Progress®
Corticon®

Corticon
Server

Copyright

Visit the following page online to see Progress Software Corporation's current Product Documentation Copyright Notice/Trademark Legend: https://www.progress.com/legal/documentation-copyright.

Last updated: Corticon 7.2

Updated: 2025/07/20

Table of Contents

1

About Corticon Servers

Progress® Corticon® Business Rules Server processes the rules modeled, verified and tested in Corticon Studio. Corticon Server is a natural fit for today's deployment architectures, supporting on-premise and cloud deployment, web service deployment in popular application servers, in-process deployment for real-time applications, and application containers.

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

- The **Corticon Server for Java** provides the necessary components to deploy Corticon as a REST or SOAP service on a Java application server or to deploy Corticon in-process in your custom Java application. Corticon Server provides installers for both Windows and Linux. The actual deployment artifacts - the JAR and WAR files - are platform independent. See the Web Services and In-Process guides for more information. See the Corticon Supported Platforms Matrix for a list of supported application servers.

  Note: IMPORTANT: No default Java and application server - Corticon Server and Web Console no longer install Java or a standard Tomcat distribution. See the various topics in the *Corticon Installation Guide* that describe the setup of Java, Tomcat application server, and the Corticon Server.

- The **Corticon Server for .NET** provides the necessary components to deploy Corticon as a REST or SOAP service to Microsoft Internet Information Services (IIS) or to deploy Corticon in-process in your custom .NET application. Corticon Server .NET install is only available on Windows. Corticon uses a high performance bridging technology to call from .NET languages such as C# to Corticon Server. See the Web Services and In-Process guides for more information. See the Corticon Supported Platforms Matrix for a list of supported platforms.

2

Inside Corticon Server

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

For details, see the following topics:

- The basic path
- About working memory
- About Server properties
- Multi-threaded execution
- Reactor state
- Server state
- Dynamic discovery of new or changed Decision Services
- Exception handling
- Batch processing
- Set Server startup to auto load CDD files

The basic path

Client applications invoke Corticon Server, sending a data payload as part of a request message. The invocation of Corticon Server can be either indirect (such as when using REST) or direct (such as when making in-process Java calls). This request contains the name of the Corticon Decision Service (the Decision Service Name assigned in the Deployment Descriptor file that should process the payload.) Corticon Server matches the payload to the Decision Service and then commences execution of that Decision Service. One Corticon Server can manage multiple, different Decision Services, each of which might reference a different Vocabulary.

About working memory

When a Reactor (an instance of a Decision Service) processes rules, it accesses the data resident in “working memory”. Working memory is populated by any of the following methods:

1. **The payload of the Corticon Request**- The payload – whether a REST or XML document, or a reference to Java business objects – is inserted into working memory when the client’s request (invocation) is received. When running a Studio Test, the Studio itself is acting as the client, and it inserts the data from the Input Ruletest into working memory.
2. **The results of rules**- During rule processing, some rules may create new data, modify existing data, or even delete data. These updates are maintained in working memory until the Decision Service completes execution.
3. **An external Datasource**- If, during the course of rule execution, some data is required that is not already in working memory, then the Reactor asks Corticon Server to query and retrieve it from an external Datasource. For Datasource access to occur, Corticon Server or Studio Tester must be configured correctly and the Vocabulary must be mapped to the datasource's schema.

About Server properties

The Corticon Server is configured by default to fit the majority of use cases. Recognizing that this configuration may not be optimal for all situations, Corticon provides numerous properties for altering the behavior of Corticon Server. These properties are set via the Corticon `brms.properties` file in your `CORTICON_WORK` directory. When deploying Corticon in-process, you can also set these properties when instantiating a Corticon Server. Where properties are referenced, this guide presents how to set them via `brms.properties`. See the topic "Properties in context" and Corticon Server API JavaDoc for more information.

Multi-threaded execution

Multiple Decision Services place their requests in a queue for processing. Server-level thread pooling is implemented by default, using built-in Java concurrent pooling mechanisms to control the number of concurrent executions. This design allows the server to determine how many concurrent requests are to be processed at any one time.

Execution queue

Each thread coming into the Server gets an available Reactor from the Decision Service, and then the thread is added to the Server's Execution Thread Pooling Queue, or, simply put, the ***Execution Queue***. The Execution Queue guarantees that threads do not overload the cores of the machine by allowing a specified number of threads in the Execution Queue to start executing, while holding back the other threads. Once an executing thread completes, the next thread in the Execution Queue starts executing.

The Server will discover the number of cores on a machine and, by default, limit the number of concurrent executions to that many cores, but a property can be set to specify the number of concurrent executions. Most use cases will not need to set this property. However, if you have multiple applications running on the same machine as Corticon Server, you might want to set this property lower to limit the system resources Corticon uses. While this tactic might slow down Corticon processing when there is a heavy load of incoming threads, it will help ensure Corticon does not monopolize the system. Conversely, if you have Decision Services which make calls to external services (such as connection to a Datasource) you may want to set this property higher so that a core is not idle while a thread is waiting for a response.

Memory management

Allocation means that you could allocate hundreds of execution threads for one Decision Service. The way Reactors are maintained in each Decision Service, the Server can re-use cached processing data (not including payload data) across all Reactors for the Decision Service. Runtime performance should reveal only modest differences in memory utilization between a Decision Service that contains just one Reactor and another that contains hundreds of Reactors. Because each Reactor reuses cached processing data, the Server can dynamically create a new Reactor per execution thread (rather than creating Reactors and holding them in memory.) Even when an allocation is set to 100, the Server only creates a new Reactor (with cached data) for every incoming execution thread, up to 100. If there are only 25 execution threads against Decision Service 1, then there are just 25 Reactors in memory. Large request payloads are more of a concern than the number of concurrent executions or the number of Reactors.

Related Server properties

The following Server properties let you adjust Server executions:

- Determines how many concurrent executions can occur across the Server. Ideally, this value is set to the number of Cores on the machine. By default, this value is set to `0`, which means the Server will auto-detect the number of Cores on the server.

  ```
  com.corticon.server.execution.processors.available=0
  ```

- This is the timeout setting for how long an execution thread can be inside the Execution Queue. The time starts when the execution thread enters the Execution Queue and ends when it completes executing against a Decision Service. A `CcServerTimeoutException` will be thrown if the execution thread fails to complete in the allotted time. The value is in milliseconds. Default value is `180000` (180000ms = 3 minutes)

  ```
  com.corticon.server.execution.queue.timeout=180000
  ```

Note: The property `com.corticon.server.execution.queue.timeout` sets how long Corticon will allow the execution thread to execute before killing it, thus preventing a reactor from being permanently locked by the execution. Examples where this is appropriate are: (1) when your rules access a data source that blocks but does not return in a reasonable amount of time, or (2) when calling into a custom extension that goes into an infinite loop or blocked state.

Reactor state

A Reactor is an executable *instance* of a deployed Decision Service. Corticon Server acts as the broker to one or more Reactors for each deployed Decision Service. During Decision Service execution, the Reactor is a stateless component, so all data state must be maintained in the message payloads flowing to and from the Reactor.

If a deployed Ruleflow contains multiple Rulesheets, state is preserved across those Rulesheets as the Rulesheets successively execute within the Ruleflow. However, no interaction with the client application occurs between or within Rulesheets. After the last Rulesheet within the Ruleflow is executed, the results are returned back to the client as a `CorticonResponse` message. Upon sending the `CorticonResponse` message, the Reactor is deleted from memory. A new Reactor will be created for the next incoming `CorticonRequest`.

As an integrator, you must keep in mind that there are only two ways for you to retain state upon completion of a Decision Service execution:

1. Receive and process the data from within the `CorticonResponse` message.
2. Persist the results of a Decision Service execution to an external database.

Once a Decision Service execution has completed, the Reactor itself does not remember anything about the data it just processed.

Server state

Although data state is not maintained by Reactors from transaction-to-transaction, the names and deployment settings of Decision Services deployed to Corticon Server are maintained. The file `ServerState.xml`, located in `[CORTICON_WORK_DIR]\{INP|SER}\CcServerSandbox\DoNotDelete`, maintains a record of the Ruleflows and deployment settings currently loaded on Corticon Server for each Decision Service. If Corticon Server inadvertently shuts down, or the container crashes, then this file is read upon restart and the prior Server state is re-established automatically.

How to turn off server state persistence

By default, Corticon Server automatically *creates and maintains* the `ServerState.xml` document during normal operation, and *reads* it during restart. This allows it to recover its previous state in the event of an unplanned shutdown (such as a power failure or hardware crash)

However, Corticon Server can also operate without the benefit of `ServerState.xml`, either by not reading it upon restart, or by not creating/maintaining it in the first place. In this mode, an unplanned shutdown and restart results in the loss of any settings made through the Corticon Web Console. For example, any properties settings made, or `.eds` files deployed using the Console, will be lost. The Server will not auto reload the Decision Services because there is no ServerState.xml; however, from within the Web Console, the Decision Services will be in an undeployed state and can be redeployed from the Web Console.

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

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

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

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

You can customize Corticon Server's state and restart behavior by combining these two property settings:

serverstate .persistchanges	serverstate .load	Server Restart Behavior
`true`	`true`	Corticon Server maintains `ServerState.xml` during operation, and automatically reads it upon restart to restore to the old state.
`true`	`false`	Corticon Server maintains `ServerState.xml` during operation, but does NOT automatically read it upon restart. New Server state upon restart is unaffected by `ServerState.xml`. This allows a system administrator to manually control state restoration from the `ServerState.xml`, if preferred.
`false`	`true`	Corticon Server attempts to read `ServerState.xml` upon restart, but finds nothing there. No old state restored.
`false`	`false`	no `ServerState.xml` document exists, and Corticon Server does not attempt to read it upon restart. No old state restored.

Dynamic discovery of new or changed Decision Services

To support the use of Corticon Deployment Descriptors (CDD) files for deployment, the Corticon Server periodically checks for new CDD files or new versions of EDS files referenced by CDD files. When a new CDD file is detected, the EDS files referenced in it are loaded. When a new EDS file is detected, it is loaded. Any changes are automatically reflected in the Server's state.

The default interval for dynamic discovery is 30 seconds. It can be changed by setting the property:

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

Dynamic discovery can be disabled by setting the property:

```
com.corticon.ccserver.dynamicupdatemonitor.autoactivate=false
```

Exception handling

When an exception occurs, the Corticon Server throws a Java exception. Exceptions are typically recorded in the Corticon Server log file. When Corticon is deployed in-process, exceptions can be thrown by API methods as defined in the Corticon Server API JavaDoc.

Batch processing

Batch processing enables Corticon Server to efficiently retrieve, process, and update large data sets stored in a relational database. A special "execute" request starts a batch execution where Corticon will iterate through a set of records in a database, retrieving them in "chunks", passing them to a Decision Service for processing, and saving updates to a database. This process continues until the full data set is processed. See *"Getting Started with Batch" in the Data Integration Guide*.

Set 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:/Program Files/Progress/cdd`

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

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

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

3

Corticon Server logs

Corticon Server writes log files to the `CORTICON_WORK/logs` folder by default. The default log configuration records information about the decision services that are loaded, all errors and exceptions that occur, and diagnostics on the performance of Corticon Server. All messages include date and time information as well as severity and type of message logged. For example:

```
2018-08-20 14:39:52.784 INFO DIAGNOSTIC
```

Logging functions have to be able to cover the range between getting enough information to understand how to resolve general processing issues, and not getting so much information that the time and space for server operations is compromised. In development environments, more detailed settings can be helpful, while production systems need to capture significant events yet be able to tolerate short-term application of detailed logging.

Logging server activities is an important part of administering Corticon Server deployments. It is also a feature of the built-in Server in Corticon Studios. You can set logs to be as detailed or as brief as your needs for information and performance change.

For details, see the following topics:

- How users typically use logs
- How to change logging configuration

How users typically use logs

Here are some ways you might use and manage logs.

- Configure logs to expose information:
  - Audit rule execution with log files (`logFiltersAccept` list includes `RULETRACE`)

    - Prevent sensitive data in log files (`logFiltersAccept` does not include `RULETRACE`)
    - Record diagnostic performance data (`logFiltersAccept` list includes `DIAGNOSTIC`), and then transform log data into CSV data for analysis tools

- Resolve problems:
    - Assess problems at server startup (Logs not created)
    - Assess problems with malformed requests
    - Assess problems with Corticon licensing
    - Assess problems with deployments
    - Assess problems with object translations

- Administer log files:
    - Produce Decision Service specific logs
    - Specify a preferred log path
    - Change retention policy for log file archives

How to change logging configuration

The default logging configuration enables basic logging without any need for further tailoring. This section describes the settings that let you adjust the configuration to suit your needs. The several properties that control logging features are described in the top section of the `brms.properties` file as these properties are often adjusted by users. These properties and how you can change them are described in two sections:

- **Log content** - The types and volume of information to be logged that are set in the `loglevel` and `logFiltersAccept` properties.
- **Log files** - The persistence techniques for log data that are set in the logpath, log rollover features, and option to log each Decision Service.

Configure logs

What gets entered into logs is up to you. There are two dimensions to what produces log content. The `loglevel` records ascending levels of operational information from nothing to everything. The `logFiltersAccept` let you control whether each information type created by each of the process reporting mechanisms is accepted into the logs. The resulting logs meld entries from both dimensions sequentially, and record all the information into log files.

Log Level

The `loglevel` specifies the depth of detail in standard logging. When set to `OFF`, no log entries are produced. Each higher level enables log entries triggered for that level, as well as each lower level. Set your preferred log level by uncommenting the line `# loglevel=` in `brms.properties`, and then setting the value to exactly one of:

- `OFF` - Turn off all logging
- `ERROR` - Log only errors
- `WARN` - Log all errors and warnings
- `INFO` - Log all info, warnings and errors (Default value)
- `DEBUG` - Log all debug information and all messages applicable to `INFO` level
- `TRACE` - Equivalent to DEBUG plus some tracing logs
- `ALL` - Maximum detail

Log Filters

The `logFiltersAccept` setting lets you include specified types of information emanating from running services in the logs. When the log level is set to `INFO` or higher, this property accepts logging of information types that are listed. Set your preferred log filters by uncommenting the line `# logFiltersAccept=` in `brms.properties`, and then listing functions you want to have in logs as comma-separated

values from the following:

- `RULETRACE` - Records performance statistics on rules
- `DIAGNOSTIC` - Records of service performance diagnostics at a defined interval (default is 30 seconds)
- `TIMING` - Records timing events
- `PAYLOAD` - Records the request payload and the response payload
- `VIOLATION` - Records exceptions
- `INTERNAL` - Records internal debug events
- `SYSTEM` - Records low-level errors and fatal events

The default `logFiltersAccept` setting is: `DIAGNOSTIC,SYSTEM`

Examples:

- Accept all:
  `logFiltersAccept=RULETRACE,DIAGNOSTIC,TIMING,PAYLOAD,VIOLATION,INTERNAL,SYSTEM`
- Accept none: `logFiltersAccept=`
- Accept just ruletracing, diagnostics, and timing: `logFiltersAccept=RULETRACE,DIAGNOSTIC,TIMING`

Configure log files

The files that record log entries can be relocated, created for each Decision Service, and archived for a specified number of cycles.

Log path

All log files are placed in a common location:

```
logpath=%CORTICON_WORK_DIR%/logs
```

The target folder can be changed to a preferred network-accessible location by uncommenting the line # `logpath=` in `brms.properties`, and then entering your location as the value. For example, on Windows::

```
logpath=H:/CorticonLogs/Server
```

If the folder structure does not exist, it will be created. You must use forward slashes as the separator; if you do not, the level preceding a backslash and all lower levels will be ignored.

Log for each Decision Service

The default logging approach is a single log for a running server. In server deployments, you can choose to maintain a log for each Decision Service. When Decision Service logging is enabled, log entries specific to a Decision Service are written only to that Decision Service's log file. Tp enable the feature, edit the `brms.properties` file to uncomment the following property and set it to `true`. The default value is `false`.

```
com.corticon.server.execution.logPerDS=true
```

When you save the file and restart the server, every transaction specific to a Decision Service is recorded in a file in the logs directory with the name pattern `Corticon-`*`DSname`*`.log`.

Archiving log histories

Logs 'rollover' on a regular basis, compressing each current `.log` file at the log path -- the general log and every Decision Service log -- into a separate dated archive. The default action is to do this. If you decide that you do not want to rollover or archive logs, uncomment the line `# logDailyRollover` in `brms.properties`, and then set the value to `false`.

When log rollover is in effect, the `logRolloverMaxHistory` setting specifies the number of rollover logs to keep. (If the server is not always running or has low traffic, this might not be a number of days). The default is five archives. You can set your preferred number of archives by uncommenting the line # `logRolloverMaxHistory=` in `brms.properties`, and then entering your preferred positive integer value. For example:

```
logRolloverMaxHistory=21
```

4

Troubleshooting Corticon Server

Corticon Server license

Whatever way you are deploying Corticon Server, you need to include your valid, active `CcLicense.jar` - Java web service installed directly or as a WAR file, .NET web service, Java in-process, or .NET in-process.

The basic Server license issues are as follows:

License not installed - The `CcLicense.jar` license file must be located in the same directory as your server installation's `CcServer.jar` file. In the default installation, `CcServer.jar` is located in `[CORTICON_HOME]\Server\tomcat\webapps\axis\WEB-INF\lib`, so ensure your valid license file is there.

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

License Invalid - If your license indicates that it is invalid, contact your Progress Corticon representative to obtain a valid license file. You get a notification in the log similar to:

```
Corticon Server license has expired. Please contact Customer Support to receive a valid
 key.
```

License capacity exceeded - License capacity is measured in several ways:

- Number of unique Decision Services that may run concurrently in Corticon Server. Make sure your license can support the total number of unique `.erf` files referenced by deployed `.cdd` files.
- Number of rules allowed for all Decision Services deployed. Make sure your license can support the total number of rules contained in all the deployed `.erf` files.

Log files

When problems occur with Corticon Server, the primary troubleshooting tool is the set of log files produced during Corticon Server operation, whether as Studio test server or as deployed Server.

By default, Corticon Server produces one log that records all the activities of running Decision Services at the specified log level. While higher detail levels produce a more comprehensive basis for analysis, the details of all running Decision Services will also generate detail information into the log. When diagnosing problems, you might want to use Corticon Server's ability to generate logs for each Decision Service so that you can produce detailed logs for each service.

The following procedure shows how to set the Server log to capture additional information..

Note: Consider backing up and then deleting all the log files and archives in the `\logs` folder so that you get only what is logged from your tests under the log settings. It is good to start with a fresh logs that record only the problematic transactions. The next time Corticon Server processes a transaction, a new log file will be created and entries recorded in it.

Setting logging

See Configure logs on page 16 for more information.

1. Edit the installation's `brms.properties` file
2. Change the `loglevel` to a level that should bring in the operational activities that will reveal the problem, perhaps `DEBUG`.
3. Change `logFiltersAccept` to list activity elements that you think will be relevant.
4. Save the file.
5. Stop, then restart the server.

Setting logging for each Decision Service

See Log for each Decision Service on page 18 for more information.

1. Edit the `brms.properties` file.
2. Change the value of `com.corticon.server.execution.logPerDS` to `true`.
3. Save the file.
4. Stop, then restart the server.

Examining log files

1. Once you have restarted the server, rerun your tests.
2. In a text editor, navigate to `[CORTICON_WORK_DIR]\logs` to open the appropriate current logfile: Studio's is `CcStudio.log`, Server's is `CcServer.log`, and individual Decision Service (*DSname*) logs are `Corticon-`*`DSname`*`.log`.
3. Look for the indicators of problems that are described in the following sections.

Logs not created

Logging does not start until the server is invoked. Even though started, as viewed in its startup window, logs are not initialized until an activity occurs.

However, if you have routed a Corticon Request to the server and no log is produced, it is likely that the invocation/request is not even reaching the server. The most common causes of a non-responsive (invocation produces no log file entry) Corticon Server include:

- Incorrect Corticon Server deployment. Review your deployment procedures to confirm the deployment files and paths.
- Incorrect Corticon Server invocation
  - **Incorrect URL** - If using a web services deployment, ensure the SOAP message is addressed correctly, and that no firewalls or port configurations prevent the SOAP message from reaching Corticon Server.
  - **Incorrect API** - Review the APIs available for Corticon Server invocation.

    Note: See the complete Java Server JavaDocs in Studio and Java Server installations at `[CORTICON_HOME]\JavaDoc`.

- Even though Corticon Server may not respond to an incorrect invocation, the host server or container (app server, web server, and similar) may respond either at a command line or log level. Check to see if the host server has responded to your invocation.

Response containing errors

The most common causes of erroneous Corticon Server responses include:

Problems with a request

Invalid syntax and misnamed targets are common problems with requests:

Poorly formed JSON request - Syntax errors in a JSON message generate an HTTP 500 error message in the web server window.

Message payload does not conform to service contract - Compare your message to the service contract to ensure compliance. Many third-party tools are available that automatically validate a document to its schema. Notice that if Corticon Server cannot even parse the inbound message, no entry will be made in Corticon Server's log. Instead, an error message is displayed.

Incorrect or missing Decision Service Name - Ensure the message's Decision Service Name attribute matches the name of the Decision Service as it was deployed by either a deployment descriptor file or an API method call.

Review Decision Service metadata

Corticon log files add metadata on each Decision Service as it is loaded so that you can confirm consistent loading of a Decision Service.

The metadata is recorded in the log for each Decision Service at every server startup, at every log rollover, and whenever a Decision Service is added, updated, or deleted.

Deployment Descriptor (`.cdd`) file problems

The following items are common Deployment Descriptor file problems:

Missing Deployment Descriptor (`.cdd`) file - The `.cdd` file is missing from the `\cdd` directory, or the taskname contained in the SOAP request message does not match any of the tasknames in any of the `.cdd` files deployed to Corticon Server. For example, the log might record:

```
...
|ERROR|com.corticon.service.ccserver.exception.CcServerDecisionService.notRegisteredException:
CcServerDecisionService.lookupCcServerPoolForExecution () Decision Service:
OrderProcess is not registered. Update failed. (Missing pool manager)
```

Missing \cdd directory - The default location of the `cdd` directory in a server installation is `[CORTICON_WORK_DIR]\cdd`. For example, the log might record:

```
java.rmi.RemoteException: Unexpected Error; nested exception is:
com.corticon.service.ccserver.exception.CcServerInvalidArgumnentException:
CcServer.loadDromCddDir (String)   Directory does not exist.
```

Object translation errors due to incorrect Vocabulary external name mappings

- External names mapped incorrectly
- External data types specified incorrectly
- ALL entities must be mapped, even those where all attributes are transient.

5

Performance and tuning

This section discusses aspects of Corticon Server performance.

For details, see the following topics:

- Rulesheet performance and tuning
- Server performance and tuning
- Optimize pool settings for performance
- Single machine configuration
- Capacity planning
- The Java clock
- Diagnose runtime performance

Rulesheet performance and tuning

Corticon Studio includes many features that help rule authors write efficient rules. A significant contributor to Decision Service (Ruleflow) performance is the number of rules (columns) in the component Rulesheets, so reducing this number may improve performance. Using the **Compression** tool to reduce the number of columns in a Rulesheet has the effect of reducing the number of rules, even though the underlying logic is unaffected. In effect, you can create smaller, better performing Decision Services by compressing your Rulesheets prior to deployment.

Server performance and tuning

Important: Before doing any performance and scalability testing when using an evaluation version of Corticon Server, check with Progress Corticon support or your Progress representative to verify that your evaluation license is properly enabled to allow unlimited concurrency. Failure to do so may lead to unsatisfactory results as the default evaluation license does not permit high-concurrency operation.

All Decision Services are stateless and have no latency; that is, they do not call out to other external services and await their response. Therefore, increasing the capacity for thread usage will increase performance. This can be done through:

- Using faster CPUs so threads are processed faster.
- Using more CPUs or CPU cores so more threads may be processed in parallel.
- Allocating more system memory to the JVM so there is more room for simultaneous threads.
- Distributing transactional load across multiple Corticon Server instances or multiple CPUs.

Server build properties

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

Specifies the amount of time in milliseconds that the Ant build processor will wait before automatically timing out.

Default value is 300000 (5 minutes).

```
com.corticon.BuildWaitTime=300000
```

Specifies whether the Decision Service compile process should include all the JARs that are in the same directory as the CcServer.jar in the Ant Compile Classpath. This may need to be set to true dependent on the type of Application Server the Decision Services are deployed on. Primary focus is to incorporate customer Business Objects in the Ant Classpath so that Listener Generation will succeed.

Default value is true.

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

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

Default value is true.

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

Compile option: Add the Rule Asset's Report to the compiled EDS file. By having the Report inside the EDS file, any user can get the report for a deployed Decision Service through an in-process or a SOAP call to the

Corticon Server. Including the Report in the EDS file will increase the EDS file significantly. Default value is false

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

Compile option: Add the Rule Asset's WSDL to the compiled EDS file. By having the WSDL inside the EDS file, any user can get the WSDL for a deployed Decision Service through an in-process or a SOAP call to the Corticon Server. Including the WSDL in the EDS file will increase the EDS file significantly. Default is false.

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

Compile option: This property lets you configure the memory settings that are used to compile the Rule Assets into an EDS file. Default is -Xms256m -Xmx512m.

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

---------- Option that will restrict certain types of Rule Messages from being posted to the output of an execution. There are 3 different properties to allow the user to select exactly what is returned to them from the execution. Default is false (for all three properties).

```
com.corticon.server.restrict.rulemessages.info=false
com.corticon.server.restrict.rulemessages.warning=false
com.corticon.server.restrict.rulemessages.violation=false
```

Server execution properties

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

Determines whether CDO association accessor ("getter") methods will return clones of their association HashSets. Normally, an association getter will return a direct reference to the association HashSet.

The default value (false) provides the best performance, particularly for EDC-enabled applications, because cloning an association HashSet can trigger unnecessary database I/O due to lazy-loading.

You can use this property to overcome ConcurrentModificationException errors which may arise when a Rulesheet has two aliases assigned to the same association, and that Rulesheet contains action statements that modify the association collection.

Note that this property only applies to N-to-many associations; for N-to-1 associations, the CDOs will always return a direct reference to the "singleton" HashMap.

Default value is false

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

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

The `lenientDateTimeFormat` sub-property does the following:

* If `false`, forces all dateTime values to Zulu format which is the XML standard

* If `true`, allows any dateTime format supported by Java to be used in the payload

Default for `ensureComplianceWithServiceContract` is `true` (sort)

Default for `lenientDateTimeFormat` is `false`

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

Specifies which implementation class to be used when a supported interface is used for an association inside the user's mapped Business Object. This is needed by the BusinessObject Listener class, which is compiled during Decision Service deployment.

Support interfaces include:

- java.util.Collection
- java.util.List
- java.util.Set

Default values are:

- java.util.Collection=java.util.Vector
- java.util.List=java.util.ArrayList
- java.util.Set=java.util.HashSet

```
com.corticon.cdolistener.collectionmapping=java.util.Vector
com.corticon.cdolistener.listmapping=java.util.ArrayList
com.corticon.cdolistener.setmapping=java.util.HashSet
```

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

Default value is true.

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

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

Default value is true

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

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

Default value is 1000

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

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

Possible values are:

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

Default value is NONE (for all types)

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

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

Default value is 180000 (180000ms = 3 minutes)

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

By default, the allocation algorithm allows the system to dynamically increase the number of execution threads beyond the usual maximum limit when the CcServer is underutilized, potentially improving performance during periods of low server load. This property will disable the algorithm so that Decision Services will not exceed the maximum limit even if the CcServer is underutilized.

Setting this property to `true` disables that behavior, ensuring that Decision Services will strictly adhere to the maximum thread limit, even if the server is not fully utilized.

Default value is false

```
com.corticon.ccserver.allocation.disable.underutilization.algorithm=false
```

Note: This property only takes effect if allocation is explicitly enabled. To enable thread allocation, the following property must be set to `true`.

Default value is `false`.

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

Timeout setting for how long an execution thread can be inside the Execution Queue. The time starts when the execution thread enters the Execution Queue and ends when it completes executing against a Decision Service. A CcServerTimeoutException will be thrown if the execution thread fails to complete in the allotted time.

The value is in milliseconds.

Default value is 180000 (180000ms = 3 minutes)

```
com.corticon.server.execution.queue.timeout=180000
```

Option that will restrict certain types of Rule Messages from being posted to the output of an execution. There are 3 different properties to allow the user to select exactly what is returned to them from the execution. Default is false (for all three properties)

```
com.corticon.server.restrict.rulemessages.info=false
com.corticon.server.restrict.rulemessages.warning=false
com.corticon.server.restrict.rulemessages.violation=false
```

Note: The rulemessage restrictions set in a server's `brms.properties` file but will not override settings in a CDD. Also, a payload can dynamically override these properties for each execution by adding execution properties to its payload.

Option that will restrict the CorticonResponse to only contain RuleMessages in it. Default is false

```
com.corticon.server.restrict.response.rulemessages.only=false
```

Option that lets the user to define how many Rule Messages will be returned from the execution of a Decision Service. This helps to prevent users from accidentally deploying a Decision Service with diagnostic Rule Messages posted when each Rule is fired.

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

Defines how many messages will be returned in the output

Default value is 5000

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

Defines which block of messages will be returned in the response. This allows the user to specify the 2nd, 3rd, or nth number of 1000 messages to be returned

Default value is 1 (the first block)

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

Option to specify to capture HTTP Headers information

Default value is true

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

Provides the option to not add the Entity Name as an xsi:type value for those new Entities that are create through Rules using the `.new` operator. This only applies to XML payloads.

Default value is true (Entity Name will be added as an xsi:type)

```
com.corticon.server.xml.newentities.addtype=true
```

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

Default value is true

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

Related to XML Translation. Base on this value, extra processing will ensure that the Incoming Document's Enities, Attributes, and Associations match the Namespaces defined in the Vocabulary. If the Vocabulary Entity or Attribute does not have an explicitly set Namespace value, the Element's Namespace must be the same as the Namespace for the WorkDocuments Element. If the Namespaces don't match, the Entity, Attribute, or Association will not be read into memory during execution. Also, if new Entities, Attributes, or Associations are added to the XML because of Rules, then explicitly set Vocabulary value will be used, otherwise the WorkDocument's Namespace will be used.

Default value is false.

```
com.corticon.xml.namespace.ignore=false
```

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

Default value is false

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

Default JDOM formatting setting when converting JDOM to a String:

- `PRESERVE` - All content is printed in the format it was created, no whitespace or line separators are added or removed.
- `TRIM_FULL_WHITE` - Content between tags consisting of all whitespace is not printed. If the content contains even one non-whitespace character, it is printed verbatim, whitespace and all.
- `TRIM` - Same as `TRIM_FULL_WHITE`, plus leading/trailing whitespace are trimmed.
- `NORMALIZE` - Same as `TRIM`, plus addition interior whitespace is compressed to a single space.

Default value is `NORMALIZE`.

```
com.corticon.format.documentToString=NORMALIZE
```

Option to append the Server Execution start and stop times to the CorticonResponse document after ICcServer.execute(String) or ICcServer.execute(Document) is performed.

Default value is false

```
com.corticon.ccserver.appendservertimes=false
```

Option to generate WSDL with the vocabulary filename in the "portType", "binding", and "service" names. This option is set in the brms.properties of a machine running corticonmanagement utilities.

Default value is false

```
com.corticon.servicecontracts.global.use.vocabulary.filename=false
```

For information about related properties

See:

- *"Setting Server build properties" in the Performance and tuning section.*
- *"Properties that tune service contract output" in the Deployment topics.*
- *"Properties that are incorporated into Decision Services" in the Deployment topics.*

Ability to allocate execution threads

The following Server properties let you adjust Server allocations:

- The Corticon Server takes each thread (regardless of which Decision Service the thread is executing against), and then adds the thread to the Execution Queue in a first-in-first-out strategy. While that generally satisfies most use cases, you might want more control over which Decision Services get priority over other Decision Services. For that, you first set the Server property `com.corticon.server.decisionservice.allocation.enabled` to `true`, and then set the maximum number of execution threads (`maxPoolSize`) for each specified Decision Service in the Execution Queue. Once you set the allocation on *every* Decision Service, the Server will try to maintain corresponding allocations of execution threads from the Decision Services inside the Execution Queue. Once the property is set to `true`, the Decision Services will allocate based on the `maxPoolSize` that was assigned when the Decision Service was deployed. You can then dynamically change a Decision Service's `maxPoolSize` when deployed using a CDD file by setting the value of the `PROPERTY_MAX_POOL_SIZE`property in the CDD. When the `CcServerMaintenanceThread` detects the change, it will update the Decision Service.

  Note: You might have circumstances where you want to disable allocation behavior, ensuring that Decision Services will strictly adhere to the maximum thread limit, even if the server is not fully utilized. Default value is `false`

  ```
  com.corticon.ccserver.allocation.disable.underutilization.algorithm=false
  ```

- In some cases, you might want to enable Decision Service level allocations to control the number of Decision Service instances that can be added to the queue at a particular time. This will cause prioritizing of one Decision Service over another, making more resources available to that type. To do this, set the property's value to `true`. Default value is `false`

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

- Once Decision Service allocation is turned on, prioritization of one Decision Service may occur in the Execution Queue. If a particular Decision Service is fully allocated in the Execution Queue, other execution threads for that Decision Service will have to wait until one of the allocated execution Threads completes its execution. The wait time, in getting into the Execution Queue varies, based on load and other Decision Service allocations. You can allow those waiting Threads to timeout if they wait longer than specified. A `CcServerTimeoutException` will be thrown if the execution thread fails to complete in the allotted time. The value is in milliseconds. Default value is `180000` (180000ms = 3 minutes)

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

Optimize pool settings for performance

When a Decision Service is deployed and allocation is enabled, the person responsible for deployment (typically an IT specialist) decides how many instances of the same Decision Service (Reactors) may run concurrently. This number establishes the maximum pool size for that particular Decision Service. Different Decision Services may have different pool sizes on the same Corticon Server because caller demand for different Decision Services may vary.

Choosing how large to make the pool depends on many factors, including the incoming arrival rate of requests for a particular Decision Service, the time required to process a request, the amount of other activity on the server box and the physical resources (number and speed of CPUs, amount of physical memory) available to the server box. A maximum pool size of one (1) implies no concurrency for that Decision Service. If you are evaluating Corticon, your license requires that you set the parameter to 1. See Multi-threading and Concurrency for more details.

The recommendations that follow are not requirements. High-performing Corticon Server deployments may be achieved with varying configurations dictated by the realities of your IT infrastructure. Our testing and field experience suggests, however, that the closer your configuration comes to these standards, the better Corticon Server performance will be.

Configuring the runtime environment revolves around a few key quantities:

- The number of CPUs in the server machine on which the Corticon Server is running.
- The number of wrappers deployed. The wrapper is the intermediary “layer” between the web/app server and Corticon Server, receiving all calls to Corticon Server and then forwarding the calls to the Corticon Server via the Corticon API set. The wrapper is the interface between deployment-specific details of an installation, and the fixed API set exposed by Corticon Server. A sample Servlet wrapper, `axis.war`, is provided as part of the default Corticon Server installation.
- The maximum pool size settings for each deployed Decision Service that has enabled allocation. These pool sizes are set in the Deployment Descriptor file (`.cdd`).

Single machine configuration

CPUs AND WRAPPERS

For optimal performance, the number of wrappers or Servlets deployed should never exceed the number of CPU cores on the server hardware, minus an allocation to support the OS and other applications resident on the server, including middleware. Typically, the number of these wrappers is controlled via a configuration file.

Note: The `CcServer.war` files are available from the Progress download site in the `PROGRESS_CORTICON_6.3_SERVER.zip` package. The unpackaged files are typically installed in the Corticon directory `[CORTICON_HOME]\Server\Containers\WAR`. Refer to the Progress Software web page Corticon Supported Platforms Matrix to review the currently supported UNIX/Linux platforms and brands of Application Servers. Also see the Corticon KnowledgeBase entry Corticon Server 7.x sample WAR installation for different Application Servers for detailed instructions on configuring Corticon Server on all supported platforms.

WRAPPERS AND POOLS

The number of wrappers should be greater than or equal to the number of available CPUs on the server. By default, the Execution Queue will be configured to match the number of available CPUs on the machine, but this can be overridden by changing the following Corticon property in the `brms.properties` file.

This property will be used by the CcServer to determine how many concurrent executions can occur across the CcServer. Ideally, this value will be set to the number Cores on the machine. By default, this value is set to 0, which means the CcServer will auto-detect the number of Cores on the server.

```
com.corticon.server.execution.processors.available=0
```

MAXIMUM POOL SIZES

This value in the Decision Service is utilized when the CcServer is configured with Decision Service Allocation turned on. In the `brms.properties` file, enter the following line:

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

That sets the allocation property to true, so that the CcServer can control how many incoming execution threads from each Decision Service will be added to the CcServer's Execution Queue. If additional execution threads come into for a particular Decision Service, and that Decision Service has already allocated its maximum to the Execution Queue, then the incoming execution thread waits until an execution thread for that Decision Service leaves the Execution Queue.

HYPER-THREADING

Hyper-threading is an Intel-proprietary technology used to improve parallelization of computations (doing multiple tasks at once) performed on PC microprocessors. For each processor core that is physically present, the operating system addresses two virtual processors, and shares the workload between them when possible. Field experience suggests that Hyper-threading does not allow doubling of wrappers or Reactors for a given physical CPU core number. Doubling wrappers or Reactors with the expectation that Hyper-threading will double capacity will result in core under-utilization and poor performance. We recommend setting wrapper and Reactor parameters based on the assumption of one thread per CPU core.

Capacity planning

In a given JVM, the Corticon Server and its Decision Services occupy the following amounts of physical memory:

State of Corticon Server	RAM required
Basic Corticon Server overhead with no Decision Services (excludes memory footprint of the JVM which varies by JDK version and platform)	25 MB
Load a single Decision Service from the Deployment Descriptor or `addDecisionService( )` method API.	~ 5 MB (this is the overhead for the Trade Allocation sample application with seven (7) Rulesheets, twenty-four (24) rules, five (5) associated entities)
Working memory to handle a single CorticonRequest	~ 1 MB (this is the overhead for the Trade Allocation sample application with seven (7) Rulesheets, twenty-four (24) rules, five (5) associated entities (steady-state usage)).

You may reduce the amount of memory required in a large system by dynamically loading and unloading specific Decision Services. This is especially relevant in resource-constrained handheld or laptop scenarios where only a single business transaction occurs at a time. After the first Decision Service is invoked, it is unloaded by the application and the second Decision Service is loaded (and so on). While this will be slower than having all Decision Services always loaded, it can address tight, memory-constrained environments. A compromise alternative would only dynamically load/unload infrequently used Decision Services.

The Java clock

When performance of Java applications needs to be measured in milliseconds, note that Java is dependent upon the operating system's internal clock. Operating systems vary in the precision that they track time . For example, many operating systems measure time in units of tens of milliseconds such that slight discrepancies might be seen between computer time and coordinated universal time (UTC).

Diagnose runtime performance

When performance issues arise, analyzing usage characteristics might reveal the performance bottlenecks. Corticon servers always start a diagnostic thread, and a snapshot of key diagnostic metrics is taken at a regular interval. The diagnostic data is forwarded to the logging system where they are recorded. You can then run a utility that extracts diagnostic lines, and transforms them to a standard comma-separated value format you can use in a data analysis product such as Tableau or Excel to create visualizations.

Properties that control diagnostic logging

The user can control the starting of the diagnostic thread, the intervals at which diagnostic snapshots are taken, and whether diagnostic lines are accepted into logs. Add or edit the following properties in the `brms.properties`:

- To enable (`true`) or disable (`false`) automatic startup and configuration of the server monitor thread when an `ICcServer` is created in the `CcServerFactory.` Default value is `true`.

  ```
  com.corticon.server.startDiagnosticThread=true
  ```

- To enable server diagnostic data collection (Data is only posted if service thread is running) Default value is `true`.

  ```
  com.corticon.server.EnableServerDiagnostics=true
  ```

- To specify the wait time in milliseconds of the Server Diagnostic Monitor. Default is `30000` - 30 seconds.

  ```
  com.corticon.server.DiagnosticWaitTime=30000
  ```

- To set the log level and/or filters to accept diagnostic entries. The default loglevel, `INFO`, and higher loglevels enable filters that are set to accept `DIAGNOSTIC` entries. Choosing a lower loglevel and/or removing `DIAGNOSTIC` from the `logFiltersAccept` list denies generated diagnostic data entry into the log.

Note: See the topic How to change logging configuration on page 16 for more information.

Example of diagnostic log entries for a server and four Decision Services

```
2015-05-13 14:02:34.831 INFO DIAGNOSTIC [CcDiagnosticsThread] Cc  -
    id=1431540154831,sthp=509.6875,shp=356.08567810058594,sex=0,stq=1564,
    sec=1564,sfc=0,saex=3.9801790281329925,sawt=0
2015-05-13 14:02:34.831 INFO DIAGNOSTIC [CcDiagnosticsThread] Cc  -
    id=1431540154831,ds=ProcessOrder.1.10,ec=1564,aex=3,awt=0,fc=0
2015-05-13 14:02:34.831 INFO DIAGNOSTIC [CcDiagnosticsThread] Cc  -
    id=1431540154831,ds=Candidates.1.14,ec=0,aex=0,awt=0,fc=0
2015-05-13 14:02:34.831 INFO DIAGNOSTIC [CcDiagnosticsThread] Cc  -
    id=1431540154831,ds=AllocateTrade.1.14,ec=0,aex=0,awt=0,fc=0
2015-05-13 14:02:34.831 INFO DIAGNOSTIC [CcDiagnosticsThread] Cc  -
    id=1431540154831,ds=Cargo.1.0,ec=0,aex=0,awt=0,fc=0
2015-05-13 14:03:04.842 INFO DIAGNOSTIC [CcDiagnosticsThread] Cc  -
    id=1431540184842,sthp=509.6875,shp=371.9812469482422,sex=0,stq=1564,
    sec=1564,sfc=0,saex=3.9801790281329925,sawt=0
```

To generate DIAGNOSTIC data into the server log:

The default Corticon Server settings generate diagnostic data and capture the diagnostic entries in the log.

1. In `brms.properties`, set the monitor interval so that unusual spikes of activity in a key metric (such as heap size) are captured. If the window is large, the heap size might go from normal to the server crashing with OutOfMemory exceptions without leaving any trail in the diagnostics entries in the log. The default value is 30000 milliseconds. You could change it to, say, 10 seconds by locating the line: `#com.corticon.server.DiagnosticWaitTime=30000`and then changing its value to `com.corticon.server.DiagnosticWaitTime=10000`
2. Confirm that the `logLevel` is `INFO` or higher and that `logFiltersAccept` includes `DIAGNOSTICS`.
3. Save the file.
4. Start Corticon Server.
5. Execute some requests or activities through the server.
6. Examine the server log at `[CORTICON_WORK_DIR]\logs\` to note lines that contain "`DIAGNOSTIC`".

You can now run the utility that extracts only the diagnostic lines and transforms each from *name=value* pairs to comma-separated integer and string values. The Server and Decision Service Diagnostic data and procedures are discussed separately.

SERVER DIAGNOSTICS

The server diagnostic log entries provide general server health metrics. The following metrics are logged:

Table 1: Content of a Server diagnostic entry

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

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

Once you have a log file with diagnostic entries, running a Corticon management utility takes an input file and produces each `DIAGNOSTIC` line as CSV data in its output file. To run the utility against a log that has been archived, extract the log file to a temporary location.

To extract server diagnostic data from a Corticon Server log:

1. Open a command prompt window at `[CORTICON_HOME]\Server\bin`.
2. Enter:

   ```
   corticonManagement.bat -e -s -i {input_file}  -o {output_file}
   ```

   For example:

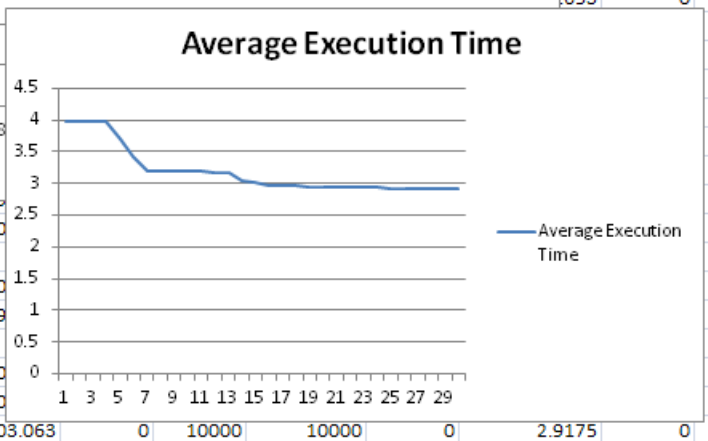
   ```
   corticonManagement.bat -e -s -i C:\CcServer.log  -o C:\CcServer_20150513.csv
   ```

When the processing completes, the input file is unchanged. The output file extracts only Server diagnostic lines, transforming each line into CSV values and a header line as shown for the log example above:

```
Date Time,Heap Size,Total Heap Size,Thread Pool Size,Tasks Queued,
    Executions,Failures,Average Execution Time,Average Wait Time
2015-05-13 14:03:04.842,371.9812469482422,509.6875,0,1564,1564,0,3.9801790281329925,0
2015-05-13 14:03:34.854,328.77896881103516,509.75,0,1564,1564,0,3.9801790281329925,0
2015-05-13 14:04:04.827,342.8452453613281,509.8125,0,1564,1564,0,3.9801790281329925,0
2015-05-13 14:04:34.829,364.4458999633789,501.1875,0,2020,2020,0,3.704950495049505,0
2015-05-13 14:05:04.831,372.962158203125,503.625,0,2879,2879,0,3.4105592219520666,0
2015-05-13 14:05:34.833,345.2066192626953,504.875,0,4335,4335,0,3.200461361014994,0
2015-05-13 14:06:04.835,366.8616409301758,503.4375,0,4335,4335,0,3.200461361014994,0
2015-05-13 14:06:34.837,345.76478576660156,502.9375,0,4335,4335,0,3.200461361014994,0
2015-05-13 14:07:04.839,358.3552551269531,503.5,0,4448,4448,0,3.1913219424460433,0
2015-05-13 14:07:34.841,337.6990051269531,495.75,0,4448,4448,0,3.1913219424460433,0
```

The Server Diagnostic CSV data is compatible with analytic and visualization products such as Excel and Tableau, as illustrated:

	A	B	C	D	E	F	G	H	I
1	*Date Time*	*Heap Size*	*Total Heap Size*	*Thread Pool Size*	*Tasks Queued*	*Executions*	*Failures*	*Average Execution Time*	*Average Wait Time*
2	02:34.8	356.0856781	509.688	0	1564	1564	0	3.980179028	0
3	03:04.8	371.9812469	509.688	0	1564	1564	0	3.980179028	0
4	03:34.9	328.7789688	509.75	0	1564	1564	0	3.980179028	0
5	04:04.8	342.8452454	509.813	0	1564	1564	0	3.980179028	0
6	04:34.8	364.4459	501.188	0	2020	2020	0	3.704950495	0
7	05:04.8	372.9621582	503.625	0	2879	2879	0	3.410559222	0
8	05:34.8	345.2066193	504.875	0	4335	4335	0	3.200461361	0
9	06:0							.361	0
10	06:3							.361	0
11	07:0							.942	0
12	07:3							.942	0
13	08:0							.654	0
14	08:3							.654	0
15	09:0							.524	0
16	09:3							.653	0
17	10:0								
18	10:3								
19	11:0								
20	11:3								
21	12:0								
22	12:3								
23	13:0								
24	13:35.0	368.4134293	50						
25	14:05.0	350.557579							
26	14:35.0	392.2212219	50						
27	15:05.0	368.0653381	49						
28	15:35.0	350.1802902							
29	16:05.0	377.8211441	50						
30	16:35.0	361.4021835	50						
31	17:05.0	388.716217	503.063	0	10000	10000	0	2.9175	0



DECISION SERVICE DIAGNOSTICS

A diagnostic entry is created for each Decision Service that is deployed to the Corticon server. If you are creating separate logs for each Decision Service, the utility runs against its corresponding log; otherwise, the Decision Service diagnostic is captured into the server log where the utility will, in turn, extract the data for one specified Decision service at a time against the same server log. The following metrics are logged for each Decision Service.

Table 2: Content of a Decision Service diagnostic entry

Item	Description
Diagnostic set ID (`id`)	Grouping of log lines from a common time slice
Date Time	Timestamp of log entry
Decision Service name (`ds`)	The name and Major.minor version of the deployed Decision Service
Execution Count (`ec`)	The total execution count for this Decision Service since the Corticon server startup
Average Execution time (`aex`)	Average execution time in the designated Decision Service since the Corticon server startup

Item	Description
Average Wait Thread time (`awt`)	The Average execution wait time for threads entering the designated Decision Service since the Corticon server startup
Failure Count (`fc`)	Number of failures recorded in the designated Decision Service since the Corticon server startup

Once you have a log file with diagnostic entries, running a Corticon management utility takes an input file and produces each `DIAGNOSTIC` line as CSV data in its output file.

To extract Decision Service diagnostic data from a Corticon Server log:

1. Open a command prompt window at `[CORTICON_HOME]\Server\bin`.
2. Enter:

```
corticonManagement.bat -e -ds {DSname} -dsv {major.minor} -i {input_file}  -o
{output_file}
```

For example:

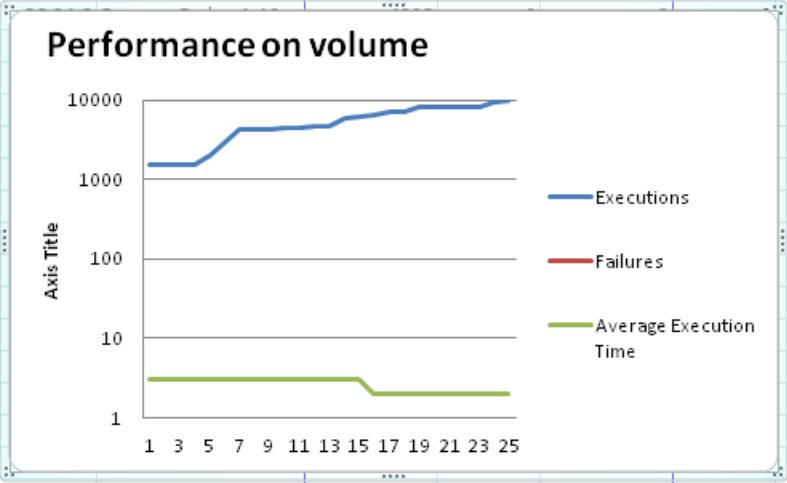
```
corticonManagement.bat -e -ds ProcessOrder -dsv 1.10
        -i C:\CcServer.log  -o C:\ProcessOrder_1.10_20150513.csv
```

When the processing completes, the input file is unchanged. The output file extracts only diagnostic lines, transforming each line into CSV values and a header line as shown for the log example above:

```
Date Time,Service Name,Executions,Failures,
  Average Execution Time,Average Wait Time
2015-05-13 14:02:34.831,ProcessOrder.1.10,1564,0,3,0
2015-05-13 14:03:04.842,ProcessOrder.1.10,1564,0,3,0
2015-05-13 14:03:34.854,ProcessOrder.1.10,1564,0,3,0
2015-05-13 14:04:04.827,ProcessOrder.1.10,1564,0,3,0
2015-05-13 14:04:34.829,ProcessOrder.1.10,2020,0,3,0
2015-05-13 14:05:04.831,ProcessOrder.1.10,2879,0,3,0
2015-05-13 14:05:34.833,ProcessOrder.1.10,4335,0,3,0
2015-05-13 14:06:04.835,ProcessOrder.1.10,4335,0,3,0
2015-05-13 14:06:34.837,ProcessOrder.1.10,4335,0,3,0
2015-05-13 14:07:04.839,ProcessOrder.1.10,4448,0,3,0
```

The Decision Service Diagnostic CSV data is compatible with analytic and visualization products, as illustrated in Excel:

	A	B	C	D	E	F
1	*Date Time*	*Service Name*	*Executions*	*Failures*	*Average Execution Time*	*Average Wait Time*
2	02:34.8	ProcessOrder.1.10	1564	0	3	0
3	03:04.8	ProcessOrder.1.10	1564	0	3	0
4	03:34.9	ProcessOrder.1.10	1564	0	3	0
5	04:04.8	ProcessOrder.1.10	1564	0	3	0
6	04:34.8	ProcessOrder.1.10	2020	0	3	0
7	05:04.8	ProcessOrder.1.10	2879	0	3	0
8	05:34.8	ProcessOrder.1.10	4335	0	3	0
9	06:04.8	ProcessOrder.1.10	4335	0	3	0
25	14:05.0	ProcessOrder.1.10	9549	0	2	0
26	14:35.0	ProcessOrder.1.10	10000	0	2	0



Interpreting diagnostic data

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

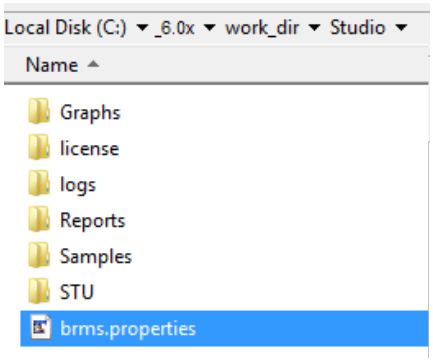
- When the number of waiting threads (`wt`) goes up, it is an indication that the request demand is greater than the server capacity. Some wait time may be necessary in periods of high demand.
- The average wait time (`awt`) can be used to determine whether server capacity should be expanded (or if consistently low, might indicate contracting server resources).
- The number of executions (`ex`) combined with the average wait time help pinpoint whether there is a need to expand server resources or just to accept a slower response in small, high demand windows.
- The number of failures (`fl`) is an indication that expert analysis/maintenance is needed.
- The average execution time (`aex`) can be used to determine if there are configuration/resource issues. If this rate is not stable, it might indicate that the resource configuration is not optimal. However, this value can be dependent upon data size -- if the input data size is not stable the execution size will not be stable.

A

Server Properties and settings

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

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



About the brms.properties file

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

Enabling settings listed in the default brms.properties file

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

To specify a preferred value for a listed property, edit the file, remove the # from the beginning of a property's line, and then add your preferred value after the equals sign. For example, to express a preference for decimal values displayed and rounded to two places instead of the six places preset for this property, locate the line:

```
#decimalscale=6
```

and then change it to

```
decimalscale=2
```

Adding unlisted settings to brms.properties file

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

For example, to change interval of diagnostic readings from five minutes to two minutes, add the following line to the `brms.properties` file -- it does not matter where in the file as long as it is on a separate line:

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

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

Saving and applying the revised Server property settings

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

However, Server requires that the file named `brms.properties`, and is located at the root of the work directory. You can copy an alternative properties file to rename it, and then paste into the standard location with the expected name.

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

Note: Property settings you list in your `brms.properties` *replace* corresponding properties that have default settings. They do not *append* to an existing list. For example, if you want to add a new `DateTime` mask to the built-in list, be sure to include *all* the masks you intend to use, not just the new one. If your `brms.properties` file contains only the new mask, then it will be the only mask Corticon uses.

Note: Adminstrative APIs for certain transient property settings - A running instance of Corticon Server can modify certain properties through API method calls. While settings in `brms.properties` persist across Corticon Server sessions, changes applied through APIs only remain in effect for that Corticon Server session. When Corticon Server starts a new session, it will use its default settings and apply the `brms.properties` file.

For details, see the following topics:

Settable properties described in context

Setting properties that change behaviors in the Corticon products are put into effect as line items in the `brms.properties` file that is installed at the `[WORK_DIR]` root. For information about how that file is handled, see Server Properties and settings on page 39.

The documentation describes the properties you can set in:

- Dynamic discovery of new or changed Decision Services on page 13
- Server state on page 12
- *"How to terminate infinite loops" in the Rule Modeling guide.*
- Single machine configuration on page 31
- Set Server startup to auto load CDD files on page 14
- *"How to relax enforcement of custom data types" in the Rule Modeling guide*. Also see *"Use Corticon Studio to reproduce the behavior" in the Rule Modeling guide.*
- *"Specify server URLs to access test subjects" in the Quick Reference guide.*
- *"Corticon Server files and API Tools" in the Web Services guide.*
- *"Trace Rule Execution" in the Rule Modeling guide.*
- Mapping XML and JSON topics under *"How to prepare Studio files for deployment" in the Deployment guide.*
- *"Properties that tune service contract output" in the Deployment guide.*
- *"Extended service contracts: newOrModified" in the Deployment guide.*
- Diagnose runtime performance on page 33
- Configure log files on page 17
- *"How to set properties in a CDD file" in the Deployment guide.*
- *"Studio properties and settings" in the Rule Modeling guide.*
- *"Formats for Date, Time, and DateTime properties" in the Rule Language guide.*
- Server execution properties on page 25
- Server registration with Web Console on page 42
- *"Properties that impact Decision Service compilation" in the Deployment guide.*

- Server build properties on page 24
- Web Console server properties on page 44
- *"Properties that are incorporated into Decision Services" in the Deployment guide.*

Server registration with Web Console

When using the Web Console, you might want the servers to have the ability to register automatically, and even update that registration when their IP address changes. In an elastic environment, new server instances would not be recognized by a Web Console. New server instances would be added manually to a Web Console's Server Group. But servers might assigned a new IP address. When each server can connect to its Web Console and provide its revised IP address, the Web Console will be able to automatically report this server's metrics.

The following properties are settings you can apply on Corticon Server installations that will be managed by the Web Console by adding all the properties and appropriate values as lines in each server's `brms.properties` file, and then restarting each Server to apply the properties. The effect of these settings will be to register the server in the Web Console if it is not already registered. The Web Console Server must be running and accessible at its URL.

Web console values are defaulted to the out-of-the-box Tomcat configuration of the Web Console installation.

If a server is not already registered, the server will register itself with the specified Web Console based on the dynamic strategy . If a server is already registered and its server IP has changed, the auto-register will update the previous registration with the new IP and not register it again.

Enable the feature—When this is `false` the feature will not use any of its other properties.

`com.corticon.ccserver.autoregistration.enabled`

Indicates whether this Server will try to auto-register with a Web Console. Server instances that are already registered will not be updated. To update an existing registration (say, to change password), stop the server and delete it from the Web Console, and start again with this property set to `true`.

Default: `false`

Connection to the Web Console—The properties for managing connection to the Web Console server, shown here as `localhost` on port `8850`. Note that if you use `localhost` for a co-located installation, the Web Console is self-registered for such an installation and will create another server registration using the IP address

`com.corticon.ccserver.autoregistration.webconsoleserver.url`

The URL of the running Web Console server.

Default: `http://localhost:8850/corticon`

Note: Host and Port—Substitute the configured values of your Web Console server location and its port when configuring server registration.

`com.corticon.ccserver.autoregistration.webconsoleserver.username`

Username to use to authenticate with the Web Console. To encrypt the username, use the utility `corticonManagement --encryptstring -i` *`usernname`*.

Default (encrypted): `047043014032058`

`com.corticon.ccserver.autoregistration.webconsoleserver.password`

Password to use to authenticate with Web Console. To encrypt the password, use the utility `corticonManagement --encryptstring -i` *`password`* .

Default: `047043014032058`

`com.corticon.ccserver.autoregistration.webconsoleserver.retries`

The number of retries the Server will attempt to login and register itself to the Web Console.

Default: `20`

`com.corticon.ccserver.autoregistration.webconsoleserver.sleep`

The amount of time to wait in between retry attempts as the Web Console could be initializing as its Server starts up.

Default: `10000` (10 seconds)

Registered type of the server

`com.corticon.ccserver.autoregistration.webconsoleserver.servergroup`

The name of the server group that this Server will create or be registered in.

Default: `Auto Registered`

`com.corticon.ccserver.autoregistration.webconsoleserver.singleserver`

When value is `true`, the Server is be added as a single server and not added to a Server Group, although it might the sole member of a group.

Default: `true`

Registered identity of the server

`com.corticon.ccserver.autoregistration.local.protocol`

The protocol that the Web Console will use to call into this Server (or `https`).

Default: `http`

`com.corticon.ccserver.autoregistration.local.ip.static`

The static IP Address that the Web Console will use to call into this Server. If this value is null, then the machine's IP Address or DNS Name will be used, which depending on the value of the `com.corticon.ccserver.autoregistration.local.ip.dynamic.strategy` property.

Default:

`com.corticon.ccserver.autoregistration.local.ip.dynamic.strategy`

If the `com.corticon.ccserver.autoregistration.local.ip.static` is null, then the Server will register itself based on the machine's IP or its DNS Name. This property allows you to determine which approach to use. Possible values are IP or DNS.

Default: `IP`

`com.corticon.ccserver.autoregistration.local.port`

The port that the Web Console will use to call into this Server

Default: `8850`

`com.corticon.ccserver.autoregistration.local.context`

The name of the context that the Web Console will use to call into this Server

Default: `axis`

`com.corticon.ccserver.autoregistration.local.username`

Username to be used by Web Console to authenticate with this running Server. To encrypt the password, use the utility `corticonManagement --encryptstring -i` *`password`*.

Default: `047043014032058`

`com.corticon.ccserver.autoregistration.local.password`

Password to be used by Web Console to authenticate with this running Server. To encrypt the password, use the utility `corticonManagement --encryptstring -i` *`password`*.

Default: `047043014032058`

In summary

The properties to add to your Server's `brms.properties` file with your preferred values are the following, shown with default values or examples:

```
com.corticon.ccserver.autoregistration.enabled=false

com.corticon.ccserver.autoregistration.webconsoleserver.url=http://localhost:8850/corticon
com.corticon.ccserver.autoregistration.webconsoleserver.username=047043014032058
com.corticon.ccserver.autoregistration.webconsoleserver.password=047043014032058
com.corticon.ccserver.autoregistration.webconsoleserver.retries=20
com.corticon.ccserver.autoregistration.webconsoleserver.sleep=10000

com.corticon.ccserver.autoregistration.webconsoleserver.servergroup=Auto Registered
com.corticon.ccserver.autoregistration.webconsoleserver.singleserver=true

com.corticon.ccserver.autoregistration.local.protocol=http
com.corticon.ccserver.autoregistration.local.ip.static=
com.corticon.ccserver.autoregistration.local.ip.dynamic.strategy=IP
com.corticon.ccserver.autoregistration.local.port=8850
com.corticon.ccserver.autoregistration.local.context=axis
com.corticon.ccserver.autoregistration.local.username=047043014032058
com.corticon.ccserver.autoregistration.local.password=047043014032058
```

Web Console server properties

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

Property related to monitoring execution times of Decision Service - Version over defined

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

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

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

Default value is `true`

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

Property related to Decision Service - Version level monitoring. Specifies whether the Server will auto-start recording time measurements.

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

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

Default value is `true`

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

B

Corticon Server API

Corticon provides a rich API for managing and monitoring a Corticon Server. This API is used by the Corticon Web Console to manage and monitor Corticon Servers. When Corticon is deployed in-process or deployed as a web service and you want custom access, you can use the Corticon Server API, accessible online at https://documentation.progress.com/output/Corticon/7.1/javadoc/Server/index.html .

For details, see the following topics:

- Server in-process API
- Server REST API

Server in-process API

When deployed in-process, you use the Corticon Server API to create an instance of Corticon Server, deploy and manage Decision Services, and to execute Decision Services on your data payloads. See the *Corticon Tutorial Deploying a Progress Corticon Decision Service inProcess for Java* for an introduction to using Corticon in-process. See the Corticon Server API JavaDoc for full details of the in-process API.

Note: The API defined in the JavaDoc is available in proxy DLLs when calling Corticon from a .NET application.

Server REST API

Overview

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

Note: REST API online documentation—The REST API in static format is available in Java Server's install directory for Java and .NET (but not propagated to the IIS Server), and at https://documentation.progress.com/output/Corticon/7.1/RESTDoc/

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

Error handling in the REST Management API

Error handling

Error handling is done through HTTP error codes and appropriate error objects.

Error Objects

A failed API call will place an object in the entity similar to this:

```
{
  "error"
  {"type" : The java type of the exception that was thrown to generate this error,
   "parentError" : The nested error object that was the cause of thisexception
                        (This field is included only when this error has a nested
error),
   "message" : The message from this error code,
   "stackTrace" : This array contains lines of a stacktrace, each will correspond
                              to a stackframe or an exception label if there are
nested exceptions
   [...]
   }
}
```

Common error behavior

All REST urls defined have a common error behavior. An error response returned by the server consists of two parts:

1. A HTTP status code other than 200.
2. A JSON Object in the response payload containing the error property.

Common HTTP status codes

The server can return following codes:

- `200 OK` The request was processed successfully.
- `400 Bad request` An incorrect request.
- `500 Internal Server Error` The server encountered an error while processing the request.

How to access the Vocabulary metadata of a Decision Service

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

Structure of a request

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

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

For example:

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

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

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

```
{
   "majorVersion": 1,
   "name": "MetadataTest",
   "minorVersion": 0,
   "entities": [
      {
         "name": "Entity_1",
         "associations": [
            {
               "name": "associationObject1",
               "targetEntity": "AssociationObject1",
               "inContext": true,

               "mandatory": false
               "cardinality": "1"
            },
            {
               "name": "associationObjectOverride",
               "targetEntity": "AssociationObject2",
               "inContext": false,
               "mandatory": true
               "cardinality": "*"
            }
         ],
         "attributes": [
            {
               "dataType": "Boolean",
               "name": "boolean1",
               "inContext": true,
               "type": "Base",
               "mandatory": true
            },
            {
               "dataType": "Date",
               "name": "date1",
               "inContext": false,
               "type": "Transient",
               "mandatory": false
            },
            {
```

```
            "dataType": "DateTime",
            "name": "datetime1",
            "inContext": true,
            "type": "Base",
            "mandatory": true
         },
         {
            "cdtConstraintExpr": "value < 100.0",
                    (Constraint expression associated with this Attribute.)
            "dataType": "Decimal",
            "name": "decimal1",
            "inContext": false,
            "type": "Transient",
            "mandatory": false
         },
         {
            "dataType": "Integer",
            "name": "int1",
            "cdtEnumeration": [   (A CDT that is values only, no labels.)
              {"value": "1"},
              {"value": "2"},
              {"value": "3"},
              {"value": "4"}
            ],
            "inContext": true
            "type": "Base",
            "mandatory": true
         },
         {
            "dataType": "String",
            "name": "string1",
            "cdtEnumeration": [   (A CDT that has labels and values.)
              {
                 "value": "s"
                 "label": "Small"
              },

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

```
            ]
         },
         {
            "name": "AssociationObject2",
            "associations": [],
            "attributes": [
               {
                  "dataType": "String",
                  "name": "string1",
                  "inContext": false
                  "mandatory": false
               },
            ]
         }
      ]
   }
```