects = new ArrayList();
    llistObjects.add(lC_Applicant);
    iICcServer.execute(DECISION_SERVICE_NAME, llistObjects);
    C_Applicant lC_ApplicantReturned = (C_Applicant)llistObjects.get(0);

    txtRating.Text = lC_ApplicantReturned.StrRiskRating;
}
```

Running the sample JOM application

Now when you launch `JOMClient.exe`, Corticon Server for .NET will try to deploy the JOM Decision Service. As described above, the JOM .ecore has already imported the Java Stub Class Metadata. That's important for the next step. The JOM Client will first compile the `[CORTICON_NET_WORK_DIR]\samples\Rule Projects\JOM\jom.erf` into a deployable .eds file. For the compilation to be successful, the `CorticonLibrary.jar` must be in the `\lib` directory. During deployment of the .eds file, the IKVM loader will convert the Java byte code into CLI and also load the C# class definitions in `CorticonLibrary.dll`, effectively allowing `JomClient.exe` to use C# object instances to communicate with Corticon Server for .NET.

Preparing the C# Class files

We want the client program to instantiate `C_Applicant`, and then use that object instance to communicate with Corticon Server for .NET. First we'll need a class file we are using in our application to expose all data objects. For that, we will use `[CORTICON_NET_HOME]\samples\api-client\jom\CorticonLibrary.cs`, as shown:

```
using System;
namespace CorticonLibrary
{
    public class C_Applicant
    {
        private String strName;
        private Boolean isMarried;
        private Int32 intAge;
        private Boolean blnSkydiver;
        private String strRiskRating;
        public String strState;

        public String StrName
        {
            get { return strName; }
            set { strName = value; }
        }
        public Int32 IntAge
        {
            get { return intAge; }
            set { intAge = value; }
        }
        public Boolean BlnSkydiver
        {
            get { return blnSkydiver; }
            set { blnSkydiver = value; }
        }
        public String StrRiskRating
        {
            get { return strRiskRating; }
            set { strRiskRating = value; }
        }
        public Boolean IsMarried
        {
            get { return isMarried; }
            set { isMarried = value; }
        }
    }
}
```

Note: Having a namespace in the `CorticonLibrary.cs` file is mandatory.

Compile the `CorticonLibrary.cs` to create the `CorticonLibrary.dll`. To use the JOM interface from any C# client, it is necessary to create Java stubs that match your C# object model. For the JOM Client example, we used Visual Studio to compile the C# class `C_Applicant` into `CorticonLibrary.dll`.

Save `CorticonLibrary.dll` to both `[CORTICON_NET_HOME]\samples\gen stubs\` and `[CORTICON_NET_HOME]\samples\bin\`

Generating the Java Stubs

We can generate the Java stub file from `CorticonLibrary.dll` using the utility script `[CORTICON_NET_HOME]\samples\gen stubs\GenStubs.bat`:

```
*****
@echo off
SET IKVM_HOME=..\bin
SET INPUT_DLL_NAME=CorticonLibrary.dll
SET OUTPUT_JAR_NAME=CorticonLibrary.jar
ATTRIB -R %OUTPUT_JAR_NAME%
"%IKVM_HOME%\ikvmstub.exe" %INPUT_DLL_NAME%
ECHO Successfully completed: C# types in %INPUT_DLL_NAME% were converted into stubs in
%OUTPUT_JAR_NAME%.
Pause
*****
```

The `GenStubs.bat` utility will generate the JAR `CorticonLibrary.jar`.

Add `CorticonLibrary.dll` to `[CORTICON_NET_HOME]\samples\bin\`

Add `CorticonLibrary.jar` to `[CORTICON_NET_HOME]\samples\lib\`.

Note:

About putting the `CorticonLibrary.jar` in the `samples\lib` directory - It is important to understand where to locate the JAR file, especially when you are outside of Corticon installation directories. The JAR that the Corticon .NET Configuration (inside `CorticonShared.dll`) directs IKVM to load all JAR files that are located in the `[CORTICON_NET_HOME]\lib` directory. In the case of the `JomClient.exe`, the `JomClient.exe` defines `[CORTICON_NET_HOME]` equal to `[CORTICON_NET_HOME]\samples`. Where is this done? Each .NET Application has a `.exe.config` file where the user defines their Corticon Home and Work directories. For example `bin\JomClient.exe` has a `JomClient.exe.config`. Within that file is the following section:

```
<configuration>
  <appSettings>
    <add key="CORTICON_HOME" value=".." />
    <add key="CORTICON_WORK_DIR"
      value="C:\Users\{user}\Progress\CorticonWork_5.6 .Net"/>
  </appSettings>
</configuration>
```

That defines `CORTICON_HOME` as located up one directory level from where the `.exe` is currently located. Typically, the result is `C:\Program Files\Progress\Corticon 5.6 .Net\Server .NET\samples`. Then the `CorticonConfiguration` changes `CORTICON_HOME` by adding `"\lib"` which locates it in that directory where it loads all the JARs.

Supported .NET datatypes for Attributes

Corticon Attributes support .NET Business Objects with nullable datatypes. Using a trailing ? character converts the Datatype into a Nullable Datatype of the same type.

```
Boolean
bool
Boolean?
bool?

Byte
byte
Byte?
byte?

Char
char
Char?
char?

DateTime
DateTime?

Decimal
decimal
Decimal?
decimal?

Double
double
Double?
double?

float
float?

Int16
Int16?

Int32
Int32?

Int64
Int64?

int
int?

long
long?

short
short?

String
String?
```

Supported .NET datatypes for Associations

Corticon Associations support .NET Business Objects.

C# Association Datatypes:

```
System.Collections.ArrayList
System.Collections.IList
<Business Object Name>[] (Array of Business Objects)
```

IKVM Open JDK Datatypes:

```
java.util.ArrayList  
java.util.Vector  
java.util.HashSet  
java.util.Collection  
java.util.List  
java.util.Set
```

All datatypes that are supported are demonstrated in the `CorticonLibrary.cs` file under the `ObjectA` and `ObjectB` objects.

Using GenStubs.bat for your .NET Business Objects

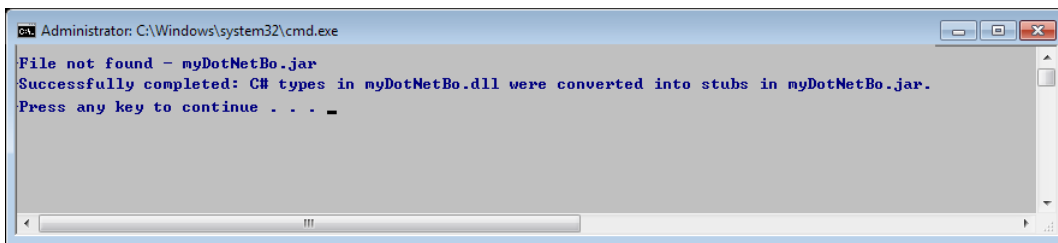
To generate stubs:

1. Compile the .NET Business Objects in `.cs` or `.vb` to create a `.dll` file. For example, use `myDotNetBo.cs` or `myDotNetBo.vb` to compile `myDotNetBo.dll`.
2. Copy the .NET Business Object `.dll` to the `[CORTICON_NET_HOME]\samples\gen stubs\` directory.
3. Locate a text editor at `[CORTICON_NET_HOME]\samples\gen stubs\` to edit `GenStubs.bat`.
4. Modify the following two properties to match the `.dll` name and the `.jar` name, as shown here for `myDotNetBo`:

```
SET INPUT_DLL_NAME= myDotNetBo.dll  
SET OUTPUT_JAR_NAME= myDotNetBo.jar
```

5. Save the file under an appropriate name (such as `GenStubs_MyBo.bat`).
6. Run your `GenStubs` script file to generate the Java stub JAR file.

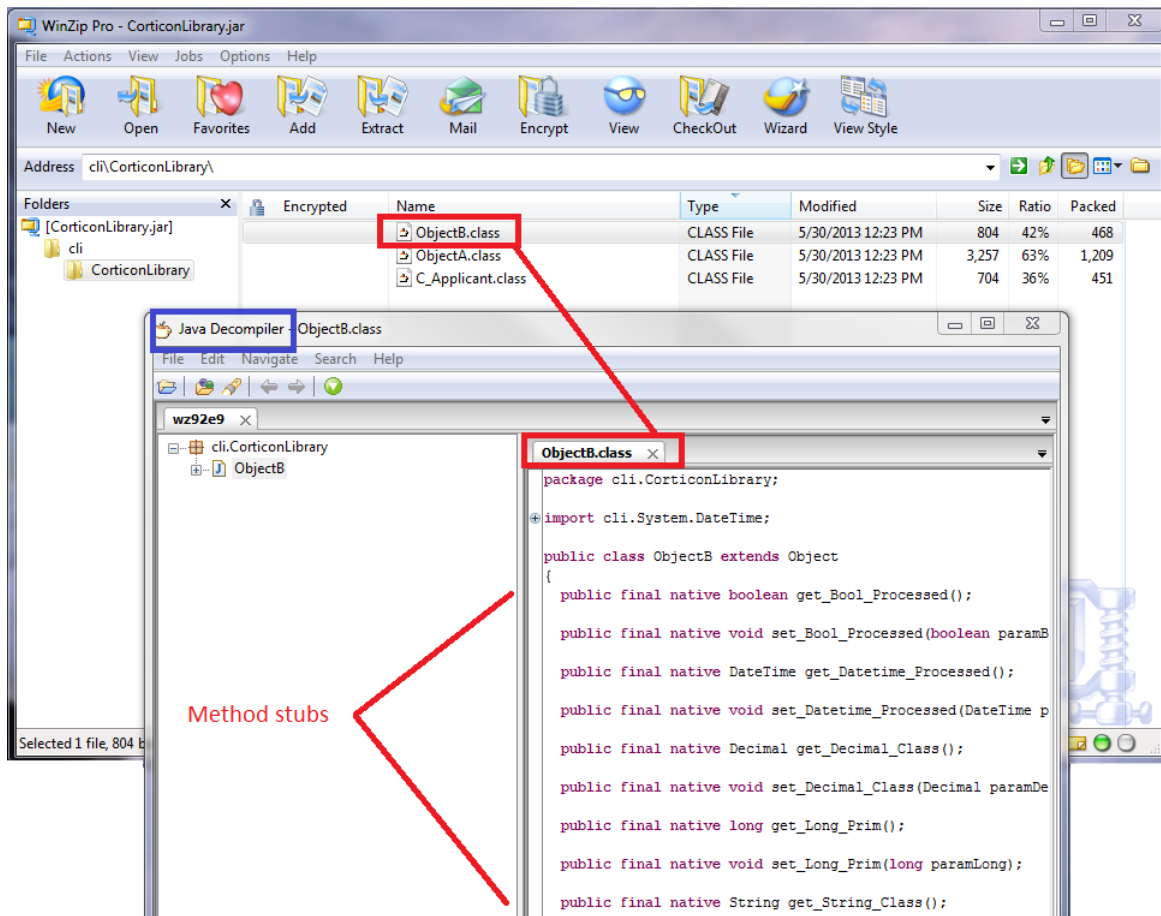
Note: The first time you do this, you get a warning message about not finding `myDotNetBo.jar`:



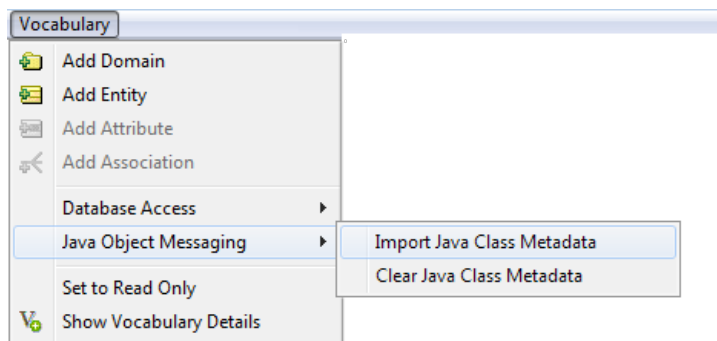
Ignore this -- it is trying to delete it first, but it isn't there.

The Java stub JAR file you created is saved at `[CORTICON_NET_HOME]\samples\gen stubs\`.

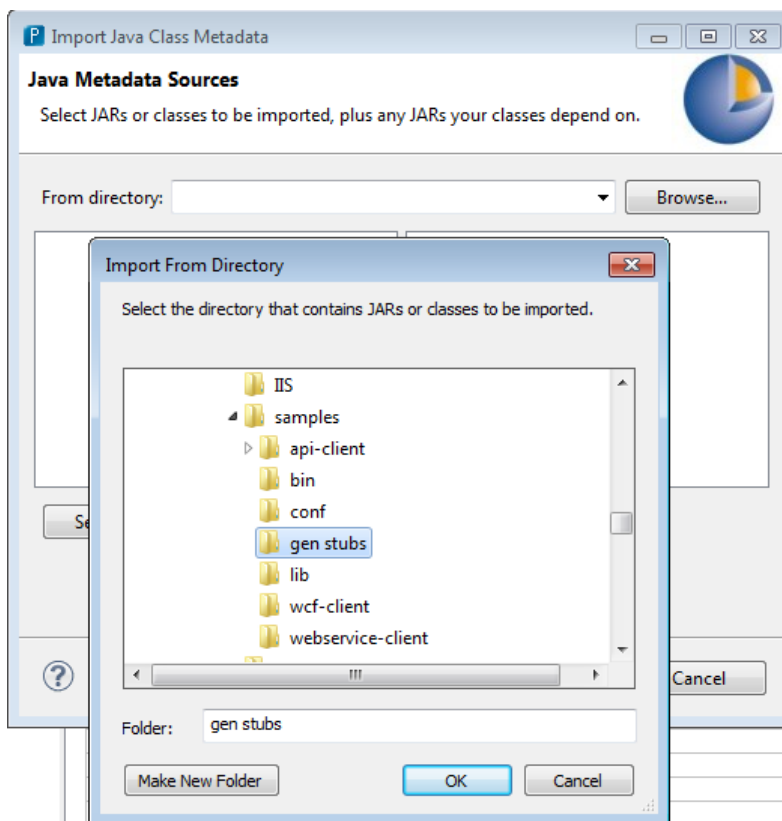
7. Copy your JAR (for example, `myDotNetBo.jar`) to `[CORTICON_NET_HOME]\samples\lib`.
8. Verify the creation of the JAR file. The following example illustrates a stub class in the `CorticonLibrary.jar` through a Java Decompiler:



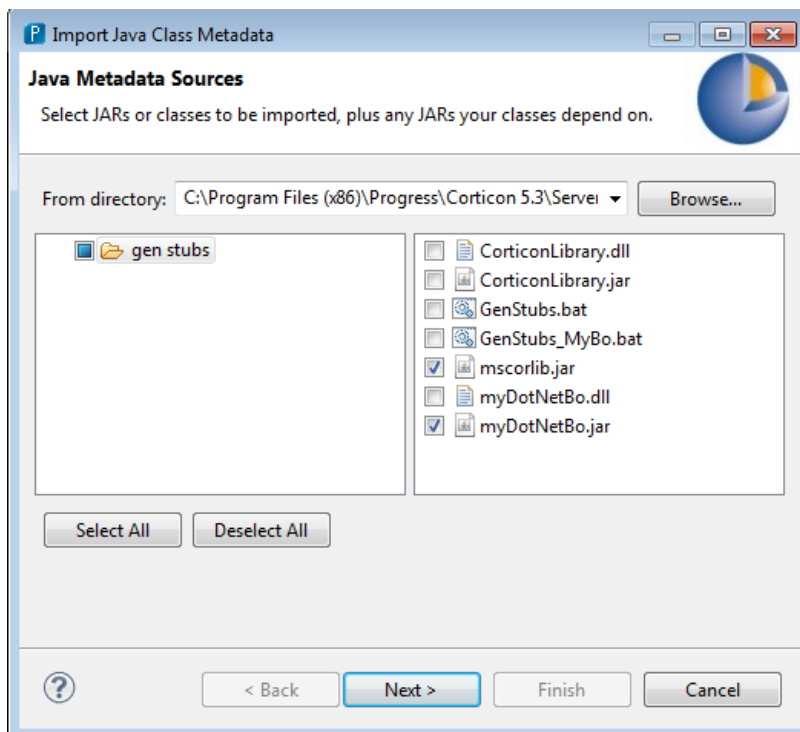
- Open the Vocabulary in Corticon Studio, and then chose the menu command **Vocabulary > Java Object Messaging > Import Java Class Metadata**, as shown:



- Browse to [CORTICON_NET_HOME]\samples\gen stubs\:

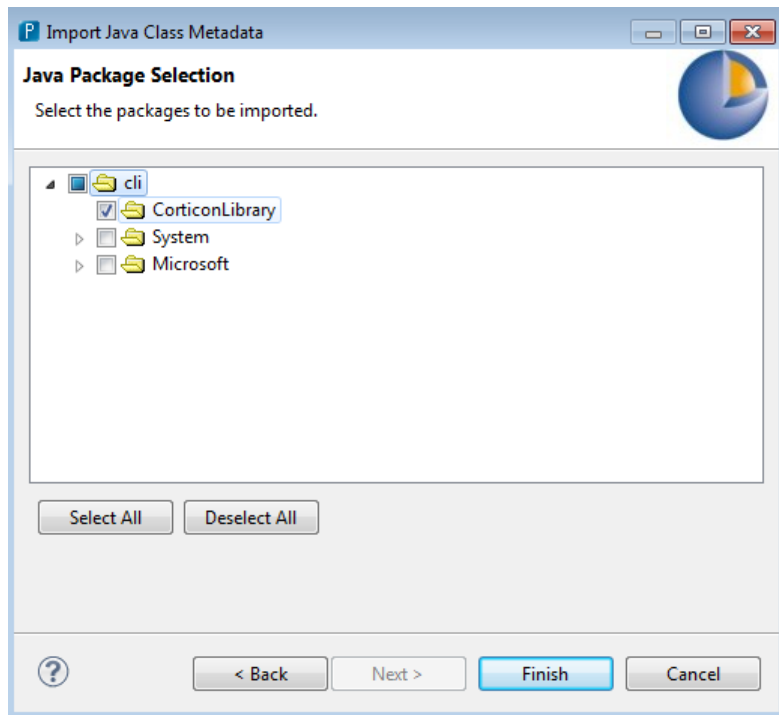


11. Select the JAR that you want imported into Corticon Studio, as well as `microsoft.jar` that is also in the `gen stubs` directory. The `microsoft.jar` needs to also be selected because the `.jar` file that was created through the `gen stubs` process depends on the classes inside the `microsoft.jar`.

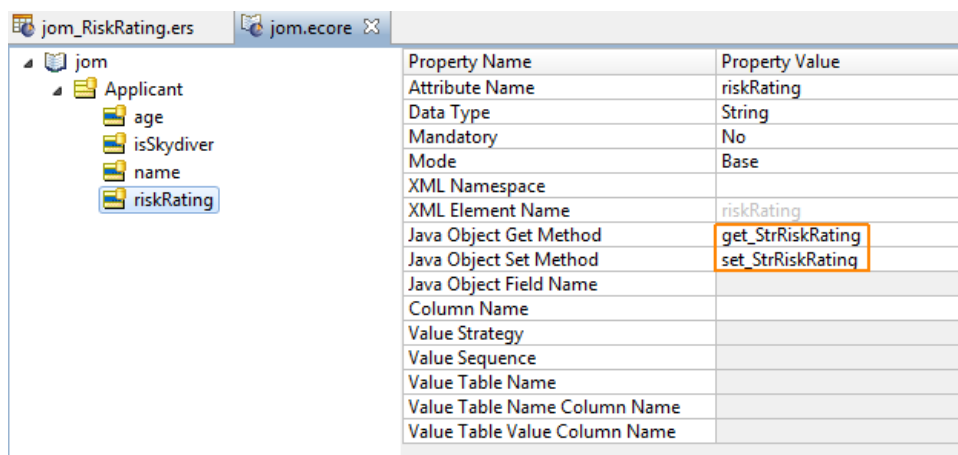


12. Select the packages that are associated with the Java Stub Classes that were created in the `gen stubs` process. In our example, the `CorticonLibrary.jar` contains classes under the `cli.CorticonLibrary` package.

Note: As noted earlier, Business Objects require a defined Namespace. If there is no defined Namespace, the Java Stub Classes will have a default package of only `cli`. That would force you to choose the `cli` checkbox, and import **ALL** the Java Class Metadata in the `CorticonLibrary.jar` and the `microsoftlib.jar`, an unwarranted scope.



13. After import is complete, verify that the **Java Object Getter/Setter Methods** have been assigned to each of the Attributes and Associations.



The Vocabulary tries to **SmartMatch** the Vocabulary Attribute and Association Names to match a Getter/Setter name in the imported Java Metadata. The **SmartMatch** looks for an appropriate Getter/Setter Method that begins with `get` or `set` as in `get<AttributeName>` or `get_<AttributeName>`. If this occurs, the

Getter/Setter value inside the Vocabulary is colored light grey, indicating that it was determined through a **SmartMatch** lookup.

However, in the example above, **SmartMatch** is unable to match the Getter/Setter with Attribute `riskRating` because the appropriate Java Class Method does not conform to the **SmartMatch** algorithm:

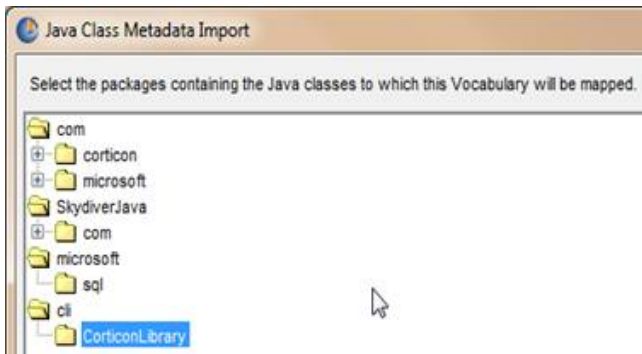
- Vocabulary Attribute: `riskRating`
- Smart Match looks for: `getRiskRating()` or `get_RiskRating()`
- Actual Java Method: `get_StrRiskRating()`

14. Since the **SmartMatch** failed to locate an appropriate Getter/Setter, you need to select the appropriate Getter/Setter Method from the drop-down for that Attribute or Association. Since the user defined their own mappings, the Getter/Setter method value is colored black (as illustrated).

15. After deploying the rules to IIS server, ensure that the Business Objects are picked up by copying:

- `myDotNetBo.jar` to `C:\inetpub\wwwroot\axis\lib`
- `myDotNetBo.dll` to `C:\inetpub\wwwroot\axis\bin`

16. When the package has been created, it will look something like this:



Testing outside IIS

If you are testing your work outside of IIS, such as in a Visual Studio Environment, you to perform some extra tasks:

1. Copy the `myDotNetBO.dll` to a location where it can be referenced. If running in IIS and the `axis` application has been deployed, the `.dll` needs to be moved to the `c:\inetpub\wwwroot\axis\bin` directory.
2. Copy the `myDotNetBo.jar` into the running application's `[CORTICON_NET_HOME]\lib` directory.

The JAR is needed during compilation of the Ruleflow (`.erf`) into an Enterprise Decision Service (`.eds`) file. The compilation step depends on the `.jar` file being in the `[CORTICON_NET_HOME]\lib` directory.

Note: About putting the JAR in the `\lib` directory when Business Objects are used in IIS

The `CorticonConfiguration` looks for `CORTICON_HOME\lib`. With IIS, `CORTICON_HOME` *could be* defined or overridden in the `web.config` file under `[IIS]\axis`.

Parameters can be set in Corticon in the `<appSettings>` section. Note there are no `CORTICON_HOME` or `CORTICON_WORK_DIR` settings. That means that the “current directory” is the `CORTICON_HOME` and `CORTICON_WORK_DIR`. For IIS, put the JAR file in `C:\inetpub\wwwroot\axis\lib`.

Compiling a Decision Service into an Assembly DLL

Compiling a Decision Service's `.eds` file into an `Assembly.dll` file provides a marked improvement in performance over IKVM dynamically converting `.class` files during deployment by compiling the `.class` files inside the `.eds` file into an `Assembly.dll` that contains Corticon Data Objects (CDOs), listeners and Rules. The resulting `Assembly.dll` is added to the `.eds` file. When the `.eds` file is deployed, the Corticon Server looks for an `Assembly.dll`, and when it finds one, it gets the information it needs from the `Assembly.dll` instead of from the `.class` files inside the `.eds` file.

Creating an Assembly.dll

A Corticon Server for .NET installation includes a script, `CompileAssembly.bat`, located in the .NET Server's home directory that creates an `Assembly.dll`. The script's inputs require the location of the `.eds` file you want to compile, and -- if you have Business Objects -- the name of the reference `.jar` to create for the Business Objects.

To compile a Decision Service into an `Assembly.dll`:

1. Generate an `.eds` file. (You could use Transaction Id; 150/151 to Precompile a RuleFlow into an `.eds` file.)
2. If you have Business Objects, you need to generate stubs (as described in [Using .NET Business Objects as payload for Decision Services](#) on page 43) into a `.jar` file that will be incorporated into your `.dll`.
3. Edit the script `CompileAssembly.bat` located at `[CORTICON_HOME]\samples\compile assembly` as follows:
 - `SET INPUT_EDS_FILE_NAME=<path of the input .eds file>`
 - `SET REFERENCE_JAR_FILE_NAME=<path of the reference .jar created by GenStubs>`
(or empty)

4. Save the edited file.
5. Open a Command Prompt to the location `[CORTICON_HOME]\samples\compile assembly`
6. Run the `CompileAssembly` script.

When the script successfully completes, the `.eds` file has embedded its `Assembly.dll` .

Support for Windows Communication Framework (WCF)

To use this approach, you should have solid .NET programming skills and familiarity with the .NET WCF Framework. This guide does not provide in-depth explanations of working within the .NET WCF environment.

Sample web service client code is provided in `[CORTICON_HOME]Server .NET\samples\wcf-client`. This sample includes the following files:

- `Cargo_FlightPlan.wsdl` - WSDL generated by the Deployment Console
- `CargoDecisionProxy_WCF.cs` - C# web service proxy generated by `svcutil.exe`
- `CallCargoService.cs` - C# code demonstrating how to call the web service
- `GenProxy.bat` - Code to generate the decision service proxy from the WSDL
- `App.config` - The configuration file for the decision service endpoint

For details, see the following topics:

- [Creating WSDL and proxy files](#)

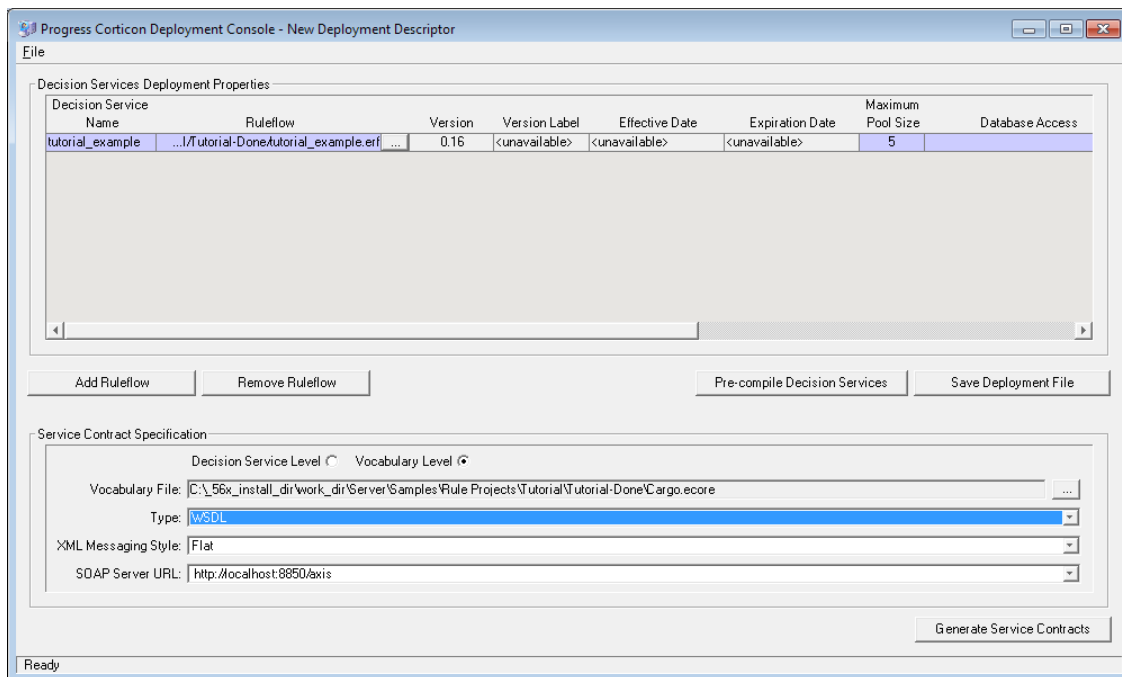
Creating WSDL and proxy files

The WSDL and the proxy files are created as follows:

1. If your .NET Server and Studio are colocated, you have the Tutorial Ruleflow in the server's [CORTICON_WORK_DIR]\Samples\Rule Projects\Tutorial\Tutorial-Done.
If your .NET Server and Studio are on separate machines, copy and stage that file so that it can be accessed on the .NET server machine.
2. Launch the Deployment Console on the Corticon Server .NET machine by choosing the **Start** menu command **All Programs > Progress > Corticon 5.6 > Corticon .NET Deployment Console**
3. Click the ... button to the right of the Ruleflow on the one empty line listed, and then locate the tutorial_example.erf file.
4. In the lower section, click the **Type** dropdown, and then choose WSDL.

The window should now look like this:

Figure 13: Creating a new WSDL using the Deployment Console



5. Click **Generate Service Contracts** to save the service contract file, which is named Cargo_Cargo.wsdl. It may be convenient to generate this file into a separate directory. Here, we use directory [CORTICON_WORK_DIR].

Note: To generate a web service proxy, you need `wsdl.exe`. When you run `wsdl.exe Cargo_Cargo.wsdl`, the file `CargoDecisionService.cs` is created. Place that file in the .NET Server's [CORTICON_HOME]. Refer to the `GenProxy.bat` file located at [CORTICON_HOME]\Server .NET\samples\wcf-client for the WSDL options, typically `/namespace:` and `/out:.`

6. Write C# client code to call the web service. We provide a sample in `CallCargoService.cs`, which sets values of attributes used in the rules.

7. Compile `CargoDecisionService.cs` and `CallCargoService.cs` using the `csc *.cs` command. Generally, the compile process needs to occur in your .NET Framework root directory, so you may need to move both C# files to that directory prior to compilation. In our case, the .NET Framework is installed at `C:\WINDOWS\Microsoft.NET\Framework\v4.0.30319`

This generates an executable file named `CallCargoService-webservice.exe`. Store the file in your `[CORTICON_WORK_DIR]`.

8. If you have not already done so, deploy the `tutorial_example` Decision Service to Corticon Server for .NET on IIS. Follow the instructions for Creating and Installing a Deployment Descriptor File.
9. Run `CallCargoService-webservice.exe` to execute the call to Corticon Server. You will see the following output:

Figure 14: Invoking Corticon Server for .NET via C# Sample Code

```
Calling service tutorial_example
-----
Response
-----
workDocumentsType
Cargo weighing > 20000 kilos and a volume <= 30 cubic meters
must be packaged in a heavyweight container.
```


Updating your Corticon license JAR for .NET

Progress Corticon embeds an evaluation license in its products to help you get started.

- Corticon Studio evaluation licenses let you use database access (Enterprise Data Connector or "EDC"), and are timed to expire on a preset date.
- Corticon Server evaluation licenses do not enable use of Enterprise Data Connector, and limit the number of decision services, rules, and pools in use. They too are timed to expire on a preset date.

When you obtain a license file, it applies to Studios as well as Servers. You must perform configuration tasks to record it for each Corticon Studio, each Corticon Server, and each Deployment Console. If you intend to use EDC on your Corticon Servers, your Corticon license must allow it. Contact Progress Corticon technical support if you need to acquire a license.

The Corticon Server license is placed at two locations in the installation to enable use and -- when specified in the license -- enable EDC functions for:

- Corticon Server
- Corticon Deployment Console

To configure Corticon Server for .NET to access its license file:

1. Copy the license JAR with its default name, `CcLicense.jar`.
2. Depending on your server strategy, the JAR must be placed in its required locations:
 - For an in-process deployment:
 1. Navigate to the .NET Server installation's `[CORTICON_HOME]\samples\lib` directory to paste the file and overwrite the existing file in that location.
 2. Navigate to the .NET Server installation's `[CORTICON_HOME]\webservice\lib` directory to paste the file and overwrite the existing file in that location.

- For an IIS deployment:
 - Navigate to the IIS installation location, typically `C:\inetpub` and then navigate to its `\wwwroot\axis\lib` directory to paste the file and, if necessary, overwrite the existing file in that location.

When you launch the Corticon Deployment Console, your license with its registration information is registered for the Corticon Deployment Console. When your license enables EDC, the **Database Access** fields and functions are enabled.

Note:

You can choose to locate the license by another JAR name at a preferred location, and then expressly identify it to the server.

To custom configure Corticon Server for .NET's license location:

1. Navigate in the file system to the Corticon Server for .NET installation's, `[CORTICON_HOME]\Server.NET\samples\bin` subdirectory.
 2. Double-click on `Corticon-API-Inprocess-Test.exe`, then do the following:
 - a. Type `416` and then press **Enter**.
 - b. Enter (or copy/paste) the complete path to the location of the license JAR file, as in this example, `C:\licenses\myCorticonEDC_CcLicense.jar`. The command echoes back `Transaction completed`.
 - c. To confirm the setting, type `415` and then press **Enter**. The path is echoed back (you might need to scroll up to the command line.)
 3. Once the .NET Server is running in remote mode, you can double-click on `Corticon-API-Remote-Test.exe` in that same directory to perform the same `416` and `415` tasks as in Step 2 above
-

Access to Corticon knowledge resources

[Complete online documentation for the current release](#)

Corticon online tutorials available in the [Corticon Learning Center](#):

- [Tutorial: Basic Rule Modeling in Corticon Studio](#)
- [Tutorial: Advanced Rule Modeling in Corticon Studio](#)
- [Modeling Progress Corticon Rules to Access a Database using EDC](#)
- [Connecting a Progress Corticon Decision Service to a Database using EDC](#)
- [Deploying a Progress Corticon Decision Service as a Web Service for Java](#)
- [Deploying a Progress Corticon Decision Service in process for Java](#)
- [Using Corticon Business Rules in a Progress OpenEdge Application](#)

Corticon guides (PDF):

- [What's New in Corticon](#)
- [Corticon Installation Guide](#)
- [Corticon Studio: Rule Modeling Guide](#)
- [Corticon Studio: Quick Reference Guide](#)
- [Corticon Studio: Rule Language Guide](#)
- [Corticon Studio: Extensions Guide](#)
- [Corticon Server: Integration and Deployment Guide](#)

- [Corticon Server: Web Console Guide](#)
- [Corticon Server: Deploying Web Services with Java](#)
- [Corticon Server: Deploying Web Services with .NET](#)

Corticon JavaDoc API reference (HTML):

- [Corticon Server API](#)
- [CorticonExtensions API](#)

See also:

- [Introducing the Progress® Application Server](#)
- Corticon documentation for this release on the [Progress download site](#): What's New Guide (PDF), Installation Guide (PDF), PDF download package, and the online Eclipse help installed with Corticon Studio.