# Corticon
## Web Services

# Copyright

© 2019 Progress Software Corporation and/or one of its subsidiaries or affiliates. All rights reserved.

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

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Deliver More Than Expected, Icenium, Kendo UI, Kinvey, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), Sitefinity, 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 Autonomous REST Connector, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustAssembly, JustDecompile, JustMock, NativeChat, NativeScript Sidekick, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, 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.

**Updated: 2019/07/31**

# Table of Contents

1

# What is a web service

**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 or REST messages, typically conveyed using HTTP or HTTPS with corresponding JSON-formatted or XML-serialized text files.

**What is a Decision Service?**

A Decision Service automates a discrete decision-making task. It is implemented as a set of business rules and exposed as a web service. By definition, the rules within a Decision Service are complete and unambiguous; for a given set of inputs, decision integrity is ensured because the Decision Service addresses every logical possibility uniquely. .

**What is a web services caller?**

A web services caller 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 call web services.

2

# Overview of Corticon web services

Corticon Server hosts web services with Java on Windows using an embedded application server, and can host Java web services on several Windows, UNIX, and Linux application servers and with .NET on Microsoft IIS server, as detailed on the Progress Software Progress Corticon 6.0 - Supported Platforms Matrix..

The embedded Java application server is Apache Tomcat, which serves as the test application server. Corticon Studio acts as a web service client, which allows you to make REST and SOAP calls to a running application server.

Corticon Studio is where Decision Services are created and tested, and then packaged for deployment on an application server as web services. The Corticon Server can be a stand alone server or deployed in a .NET or java application.

**Note:** The Corticon KnowledgeBase entries on WAR installation for different application servers provide detailed instructions on configuring Java web services on other application servers.

**Deploying web services with Java**

The Corticon Server installer contains an Apache Tomcat web server with a complete set of utilities and administrative tools. The installer creates a Tomcat instance configured with Corticon's default web services deployed using Corticon's web application axis.war. This web application allows web service clients to interact with the Corticon Server and its deployed Decision Services running on that instance.

The Tomcat application server (http://tomcat.apache.org/) includes a Java servlet container for hosting web applications. Tomcat is the most widely used application server on the market. Corticon uses a default Tomcat application server to demonstrate how web service clients can deploy and execute against Decision Services through the web service endpoints inside the axis.war web application.

**Deploying web services with .NET**

The Corticon Server for .NET contains all necessary files for the Corticon Server to run in a local Microsoft IIS server. The installer will not preconfigure IIS with Corticon's web services. You must run a bscript as Administrator to move all necessary files to a local IIS Server (`c:/inetpub/wwwroot`) and deploy the Corticon web services.

For details, see the following topics:

- Web services on Java
- Web services on .NET

## Web services on Java

The Corticon Server for Java can be installed into any application server that has a Java container. As part of the Corticon Server installer, a web application called axis.war is downloaded. This axis.war contains the web services that will call into the Corticon Server running in the application server. These web services allow clients to execute against Decision Services through SOAP and REST. The administrative web services (deploy and modify Decision Services) can be called through REST only.

As you approach production, you might want to access the server download bundle that enables the Corticon Server to run on other application servers: Progress Corticon 6.0 - Supported Platforms Matrix. There are Corticon KnowledgeBase entries that describe each application server setup.

**Note:** **HTTPS** - If you want to set up your Java server for SSL-secured communications, see the topic

### Set up Corticon Server for Java

The describes the system requirements and options for Corticon Server installation. Download and run the installer for Corticon Server. Choose the Corticon Server for Java option. The installer will install utilities, scripts, and samples as well as resources that will be provisioned to the Java server.

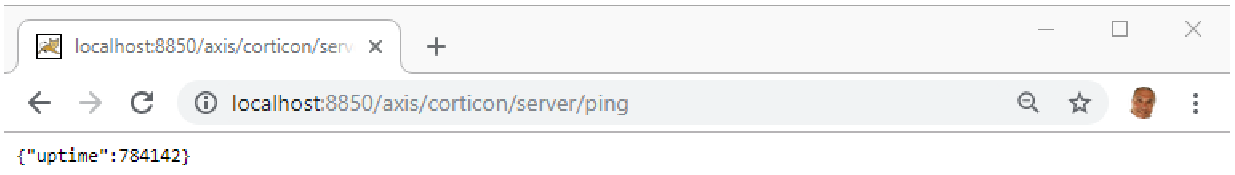
The installer installs Apache Tomacat as the default Java application server. Just start the Corticon server and then test the Java server..

### Test the installed Corticon Server

Once the axis.war has been deployed and the application server is running, type the following URL in your preferred browser:

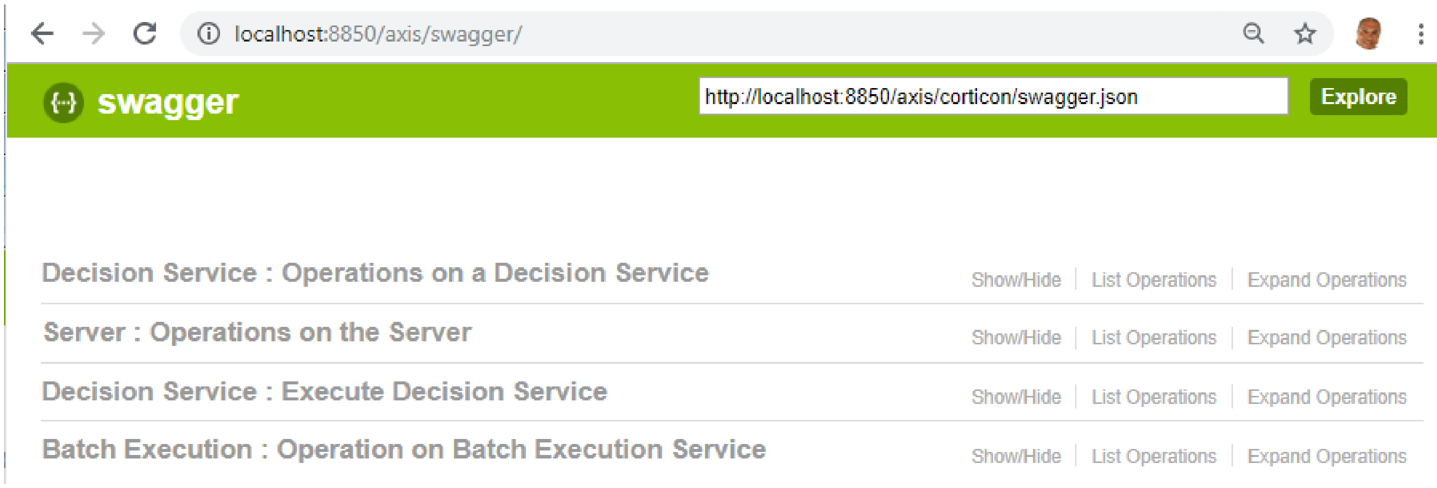
```
http://localhost:8850/axis/corticon/server/ping
```

That makes a REST call to the running Corticon Server to get the time (in milliseconds) that the Corticon Server has been running, as illustrated:

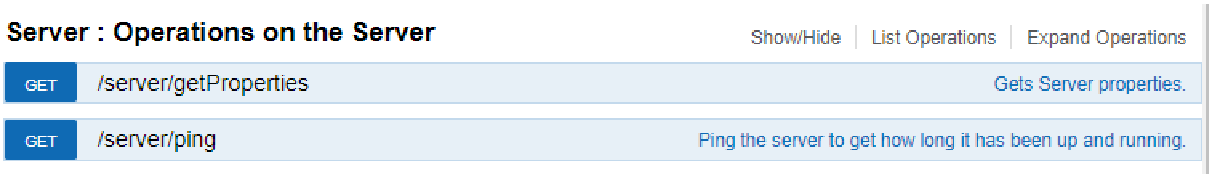

localhost:8850/axis/corticon/serv
localhost:8850/axis/corticon/server/ping

```
{"uptime":784142}
```

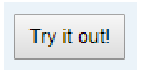
Additionally, you can use any REST or SOAP test tools, such as Postman, Swagger or SoapUI, to connect through the web services deployed inside the web application axis.war. Corticon has incorporated Swagger into our REST endpoints. To access Corticon's Swagger page, enter the URL to the machine with its port, the web application name, and the string "swagger". The Tomcat server that comes with the Corticon Server install has opened port 8850 to receive requests. The web application name, from the axis.war, is "axis". The full URL to connect to the Corticon Swagger page is: http://localhost:8850/axis/swagger


localhost:8850/axis/swagger/
swagger
http://localhost:8850/axis/corticon/swagger.json
Explore
Decision Service : Operations on a Decision Service
Server : Operations on the Server
Decision Service : Execute Decision Service
Batch Execution : Operation on Batch Execution Service
Show/Hide
List Operations
Expand Operations

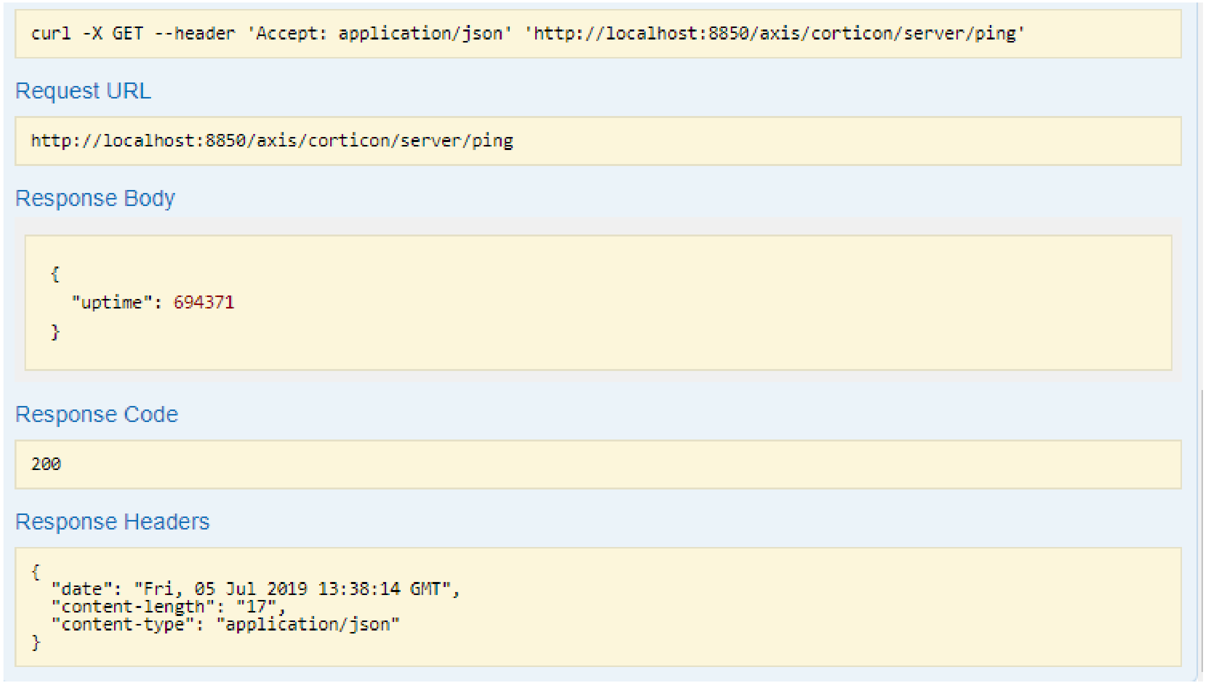
Choose **Server**, and then choose the **ping** command:



On the panel that drops down with implementation notes and response messages, click the **Try it out!** button:



The response is in the following form



```
curl -X GET --header 'Accept: application/json' 'http://localhost:8850/axis/corticon/server/ping'
```

Request URL

```
http://localhost:8850/axis/corticon/server/ping
```

Response Body

```
{
  "uptime": 694371
}
```

Response Code

```
200
```

Response Headers

```
{
  "date": "Fri, 05 Jul 2019 13:38:14 GMT",
  "content-length": "17",
  "content-type": "application/json"
}
```

The Corticon Java web service is running.

## Secured deployment on Java web services

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

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

### Implementing deployment security

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

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

#### Setting up authentication for secure server access

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

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

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

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

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

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

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

The default user definitions for Apache Tomcat are defined in the `tomcat-users.xml` file (in a default installation its location is `[CORTICON_HOME]\Server\tomcat\conf\tomcat-users.xml`) as follows:

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

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

You can modify the passwords and add additional users to this file.

## Securing Server endpoints

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

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

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

1. Corticon Execute SOAP Servlet:

   ```
   /services/CorticonExecute
   ```

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

2. Corticon Decision Service Admin REST:

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

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

3. Corticon Server Admin REST:

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

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

4. Corticon Execute REST:

   ```
   /corticon/execute
   ```

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

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

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

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

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

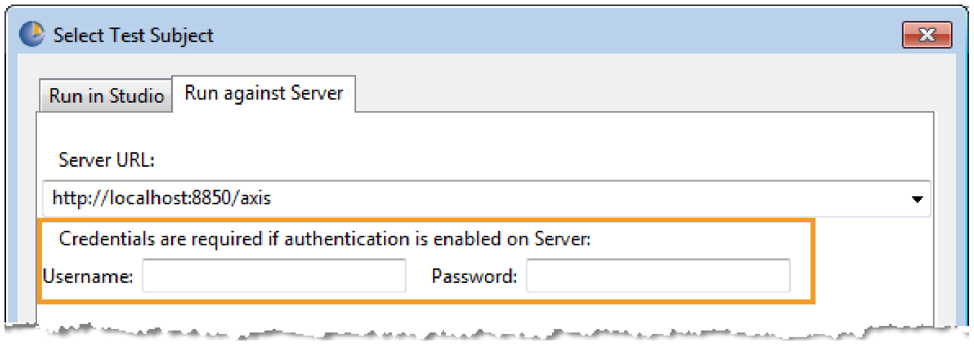
Each role-name defined in any security-constraint in the `web.xml` must be defined in a `<security-role>` tag.

```
Example: (All role names)

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

Example: (Two specific role names)

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

**Note:** If you want all role names to be valid for the web application, then add `*` in the XML Element. Even if you have set the constraint’s role-name to `*`, you have to define a `<security-role>` role-name with `*`.

Using authentication in Corticon Studio

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



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:

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

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

## Using authentication in client requests

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

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

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

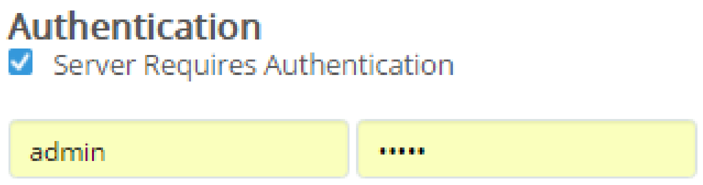
```
HttpPost postRequest = new HttpPost("http://localhost:8850/corticon/execute");
```

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

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

String base64EncodedCredentials = "Basic " + encodedString;

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

## Using authentication within Web Console

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



Authentication

☑ Server Requires Authentication

admin | •••••

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

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

Using API calls to a secure web application

The following are examples of each of the core URL patterns outlined in Securing Server endpoints on page 13.

### CorticonExecute SOAP example

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

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

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

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

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

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

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


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

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

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

### Corticon Decision Service Admin REST example

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

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

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

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

HttpGet getRequest = new HttpGet(serverURL);

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

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

String base64EncodedCredentials = "Basic " + encodedString;

getRequest.setHeader("Authorization", base64EncodedCredentials);

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

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

   throw new Exception(error);
}

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

### CorticonAdmin REST example

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

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

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

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

HttpGet getRequest = new HttpGet(serverURL);

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

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

String base64EncodedCredentials = "Basic " + encodedString;

getRequest.setHeader("Authorization", base64EncodedCredentials);

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

//  First verify for valid status code
```

```
int statusCode = response.getStatusLine().getStatusCode();
if (statusCode != 200)
{
String error = "Failed with HTTP error code " + statusCode + ": " +
response.getFirstHeader("error");

throw new Exception(error);
}

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

### CorticonExecute REST example

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

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

import javax.ws.rs.core.MediaType;

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

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

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

HttpPost postRequest = new HttpPost(serverURL);

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

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

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

String base64EncodedCredentials = "Basic " + encodedString;

postRequest.setHeader("Authorization", base64EncodedCredentials);

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

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

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

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

order.put("__metadata", orderMetadata);

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

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

item.put("__metadata", itemMetadata);

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

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

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

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

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

   throw new Exception(error);
}

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

## Setting up encryption between servers and clients

### Enabling HTTPS

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

---

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

---

To enable HTTPS on Corticon Server for Java:

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

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

   Add in the following block to replace … above:

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

   ---

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

   ---

5. Save the file.

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

---

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

---

### Enabling the Corticon Studio to publish to a secure Corticon Server

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

### How to enable Kerberos Authentication in Studio and Server

When your organization has established a Kerberos realm on an EDC or ADC Datasource, you can enable your Studio and Server Windows installations to successfully connect using the Kerberos Authentication option instead of user credentials by enabling the `AllowTgtSessionKey` registry key, and then running Corticon as Administrator.

**To enable the `AllowTgtSessionKey` registry key on Studio or Server:**

1. Open `regedit.exe`, and then navigate to the node:

   ```
   HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\Kerberos\Parameters
   ```

2. Right-click on `Parameters`, then choose **New > DWORD**, and name it `AllowTgtSessionKey`. Double-click on the new value name to enter the **Value Data** `1`, and then click **OK**.
3. Restart the computer to allow the registry changes to take effect and to refresh the JVM.

**To run Corticon Studio or Server as Administrator:**

1. Locate the shortcut to **Start Corticon Server** or **Start Corticon Studio**.
2. Right-click, and then choose **Run as Administrator**.

Test the connection where the Datasource connection has elected to use Kerberos authentication in Studio or execute a request against a Corticon Server Decision Service.

## Web services on .NET

Corticon Server for .NET uses the Microsoft Internet Information Service (IIS)10 as its application server for web services. The first step is to get IIS up and running. On Windows 10 and newer systems, IIS and its supporting .NET Framework are pre-installed but need to be enabled to start working. More than an on/off switch, the multiple features of IIS need to be selected to support the functions it will perform. The Corticon Server for .NET has no application server of its own.

The features you set up for IIS on Windows 10 or Windows 2016 are detailed in a Progress Knowledgebase document.

Once IIS is set up, proceed to Set up Corticon Server for .NET on page 21

## Set up Corticon Server for .NET

The describes the system requirements and options for Corticon Server installation. Download and run the installer for Corticon Server. Choose the Corticon Server for .NET option. The installer will install utilities, scripts, and samples as well as resources that will be provisioned to the IIS server.

### Install Corticon assets onto the IIS server

You always start by installing or upgrading the Corticon Server for .NET resources, and then running its installer to push the resources to the IIS location.

**To install or upgrade Corticon Server for .NET into the IIS location:**

1. Start **Administrative Tool > IIS Manager**. In the **Actions** panel, click **Stop**.
2. In `[CORTICON_HOME_6.0]\Server .NET\IIS`, right click `install.bat` , and then select **Run as administrator**.

3. Accept or adjust the target location, and then press **Enter**.
4. Enter **A** in each section of the script to accept all the files. The 6.0 resources are copied to the IIS `\axis` directory.
5. In the **IIS Manager**, on the **Actions** panel, click **Start**.

Install or upgrade Corticon Server for .NET on your designated Windows machine so that you can provision the resources for the IIS server. See *"Running the Server and Web Console installer wizard"* and *"Preparing Corticon .NET Server resources for an IIS server" in the Corticon Installation Guide.*

## Create the application in IIS

**To convert the `axis` directory into an application:**

1. In the IIS Manager, select the **DefaultWebsite**, then right-click on the root, and select **Refresh**.
2. Expand the tree and navigate to the `axis` directory.
3. Right-click on the `axis` folder, and then choose **Convert to Application**.

In the **Add Application Pool**dialog that opened, choose **.NET v4.5 Classic**, and then click `OK` in both dialog boxes.

## Set Access Permissions for axis directories in IIS 10.0

Corticon Server for .NET needs to be able to write to various subdirectories of the IIS installation directory's `axis` subdirectory. The following task might be required for your setup, as the Classic Application Pool runs under the built in user account `ApplicationPoolIdentity`, a user that has no `WRITE` permissions for the `axis` folders, so these rights must be set.

To set access permissions:

1. Inthe IIS Manager, , select the **DefaultWebsite**, then right-click on the root, and select **Refresh**.
2. Expand the tree and navigate to the `axis` directory.
3. Right-click on the `axis` folder, and then choose **Edit Permissions**.
4. On the Secuity tab, edit to assign `WRITE` permission to the account `IIS_IUSRS`.

The installation is now complete.

It is a good idea to restart IIS now. To do so in the IIS Manager, right-click on the root to choose **Stop**, and then after a few seconds, choose **Start**. (You might prefer to execute `iisreset` in a Command Prompt to restart IIS.)

The IIS Internet Information Services is now running the `axis` web service for Corticon.

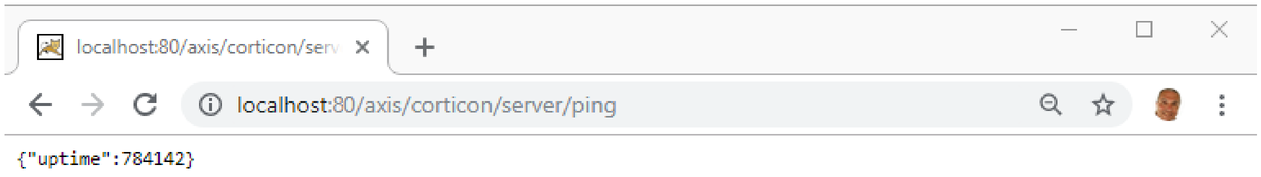
**IMPORTANT:** Once you have a non-evaluation license JAR, navigate to the IIS installation location, typically `C:\inetpub`, and then navigate to its `\wwwroot\axis\lib` directory to paste the file and, if necessary, overwrite the existing file in that location.

## Test the IIS server Corticon application

Once install.bat has copied all the necessary assets to the IIS server and the IIS server is running, type the following URL in your preferred browser:

```
http://localhost:80/axis/corticon/server/ping
```

That makes a REST call to the running Corticon Server to get the time (in milliseconds) that the Corticon Server has been running on IIS, as illustrated:


localhost:80/axis/corticon/serv
localhost:80/axis/corticon/server/ping

```
{"uptime":784142}
```

The Corticon .NET web service is running.

3

# Corticon Server files and API tools

Corticon Server is a set of Java classes, packaged in Java archive files. The required set of JARs needed to deploy and call Corticon Server is:

| JAR | Description |
|---|---|
| `CcServer.jar` | The main Corticon Server JAR, containing the core engine logic. |
| `CcConfig.jar` | Contains a set of `.property` files that list and set all the configuration properties needed by Corticon Server. These properties pages are not intended for user access. Instead, the file `brms.properties`, installed by every product at the root of `[CORTICON_WORK_DIR]`, enables you to add your override settings to be applied after the default settings have been loaded. |
| `CcLicense.jar` | An encrypted file containing the licensing information required to activate Corticon Server |
| `CcThirdPartyJars.jar` | Contains third-party software such as JDBC and WSDL. |
| `CcI18nBundles.jar` | Contains the language templates used by Corticon Server |
| `CcExtensions.jar` | Supports Service Callouts and extended language operators. |

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:

| | |
|---|---|
| `CORTICON_HOME` | The installation directory -- either the default location, `C:\Progess\Corticon 6.0`, or the preferred location you specified. |
| `CORTICON_WORK_DIR` | The work directory -- either the default location, `C:\Users\{`*`username`*`}\Progress\CorticonWork 6.x`, or the preferred location you specified. |

**Note:**

**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.

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.

## Corticon Server 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. 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_6.x\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.

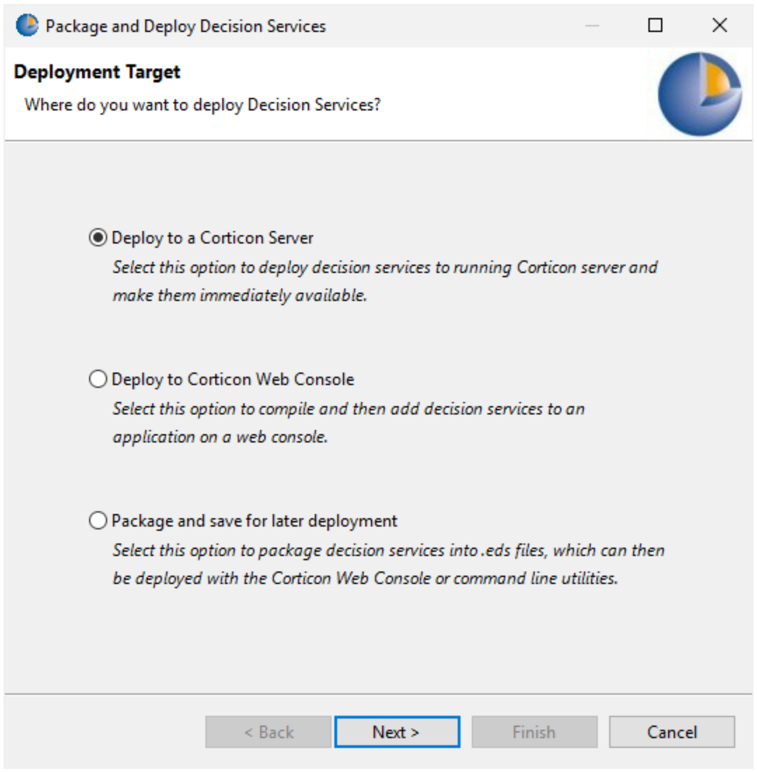
**Note:** If the location specified by `com.corticon.ccserver.sandboxDir` cannot be found or is not available, the Sandbox location defaults to the current working directory as it is typically the location that initiated the call.

4

# Deploy rules to the Corticon Server

Once you have installed the Corticon Server, it is ready to have rules deployed to it that can then run when it receives requests. Rules are defined in the Corticon Studio, and tested there against its embedded application server. Ruleflows are then versioned and packaged together with supporting files as compiled Decision Services for runtime on any Corticon Server regardless of platform and application server. The compilation into an encrypted Decision Service ensures that the rules are consistent and reliable wherever they run, whether as a web service on an application server or as an in-process application. The absence of the source files on the server and ad hoc compilation ensures security from unwarranted changes to the rules as well as exposure of the valuable rule assets.

The Corticon Studio's **Package and Deploy Decision Services wizard** summarizes the tactics that get rules onto servers:


Package and Deploy Decision Services

**Deployment Target**

Where do you want to deploy Decision Services?

[✓ Deploy to a Corticon Server]

*Select this option to deploy decision services to running Corticon server and make them immediately available.*

Deploy to Corticon Web Console

*Select this option to compile and then add decision services to an application on a web console.*

Package and save for later deployment

*Select this option to package decision services into .eds files, which can then be deployed with the Corticon Web Console or command line utilities.*

- **Deploy to a Corticon Server**

  Does the compilation and server transfer in one step. Great for pre-production testing. But many users prefer to keep the development functions and production control distinct.

  In many cases, several Decision Services need to be created in the same run. You can use compile and deploy APIs and command line utilities to create scripted runs of builds. See topics *in the "Packaging and deploying Decision Services" section in the Deployment Guide*

- **Deploy to the Corticon Web Console** While not required, the Web Console is a good way to monitor and manage servers and Decision Services. By deploying to Web Console, the development hands off the compiled Decision Service for eventual deployment to a web service it has under management. Staging a Decision Service this way is discussed in *"Decision Services and Applications" in the Web Console Guide*.

- **Package and save for later deployment** - Compiling the assets into a Decision Service file on a network location, this intermediate step typically uses Deployment Descriptor files that assembles several Decision Services and miscellaneous files into a easily edited text file.

## Create and install Deployment Descriptor files

Deployment Descriptor files (`.cdd`) let you provide a server with lists of Decision Service, related files, and optional property settings for each one. When Corticon Server starts, it looks for Deployment Descriptor files at `[CORTICON_WORK_DIR]\cdd`.

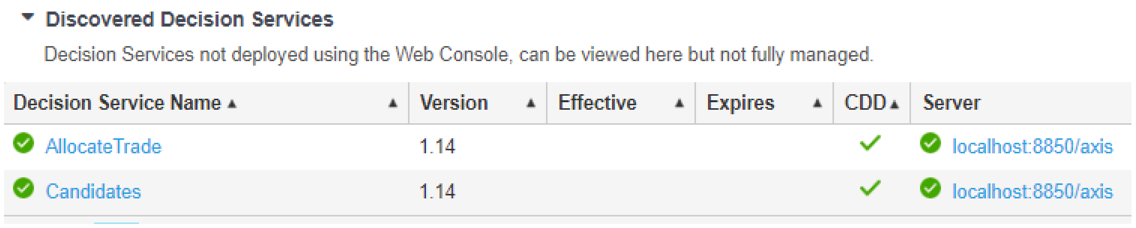
The Samples directory in a server installation provides some samples that include prewritten CDD files as well as precompiled Decision Services. Try them out to see how the Web Console shows that they immediately deploy and accept requests.

Here is the `TradeAllocation` sample's Deployment Descriptor file:

```
<?xml version="1.0" encoding="UTF-8"?>
<cdd soap_server_binding_url="http://localhost:8850/axis">
  <decisionservice>
    <name>AllocateTrade</name>
    <path>AllocateTrade.eds</path>
    <options>
      <option name="PROPERTY_AUTO_RELOAD" value="true" />
      <option name="PROPERTY_MAX_POOL_SIZE" value="1" />
    </options>
  </decisionservice>
  <decisionservice>
    <name>Candidates</name>
    <path>Candidates.eds</path>
    <options>
      <option name="PROPERTY_AUTO_RELOAD" value="true" />
      <option name="PROPERTY_MAX_POOL_SIZE" value="1" />
    </options>
  </decisionservice>
</cdd>
```

*Try it!* To explore this deployment technique, the Web Console is the best way to visualize what happens.:

1. Start the Web Console in a clean installation. Note that there are no Decision Services listed.
2. In the server install, go to `[CORTICON_WORK_DIR]\Samples\Rule Projects\TradeAllocation`
3. Copy the Deployment Descriptor CDD file and the two Decision Service EDS files to `[CORTICON_WORK_DIR]\cdd`
4. Refresh the Web Console. The **Discovered Decision Services** list both Decision Services that were deployed to the server stated in the CDD file:



**Discovered Decision Services**

Decision Services not deployed using the Web Console, can be viewed here but not fully managed.

| Decision Service Name ▲ | Version ▲ | Effective ▲ | Expires ▲ | CDD ▲ | Server |
|---|---|---|---|---|---|
| ✓ AllocateTrade | 1.14 | | | ✓ | ✓ localhost:8850/axis |
| ✓ Candidates | 1.14 | | | ✓ | ✓ localhost:8850/axis |

For more information, see the topics in *"Using Deployment Descriptors to deploy Decision Services" in the Deployment Guide*.

## Setting Server startup to auto load CDD files

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

------------------------------------------------------------------------------

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

`com.corticon.ccserver.autoloaddir.enable`

Specifies whether the CcServer will try to auto load CDD files

Default value: true

`com.corticon.ccserver.autoloaddir`

The directory where the CDD files are located.

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

- `ICcServer.setDeploymentDescriptorDirectoryPath(String)`

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

Default value: If the property value is empty, the following path is used: `user.dir/cdd`

where `user.dir` is the value of environment variable "user.dir" which in Windows and Unix returns the directory where container application was started.

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

5

# How to call a Decision Service

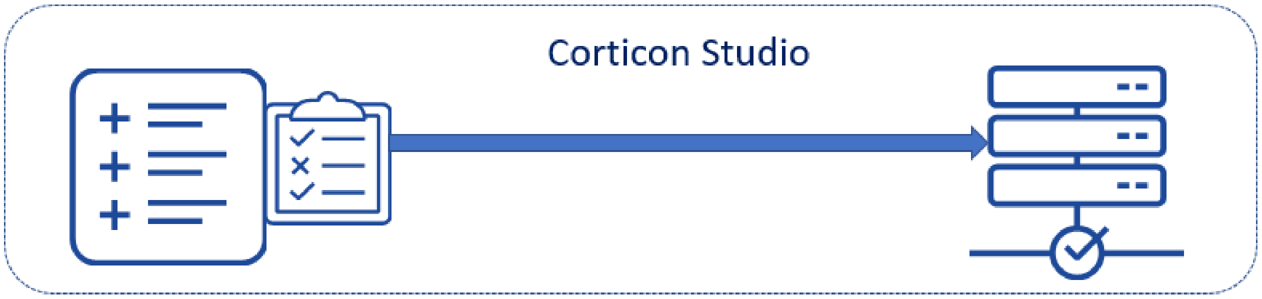
Once the Corticon Server is installed and running deployed Decision Services, you can call this Decision Service by sending a request messages, and then inspect the response messages that return.

A Decision Service in a process or application works best when it using 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 *called* the Decision Service.

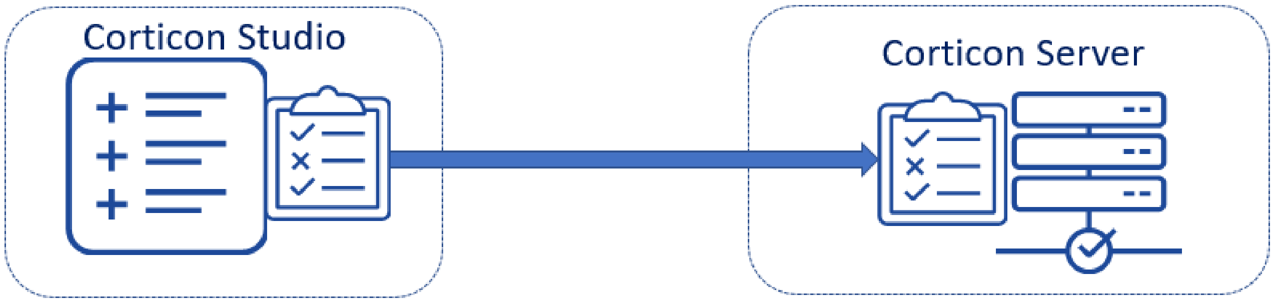
**To call a Decision Service as a web service**:

- **Within Corticon Studio -**



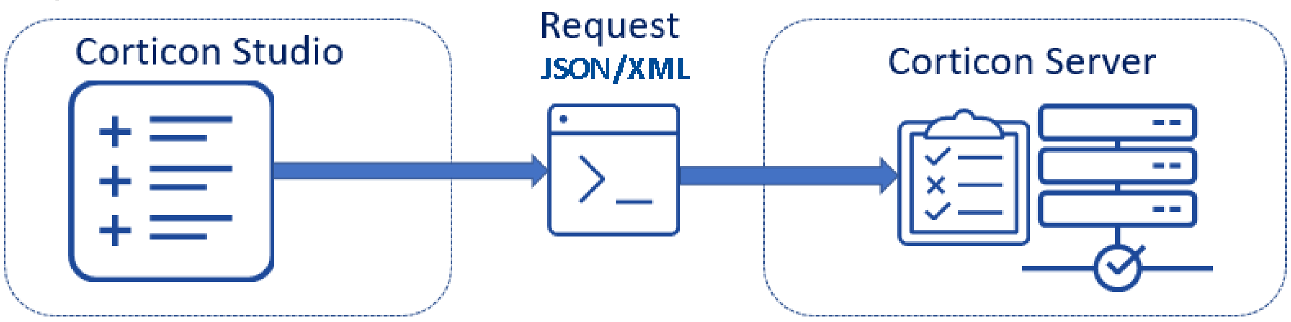
When you run a Ruletest for a Ruleflow in Corticon Studio, the embedded in-process Corticon server is called to process the request and return the response. You did this in the Basic Tutorial. It is one of Corticon's greatest strengths, the ability to real-world test right on the workbench. See the Quick Reference topics and *"Choosing a test subject in the Studio workspace" and "Executing tests".*

- **From Corticon Studio to a Corticon Server -**



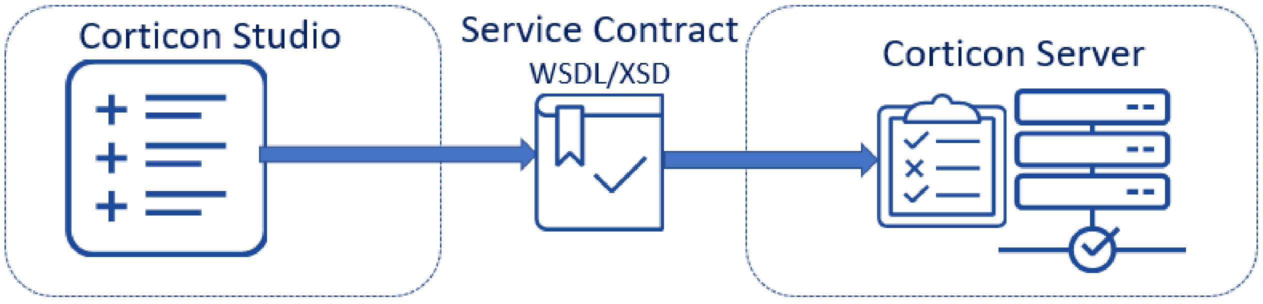
When you have an installed Corticon Server, whether Java or IIS, of the same version as Studio, you can run Ruletests as simulated Decision Services directly from the Studio. This technique enables pre-production testing of a Decision Service while it is still in development. See *"Choosing a test subject that is a deployed Decision Service" in the Quick Reference topics.*

- **As a request document submitted to a Corticon Server -**



You can abstract the tests from the Studio by exporting the JSON, SOAP and XML requests to text files. With a few minor edits, you can run them against a deployed Server from the Web Console. See the Quick Reference topics on *"Exporting Testsheets".*

- **As a service contract request submitted to a Corticon Server -**



See *"Integrating Corticon Decision Services" topics in the Deployment topics.*

6

# Invoking Corticon Server

To make a call to invoke Corticon server, you can use REST or SOAP.

For details, see the following topics:

## REST call

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

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

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

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

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

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

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

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

where *base* is the Corticon Server's IP or DNS-resolvable name and port. For example:

```
{
  "name": "ProcessOrder",
  "majorVersion":"1",
  "minorVersion": "16",
  "Objects": <JSONArray of JSONObjects>
}
```

### Success and error responses

The response returned by the server has two parts:

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

**Success response** - In addition to the processed request, when execution is successful, the response message's body contains the updated JSON string. A successful execution's shows its status (200) and header.

### Common error behavior

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

## SOAP call

The structure and contents of a SOAP message are a wrapper on an XML request. For example:

1. In the ProcessOrder sample, open its Ruletest.
2. Choose **Ruletest > Testsheet > Data > Inpput > Export Request XML**.
3. Save the file as `ProcessOrderXML.xml`
4. Choose **Ruletest > Testsheet > Data > Inpput > Export Request SOAP**.
5. Save the file as `ProcessOrderSOAP.xml`
6. Compare the two files.

The SOAP request wrapped the XML request content within:

```
<SOAP-ENV:Envelope
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <CorticonRequest/>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Once the SOAP request message has been prepared, a SOAP client must transmit the message to the Corticon Server (deployed as a Web Service) via HTTP.

If your SOAP tools have the ability to assemble a compliant request message from a WSDL you generated in Corticon Studio, then it is very likely they can also act as a SOAP client and transmit the message to a Corticon Server deployed to a supported application server. The Web Services section describes how to test this method of invocation with a third-party tool or application as a SOAP client.

When developing and testing SOAP messaging, it is very helpful to use some type of message interception tool (such as **tcpTunnelUI** or **TCPTrace**) that allows you to grab a copy of the request message as it leaves the client and the response message as it leaves Corticon Server. This lets you inspect the messages and ensure no unintended changes have occurred, especially on the client-side.

## Request Response mode

Regardless of which invocation method you choose to call Corticon Server, keep in mind that it, by default, acts in a “request—response” mode. This means that one request sent from the client to Corticon Server will result in one response sent by the Server back to the client. Multiple calls may be made by different clients simultaneously, and the Server will assign these requests to different Reactors in the pool as appropriate. As each Reactor completes its transaction, the response will be sent back to the client.

Also, the form of the response will be the same as the request. If making a SOAP call passing XML, the response will be XML. If making a REST call passing JSON, the response will be JSON.