CentraSite

Run-Time Governance Reference

Version 9.6

April 2014

This document applies to CentraSite Version 9.6.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: IINM-DG-ACTIONSR-96-20140318

Table of Contents

Preface

This document describes the run-time events and performance metrics, as well as the run-time actions that you can apply to virtual services or APIs.

The content is organized under the following sections:

Run-Time Events and Key Performance Indicator (KPI) Metrics	Describes: ■ The run-time events and Key Performance Indicator (KPI) metrics that can be collected and reported for each virtual service deployed in your system. ■ How to configure CentraSite to receive the events and metrics from the policy-enforcement point (such as Mediator) that collects them.
Built-In Run-Time Actions Reference for Virtual Services	You use these actions only when you are using CentraSite Control to create run-time policies for virtual services. This section provides: ■ A summary of the run-time actions. ■ An alphabetic reference of all actions and their parameters. ■ A listing of the action evaluation order and action dependencies. ■ Some common combinations of actions used to authenticate/identify consumers.
Built-In Run-Time Actions Reference for APIs	You use these actions only when you are using the CentraSite Business UI to create policy enforcement rules for APIs. This section provides an alphabetic reference of all actions and their parameters.
Computed Runtime Actions	Describes how to write a computed action and integrate it into the CentraSite environment.

1 Run-Time Events and Key Performance Indicator (KPI) Metrics

CentraSite can receive run-time events and Key Performance Indicator (KPI) metrics. A run-time event is an event that occurs while services are actively deployed on the target. Examples of run-time events include:

- Successful or unsuccessful SOAP requests/responses.
- Policy violation events, which are generated upon violation of service's run-time policy.
- Service monitoring events, which are generated by the service-monitoring actions in the run-time policy.

KPI metrics are used to monitor the run-time execution of virtual services. Metrics include the maximum response time, average response time, fault count, availability of virtual services, and more. If you include run-time monitoring actions in your run-time policies, the actions will monitor the KPI metrics for virtual services, and can send alerts to various destinations when user-specified performance conditions for a service are violated.

CentraSite provides predefined event types for use with any supported policy-enforcement point (PEP), such as webMethods Mediator. In addition, you can create custom event types.

The run-time event data are collected by the PEP and published to CentraSite via SNMP. The PEP publishes data for all run-time events for all instances of the PEP target.

You can view the run-time events and metrics on the CentraSite Control user interface. You can view them for all targets, for a particular target, or for a particular virtual service.

The following topics are discussed:

The Run-Time Event Types

The types of run-time events that Mediator can publish are as follows:

Event Type	Description
Lifecycle	A Lifecycle event occurs each time Mediator is started or shut down.
Error	An Error event occurs each time an invocation of a virtual service results in an error.
Policy Violation	A Policy Violation event occurs each time an invocation of a virtual service violates a run-time policy that was set for the virtual service.
Transaction	A Transaction event occurs each time a virtual service is invoked (successfully or unsuccessfully).
Monitoring	Mediator publishes key performance indicator (KPI) metrics, such as the average response time, fault count, and availability of all virtual services (described below).

The Key Performance Indicator (KPI) Metrics

For the Monitoring event type, Mediator can publish the following types of KPI metrics:

Metric	Reports on...
Availability	The percentage of time that a virtual service was available during the current interval. A value of 100 indicates that the service was always available. Only the time when the service is unavailable counts against this metric. If invocations fail due to policy violations, this parameter could still be as high as 100.
Average Response Time	The average amount of time it took the service to complete all invocations in the current interval. This is measured from the moment Mediator receives the request until the moment it returns the response to the caller.
Fault Count	The number of failed invocations in the current interval.
Maximum Response Time	The maximum amount of time it took the service to complete an invocation in the current interval.
Minimum Response Time	The minimum amount of time it took the service to complete an invocation in the current interval.
Successful Request Count	The number of successful service invocations in the current interval.
Total Request Count	The total number of requests for each service running in Mediator in the current interval.

Note: By default, Average Response Time, Minimum Response Time and Maximum Response Time do not include metrics for failed invocations. You can include metrics for failed invocations by setting the pg.PgMetricsFormatter.includeFaults parameter to true. For more information, see the section *Advanced Settings* in the document *Administering webMethods Mediator*.

The Event Notification Destinations

Mediator can publish data about the run-time events and metrics to the following destinations:

- An SNMP server. You can use one or both of the following kinds of servers:
  - CentraSite's SNMP server, which uses SNMPv3 user-security model.

    For the procedure to configure Mediator to send SNMP traps to the CentraSite SNMP server, see the section *SNMP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.
  - A third-party SNMP server, which uses either the SNMPv1 community-based security model or the SNMPv3 user-based security model.

For the procedure to configure Mediator to send SNMP traps to a third-party SNMP server, see the section *SNMP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.

- An EDA destination. Mediator can use EDA to publish run-time events and metrics to a database. Mediator uses a JDBC connection pool that you need to define in the Integration Server.

  For the procedure to configure Mediator to send this data to an EDA destination, see the section *EDA Configuration for Publishing Run-Time Events and Metrics* in the document *Administering webMethods Mediator*.

Destinations for the Monitoring and Transaction Events

For the Monitoring and Transaction event types, there are additional event notification destinations to choose from (in addition to the EDA and SNMP destinations).

Monitoring events are generated by the following run-time actions that you can configure for your virtual services in CentraSite:

- Monitor Service Performance.
- Monitor Service Level Agreement.
- Throttling Traffic Optimization.

Transaction events are generated by the run-time action Log Invocations.

The available destinations for Monitoring and Transaction events are:

- An EDA destination (a database).
- The CentraSite SNMP server or a third-party SNMP server.
- The virtual service's Events profile in CentraSite.
- An SMPT email server.
- Your Integration Server's local log.
- Your Integration Server's audit log (for Transaction events only).

You will select these destinations when you configure your virtual services in CentraSite.

These additional destinations for the monitoring and transaction events are described below.

SMTP Email Servers

To specify an SMTP email destination, you must:

- Select the "Email" option as a destination when you configure the run-time actions listed above.
- Set the "Email Configuration" parameters in Integration Server Administrator (go to **Solutions > Mediator > Administration > Email**) as described in the section *SMTP Destinations for Alerts and Transaction Logging* in the document *Administering webMethods Mediator*.

The Integration Server's Local Log

To specify the Integration Server's local log as a destination, you must:

- Select the "Local Log" option as a destination when you configure the run-time actions listed above. When configuring the actions, you must also specify the severity of the messages to be logged (the logging level).
- Set the Integration Server Administrator's logging level for Mediator to match the logging levels specified for the run-time actions (go to **Settings > Logging > Server Logger**). For example, if a "Log Invocation" action is set to the logging level of Error, you must also set Integration Server Administrator's logging level for Mediator to Error. If the action's logging level is set to a low level (Warning-level or Informationlevel), but Integration Server Administrator's logging level for Mediator is set to a higher level (Error-level), then only the higher-level messages are written to the log file.

  Entries posted to the local log are identified by a product code of MED.

The Integration Server's Audit Log

You can select the Integration Server Audit Log as a destination for the "Log Invocation" action only. If you expect a high volume of invocations in your system, it is recommended that you select the Audit Log destination. For more information, see the *webMethods Audit Logging Guide*.

The Metrics Tracking Interval

Mediator tracks performance metrics by intervals. The interval is a period of time you set in Mediator, during which metrics are collected for reporting to CentraSite. You set the interval in the Publish Interval field on the **Mediator > Administration > CentraSite Communication** page in the Integration Server Administrator. For details, see the section *Configuring Communication with CentraSite* in the document *Administering webMethods Mediator*.

Mediator only tracks metrics for the current interval. At the end of the interval, Mediator aggregates the metrics and reports them to CentraSite. Once the metrics are reported, Mediator resets its counters for the new interval. Mediator does not calculate and aggregate metrics across intervals. If Mediator is shut down or the virtual service is undeployed before the current interval expires, the performance data is discarded.

Note: To avoid the need for Mediator to store metrics during periods of inactivity, Mediator stores only first and last zero value metrics that occurs during an interval, and discards the remaining consecutive zero value metrics. Doing this drastically reduces the storage space consumed by the metrics, and speeds the queries you perform in the dashboard. Skipping the in-between zero metrics will not affect in the performance graphs shown in the dashboard.

For more information about the metrics tracking interval, see the section *Key Performance Indicator Metrics and Run-Time Event Notifications* in the document *Administering webMethods Mediator*.

Configuring CentraSite to Receive Run-Time Events and Metrics

Prerequisites:

- Ensure that Mediator is configured for publishing events to an SNMP server, as described in the section *SNMP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.
- If you use a target type other than Mediator or webMethods Insight, be sure to configure CentraSite to publish events by providing your own MIB file in your target type's definition file, as described in the section *Run-Time Targets*. (CentraSite provides a MIB file for Mediator and Insight.)
- Optionally change CentraSite's default settings for logging run-time events, as described in the section *Logging*. By default, CentraSite logs all predefined event types, but you may disable any type.

CentraSite provides an Event Receiver, which is a data collector that collects the run-time event data. The Event Receiver listens for run-time events from the target instances via the SNMP (Application-Layer) protocol, and contains the logic to parse and store event data in the Event Receiver's data store. You must configure the Event Receiver's properties file as described below.

This section includes the following topics:

- Event Modeling

Components of the Event Receiver

The Event Receiver contains the following components.

- **The SNMP Listener**

  CentraSite's SNMPv3 Trap Listener, which supports **SNMP4J**. This Listener starts automatically when CentraSite starts.

- **The Intermediate Queue**

  The queue from the SNMP Listener to the Event Processor. This queue decouples the SNMP Listener threads from the Event Processor to improve throughput. The following modes are supported.

  - FileSystem: Incoming Traps will be stored temporarily in the file system
  - InMemory: Incoming Traps will be stored temporarily in memory
  - NoQueue: Incoming Traps will not be stored in any intermediate queue; the SNMP Listener threads will be processed.

  To select the mode, set the `eventsQueueImpl` property as described in ***Setting the Events Queue Implementation Property***.

- **The Event Processor**

  The Event Processor (SOALinkSNMPEventsListener) transforms incoming SNMPv3 Traps into an XML file (Events.xml) that complies with the schema in the RuntimeEvents Collection component. The Event Processor transforms an SNMPv3 Trap to the Events.xml file as follows:

  1. Determines the Event Type (and Target Type) to which the Trap belongs, and gets the corresponding UUIDs. This involves searching all Event Type-to-Trap mappings in all the defined target types, using the Trap's OID. Since this is an expensive search, the Event Type-to-Trap mapping is cached to improve performance.
  2. Parses the Trap attributes and obtains: the Service (UUID); the Target (Name); the TimeStamp and the SessionId. The Processor then searches the registry/repository and obtains the corresponding UUID for the Target Name. This mapping is also cached to improve performance.
  3. Collects the remaining attributes from the Trap.
  4. Constructs the Events.xml file using the Event Type UUID, Target Type UUID, Service UUID, Target UUID, TimeStamp, SessionId and other collected attributes.

- **The Batch Condition**

  The Batch Condition is a set of OR conditions used by the Event Processor. The Event Processor supports two modes of event storage into CentraSite: BatchMode and NoBatchMode. BatchMode is available only for FileSystem and InMemory queues. When BatchMode is enabled, the Event

Processor continues to accumulate Events.xml documents until one of the conditions is evaluated as true. Then it inserts all the documents as a single batch into CentraSite.

To specify BatchMode or NoBatchMode, set the batch-related properties as described in ***Setting the Properties for FileSystem or InMemory***.

- **The RuntimeEvents Collection**

  The run-time events are stored in the RuntimeEvents Collection as non-registry objects. For information about how events are stored, see ***Event Type Modeling***.

Configuring the Event Receiver

The Event Receiver is bundled in the installation as a Web-Application named SOALinkSNMPEventsListener supporting the JavaEE standard. The configuration file for the Event Receiver is located here:

<CentraSite_directory>/cast/cswebapps/SOALinkSNMPEventsListener/WEB-INF/web.xml

The web.xml configuration file contains all the Event Receiver configuration properties. You must set these properties as described below, and then restart CentraSite.

- Setting the Database Configuration Properties
- Setting the SNMPv3 Transport Configuration Properties
- Setting the SNMPv3 USM Configuration Properties
- Setting the Events Queue Implementation Property
- Setting the Properties for FileSystem or InMemory

Setting the Database Configuration Properties

In the Event Receiver's configuration file, set the following properties related to the RuntimeEvents Collection database .

Database Property	Description
com.softwareag.centrasite.soalink.events.dbUrl	The URL of the RuntimeEvents Collection database. All run-time events will be persisted to this database.
com.softwareag.centrasite.soalink.events.dbUserId	The user name that the Events Listener will use for authentication before persisting event data to the RuntimeEvents Collection database. The default value of this property is the predefined user EventsUser. Optionally, you can change the value EventsUser to any login user who has the following privileges:

Database Property	Description
	■ Write access on the Tamino collection "RuntimeEvents". ■ Read access on "TargetTypes", "Targets", "RuntimeEventTypes" and "LogUnit", which are under the Tamino collection "CentraSite". If you want to change the value to a login user, enter that login user's name in the form *<hostName>\<userName>*. **Important:** The predefined password of EventsUser is EventsManager4CS (there is no need to specify the password in this file). If you want to change this password, or if you have changed the value EventsUser to a login user, *you must change the password*. For details, see the section *Users, Groups, Roles, and Permissions*. Whenever you change the password, you must restart CentraSite.
com.softwareag.centrasite.soalink.events.dbNonActivityTimeOut	The non-activity timeout in seconds for the RuntimeEvents Collection database (default 2592000 seconds (i.e., 30 days)).

Setting the SNMPv3 Transport Configuration Properties

In the Event Receiver's configuration file, set the following properties related to a SNMPv3 Transport.

SNMPv3 Transport Property	Description
com.softwareag.centrasite.soalink.events.snmp.transport	Wire transport protocol that will be used by the SNMP Listener. Supported values are: TCP and UDP.
com.softwareag.centrasite.soalink.events.snmp.host	The CentraSite host name or IP address to which the SNMP listener will bind.
com.softwareag.centrasite.soalink.events.snmp.port	The port to which the SNMP listener will bind. The default is 8181. If Microsoft Internet Information Services (IIS) is installed (or will be installed) on the same machine hosting

SNMPv3 Transport Property	Description
	IS/Mediator, then you may want to change the default SNMP port of 8181 to something else, to avoid any potential runtime conflicts when sending SNMP packets.
com.softwareag.centrasite.soalink.events.snmp.maxInboundMessageSizeInBytes	Maximum inbound message size in bytes (an integer). Traps that exceed this size limit will be rejected. Default value is 256Kb.
com.softwareag.centrasite.soalink.events.snmp.dispatcherPoolSize	The SNMP Listener's Worker-Thread pool size (default is 10). This determines the throughput of the Listener.

Setting the SNMPv3 USM Configuration Properties

In the Event Receiver's configuration file, set the following properties related to SNMPv3 USM.

SNMPv3 USM Property	Description
com.softwareag.centrasite.soalink.events.snmp.engineId	EngineId to be used by the SNMP Listener. If the parameter is left blank, the SNMP Listener will auto-generate the engineId.
com.softwareag.centrasite.soalink.events.snmp.securityName	The SecurityName to be used by the SNMP Listener.
com.softwareag.centrasite.soalink.events.snmp.securityLevel	The Maximum SecurityLevel to be supported by SNMP Listener. Supported values in order are: NOAUTH_NOPRIV, AUTH_NOPRIV and AUTH_PRIV. For example, AUTH_PRIV provides the highest level of security but also supports the other two levels. Similarly AUTH_NOPRIV supports NOAUTH_NOPRIV.
com.softwareag.centrasite.soalink.events.snmp.authProtocol	AuthorizationProtocol to be used by the SNMP Listener for decoding the incoming trap. Supported values are: MD5 and SHA.
com.softwareag.centrasite.soalink.events.snmp.authPassPhraseKey	The PassPhrase key to be used by the AuthorizationProtocol. The passphrase key length should be >= 8. The key is stored in this file; the passphrase value is stored securely in passman.
com.softwareag.centrasite.soalink.events.snmp.privProtocol	The PrivacyProtocol to be used by the SNMP Listener for decoding the incoming trap.

SNMPv3 USM Property	Description
	Supported values are: DES, AES128, AES, AES192, AES256, 3DES and DESEDE.
com.softwareag.centrasite.soalink.events.snmp.privPassPhraseKey	The PassPhrase key to be used by the PrivacyProtocol. The passphrase length should be >= 8. The key is stored in this file; the passphrase value is stored securely in passman.

Setting the Events Queue Implementation Property

In the Event Receiver's configuration file, set the following property related to the implementation of the events queue.

Events Queue Property	Description
com.softwareag.centrasite.soalink.events.eventsQueueImpl	Supported values are: ■ FileSystem: Incoming Traps will be stored temporarily in the file system ■ InMemory: Incoming Traps will be stored temporarily in memory ■ NoQueue: Incoming Traps will not be stored in any intermediate queue; the SNMP Listener threads will be processed one by one Additional, related properties are described in *Setting the Properties for FileSystem or InMemory*.

Setting the Properties for FileSystem or InMemory

When the *eventsQueueImpl* property is set to either FileSystem or InMemory, you should also set the following properties.

Property for FileSystem or InMemory	Description
com.softwareag.centrasite.soalink.events.enableBatchInsertion	Enable or disable batch insertion of events into the database. Supported values are true and false. If true, events will be batched as per the "batching rules" properties below, and the batch will be stored to the database. If false, events will be stored to the database one by one.
com.softwareag.centrasite.soalink.events.maxNumOfEventsPerBatch	Maximum number of events in a batch. Should be an integer value. A value <= 0 disables this rule. This rule is evaluated only on arrival of a new Trap.

Property for FileSystem or InMemory	Description
com.softwareag.centrasite.soalink.events.maxSizeOfBatch	Maximum size (in bytes) of a batch. Default value is 512KB. Should be an integer value. A value <= 0 disables this rule. This rule is evaluated only on arrival of a new Trap.
com.softwareag.centrasite.soalink.events.maxTimeIntervalBetweenBatches	Maximum time interval (in milliseconds) between two subsequent batch storages. Should be an integer value. A value <= 0 disables this rule. Unlike the other two rules, this rule is evaluated periodically. Hence this rule prevents any trap stuck in the batch for ever if inflow of traps stops; in short this acts as a batch-timeout. A very low value for this rule reduces batch efficiency and introduces unnecessary looping.
com.softwareag.centrasite.soalink.events.fileSystemQueueDir	(Only applies when the *eventsQueueImpl* property is set to FileSystem.) The directory that should be used as FileSystem Queue. Incoming traps will be stored in this directory temporarily and hence should have write permission. The path can be absolute or relative. It is advisable to provide the absolute path. Relative paths will be considered relative to one of the following, based on availability in the same order: 1. SOALinkSNMPEventsListener/WEB-INF directory for exploded deployments. 2. javax.servlet.context.tempdir for zipped deployments. 3. java.io.tmpdir if none of the above are available.

Event Type Modeling

Event types are modeled as registry objects. The String, Date, Integer and Boolean event attributes are stored in the registry/repository as slots. The File-Type attributes (representing payloads/binary-data) are stored as HasExternalLink associations.

For example, consider the predefined event type Transaction. If you go to the **Target Type** details page, you will see the Transaction event type attributes (which are obtained from the webMethodsESB.mib file) as follows:

Attribute Name	Object ID	Type
Service	1.3.6.1.4.1.1783.201.1.1.1	String
Target	1.3.6.1.4.1.1783.201.1.1.2	String
Timestamp	1.3.6.1.4.1.1783.201.1.1.3	Date
Consumer	1.3.6.1.4.1.1783.201.1.1.4	String
RequestStatus	1.3.6.1.4.1.1783.201.1.1.5	String
ResponsePayload	1.3.6.1.4.1.1783.201.1.1.6	File
RequestPayload	1.3.6.1.4.1.1783.201.1.1.7	File
ProviderRoundTripTime	1.3.6.1.4.1.1783.201.1.1.8	Integer
TotalRoundTripTime	1.3.6.1.4.1.1783.201.1.1.9	Integer
SessionID	1.3.6.1.4.1.1783.201.1.1.16	String
ConsumerIP	1.3.6.1.4.1.1783.201.1.1.17	String
OperationName	1.3.6.1.4.1.1783.201.1.1.21	String
NativeEndpoint	1.3.6.1.4.1.1783.201.1.1.22	String

All of these attributes (except the File-Type attributes RequestPayload and ResponsePayload) are stored as registry object slots, as follows:

Slot Key	Slot Type	Slot Value (Attribute)
uddi_16d34470-9a92-11dd-9b43-e319c2a6593c	xs:string	Service
uddi_f18b5a40-9a91-11dd-b95e-b4758b17b88b	xs:string	Target
uddi_c798d3c0-9a91-11dd-889e-b999c87ba6b7	xs:datetime	TimeStamp
uddi_a7476ff0-a108-11dd-9c38-d8fd010529cc	xs:string	Consumer
uddi_a7476ff0-a108-11dd-9c38-eac6d60fc855	xs:string	RequestStatus
uddi_a7476ff0-a108-11dd-9c38-f3f84c6111f0	xs:integer	ProviderRoundTripTime
uddi_a7476ff0-a108-11dd-9c38-d02170b3aae3	xs:integer	TotalRoundTripTime
uddi_21b67010-9a92-11dd-926a-991c4c180c79	xs:string	SessionID
uddi_a7476ff0-a108-11dd-9c38-d34f346cb3d5	xs:string	ConsumerIP
uddi_f1c8a185-4b18-4974-a360-6c70756a174a	xs:string	OperationName
uddi_524d05f5-d526-4605-b594-ace1cb750d33	xs:string	NativeEndpoint

The File-Type attributes ResponsePayload and RequestPayload are stored as HasExternalLink associations, as follows:

Association Key	Association Name (Attribute)
uddi:a747704b-a108-11dd-9c38-fde9d932116a	ResponsePayload
uddi:a745265b-a108-11dd-9c38-bf43eee17363	RequestPayload

The "Target Type to Event Type Association" Object

A target type (represented as a concept) is associated with an event type (represented as a registry object) by a "Target Type to Event Type Association" object, which defines the "UUID to MIB OID" mapping.

The following table shows the contents of a sample object that associates the target type webMethods Mediator with the event type Transaction. The table's columns are described below.

- Attribute: The Attribute column is not part of the object; it is included here simply for your reference.
- Slot Key: Contains the UUID, which is obtained from the event type registry object.
- Slot Type: Contains the slot type, which is obtained from the event type registry object.
- Slot Value: Contains the event type attribute's Object Identifier (OID), which is obtained from the MIB file.

Attribute	Slot Key (Event Type UUID)	Slot Type	Slot Value (Event Attribute OID)
Service	uddi_16d34470-9a92-11dd-9b43-e319c2a6593c	xs:string	1.3.6.1.4.1.1783.201.1.1.1
Target	uddi_f18b5a40-9a91-11dd-b95e-b4758b17b88b	xs:string	1.3.6.1.4.1.1783.201.1.1.2
TimeStamp	uddi_c798d3c0-9a91-11dd-889e-b999c87ba6b7	xs:datetime	1.3.6.1.4.1.1783.201.1.1.3
Consumer	uddi_a7476ff0-a108-11dd-9c38-d8fd010529cc	xs:string	1.3.6.1.4.1.1783.201.1.1.4
RequestStatus	uddi_a7476ff0-a108-11dd-9c38-eac6d60fc855	xs:string	1.3.6.1.4.1.1783.201.1.1.5
ResponsePayload	uddi_a747704b-a108-11dd-9c38-fde9d932116a	xs:anyURI	1.3.6.1.4.1.1783.201.1.1.6
RequestPayload	uddi_a745265b-a108-11dd-9c38-bf43eee17363	xs:anyURI	1.3.6.1.4.1.1783.201.1.1.7
ProviderRoundTripTime	uddi_a7476ff0-a108-11dd-9c38-f3f84c6111f0	xs:integer	1.3.6.1.4.1.1783.201.1.1.8
TotalRoundTripTime	uddi_a7476ff0-a108-11dd-9c38-d02170b3aae3	xs:integer	1.3.6.1.4.1.1783.201.1.1.9
SessionID	uddi_21b67010-9a92-11dd-926a-991c4c180c79	xs:string	1.3.6.1.4.1.1783.201.1.1.16
ConsumerIP	uddi_a7476ff0-a108-11dd-9c38-d34f346cb3d5	xs:string	1.3.6.1.4.1.1783.201.1.1.17
OperationName	uddi_f1c8a185-4b18-4974-a360-6c70756a174a	xs:string	1.3.6.1.4.1.1783.201.1.1.21
NativeEndpoint	uddi_524d05f5-d526-4605-b594-ace1cb750d33	xs:string	1.3.6.1.4.1.1783.201.1.1.22

Event Modeling

An event is an instance of an event type. Events are modeled in a separate schema from the event type schema. CentraSite models events as non-registry objects (to avoid storing large amounts of unwanted event data in the registry/repository), and instead stores event data in a database collection within the Event Receiver. CentraSite maps events to their corresponding event types, using the event types' UUIDs. Similarly, events are mapped to target types, targets and services using UUIDs and the event type attributes.

The stored event data will contain:

- The event Trap ID (MIB OID).
- The event Trap value, which consists of:
  - The attribute key (MIB OID).
  - The attribute value.

The event data is stored in the Event Receiver as an "events" doctype.

If an event contains payloads (e.g., File-Type attributes such as ResponsePayload and RequestPayload), the payloads are stored in the Event Receiver as a "payloads" doctype, and will be referenced by the event stored under the "event" doctype, using ino:id. This is used to reduce de-serialization of the usually large payloads, and to improve performance of queries on the stored events.

Viewing Run-Time Events and Metrics

You can view the run-time events and metrics that occurred for:

- A particular target or all targets (see ***Viewing Run-Time Events and Metrics for Targets***).
- Each virtual service (see ***Viewing Run-Time Events and Metrics for Virtual Services***).
- Each API (see ***Viewing Run-Time Events and Metrics for APIs***).

Viewing Run-Time Events and Metrics for Targets

Use the following procedure to view lists of run-time events for a particular target or for all targets.

If you are using the Mediator target, ensure that Mediator is configured to send event notifications to the destination(s) that are applicable for each event type. For details, see *SNMP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.

Note: You must have the permissions to manage targets, as described in the section *Run-Time Targets*.

To view a list of run-time events for targets

1 In CentraSite Control, go to **Operations > Events > Event List**.

2 Use the following fields to filter the event list you want to view:

In this field...	Specify...
Target Type	The type of the target whose events you want to view.
Target	The target whose events you want to view (or select **All** to view events of all targets).
Event Type	A particular event type, or select **All** to view all event types. For descriptions of the predefined event types, see the *The Run-Time Event Types*.
Service Type	Select **All** or **Virtual Service**. **Note:** CentraSite does not provide out-of-the-box policy-enforcement for web services.
Date Range	A range of dates from which to view the events.
Start Date	Alternatively, select the check box next to this field and click the calendar and select a starting date and time.
End Date	Click the calendar and select an ending date and time.

3 Click the **Search** button.

4 The generated event list displays the following information:

Field	Description
Date/Time	The date/time that the event occurred. Click this hyperlinked value to view the **Event Detail** page, which will contain the event's SOAP request or response name in the Attribute column. Click the hyperlinked request or response name to display the full SOAP request or response.
Session ID	(Read-only.) The session ID that generated the event.
Event Type	(Read-only.) The type of event (e.g., Monitoring, Policy Violation, Error, etc.).
Service Name	(Read-only.) The name of the service that caused the event.
Service Type	(Read-only.) The service's type.
Target	(Read only.) The target on which the event occurred.
Target Type	(Read only.) The type of the target on which the event occurred.

Note: To view the list of attributes that are mapped for each event type, go to the target type's detail page (see the section *Run-Time Targets*).

Viewing Run-Time Events and Metrics for Virtual Services

You can view the events and metrics for a virtual service in its Events profile and its Performance profile. For details, see the section *Virtual Services in CentraSite Control*.

Viewing Run-Time Events and Metrics for APIs

You can view the events and metrics for an API in its Runtime Events profile and its Runtime Metrics profile. For details, see the section *Virtual Services in CentraSite Control*.

Creating Custom Run-Time Events

CentraSite provides the predefined event types described in ***The Run-Time Event Types***. In addition, you can create custom run-time events that CentraSite will monitor.

Note: Prerequisite: You must have the Manage Runtime Event Types permission. By default, the predefined roles CentraSite Administrator and Operations Administrator include this permission. For more information about roles and permissions, see the section *Users, Groups, Roles, and Permissions*.

Important: To enable CentraSite to recognize custom event types, ensure that your MIB file (which is contained in your target type definition file) contains the SNMP Traps metadata and Object Identifiers for the custom events. For more information, see the section *Run-Time Targets*.

To create custom event types

1 In CentraSite Control, go to **Operations > Events > Event Types** to display the **Event Types** page.

The page displays all the predefined event types (Monitoring, Policy Violation, Transaction, Error and Lifecycle) and any custom event types that have been defined.

2 To view the details of any event type, click its hyperlinked name.

The list of attributes for the event type is displayed. You can edit the attributes of custom event types, but not the predefined event types (see ***Modifying Custom Run-Time Events***).

3 To create a custom event type, click the **Add Event Type** button. In the **Add/Edit Event Type** page specify a name and description for the event type. Event type names can contain any character (including spaces), and are not case-sensitive.

4 In the Event Type Attribute panel, the following default attributes are displayed. These attributes are required and cannot be deleted.

Attribute	Data Type
TimeStamp	Date
Target	String
Service	String
SessionID	String

To create additional attributes, perform the following steps:

1. Click the plus button at the bottom of the attribute list.
2. Specify a name in the **Name** column and a value in the **Data Type** column (Boolean, File, Date, Integer or String). Attribute names can contain any character (including spaces).
3. To add another attribute, click the plus button at the bottom of the list.
4. To delete an attribute, click the minus button for the attribute you want to delete.
5. Click **Save**.

Modifying Run-Time Events

To edit and delete custom event types, perform the following steps.

To modify a custom run-time event

1 In CentraSite Control, go to **Operations > Events > Event Types** to display the **Event Types** page.

The page displays all event types that have been defined.

2 To delete a custom event type, select the check box next to the event type and click the **Delete** button.

3 To edit the attributes of a custom event type, perform the following steps:

1. Click its hyperlinked name to display the **Add/Edit Event Type** page.
2. You can change the value of an attribute's data type, but not its name. Data types can be Boolean, File, Date, Integer or String.
3. To add another attribute, use the plus button at the bottom of the list.
4. To delete an attribute, click the minus button next to the attribute.
5. Click **Save**.

2 Built-In Run-Time Actions Reference for Virtual Services

This section describes the built-in run-time actions that you can include in run-time policies for virtual services. You use these actions only when you are using CentraSite Control to create run-time policies for virtual services. The content is organized under the following sections:

Summary of the Run-Time Actions for Virtual Services

You can include the following kinds of built-in run-time actions in the run-time policies for virtual services:

- WS-SecurityPolicy 1.2 Actions
- Monitoring Actions
- Additional Actions

WS-SecurityPolicy 1.2 Actions

Mediator provides two kinds of actions that support WS-SecurityPolicy 1.2: authentication actions and XML security actions.

Authentication Actions (WS-SecurityPolicy 1.2)

Mediator uses the following authentication actions to verify that the requests for virtual services contain a specified WS-Security element:

Require WSS Username Token	Uses WS-SecurityPolicy authentication to validate user names and passwords that are transmitted in the SOAP message header for the WSS Username token.
Require WSS X.509 Token	Identifies consumers based on a WSS X.509 token.
Require WSS SAML Token	Uses a WSS Security Assertion Markup Language (SAML) assertion token to validate service consumers.

XML Security Actions (WS-SecurityPolicy 1.2)

These actions provide confidentiality (through encryption) and integrity (through signatures) for request and response messages.

Require Signing	Requires that a request's XML element (which is represented by an XPath expression) be signed.
Require Encryption	Requires that a request's XML element (which is represented by an XPath expression) be encrypted.
Require SSL	Requires that requests be sent via SSL client certificates, and can be used by both SOAP and REST services.

Require Timestamps	Requires that timestamps be included in the request header. Mediator checks the timestamp value against the current time to ensure that the request is not an old message. This serves to protect your system against attempts at message tampering, such as replay attacks.

Monitoring Actions

Mediator provides the following run-time monitoring actions:

Monitor Service Performance	This action monitors a user-specified set of run-time performance conditions for a virtual service, and sends alerts to a specified destination when these performance conditions are violated.
Monitor Service Level Agreement	This action provides the same functionality as "Monitor Service Performance" but this action is different because it enables you to monitor a virtual service's run-time performance especially for particular consumer(s). You can configure this action to define a *Service Level Agreement* (SLA), which is set of conditions that defines the level of performance that a specified consumer should expect from a service.
Throttling Traffic Optimization	(Not available in Mediator versions below 9.0.) This action limits the number of service invocations during a specified time interval, and sends alerts to a specified destination when the performance conditions are violated. You can use this action to avoid overloading the back-end services and their infrastructure, to limit specific consumers in terms of resource usage, etc.

Additional Actions

Mediator provides the following actions, which you can use in conjunction with the actions above.

Identify Consumer	You use this action in conjunction with an authentication action ("Require WSS Username Token", "Require WSS X.509 Token" or "Require HTTP Basic Authentication"). Alternatively, you can use this action alone to identify consumers only by host name or IP address.
Require HTTP Basic Authentication	This action uses HTTP basic authentication to verify the consumer's authentication credentials contained in the request's Authorization header against the Integration Server's user account.
Authorize User	This action authorizes consumers against a list of users and/or a list of groups registered in the Integration Server on which Mediator is running. You use this action in conjunction with an authentication action "Require WSS Username Token", "Require WSS SAML Token" or "Require HTTP Basic Authentication".
Authorize Against Registered Consumers	This action authorizes consumer applications against all consumer applications who are registered in CentraSite as consumers for the service.
Log Invocations	Logs request/response payloads to a destination you specify.
Validate Schema	Validates all XML request and/or response messages against an XML schema referenced in the WSDL.

The watt.server.auth.skipForMediator Property

This property specifies whether Integration Server authenticates requests for Mediator. You must set this property to true.

No request to Mediator should be authenticated by Integration Server. Instead, authentication should be handled by Mediator. Thus, to enable Mediator to authenticate requests, you must set skipForMediator to true (by default it is false).

When this parameter is set to true, Integration Server skips authentication for Mediator requests and allows the user credentials (of any type) to pass through so that Mediator can authenticate them. If you change the setting of this parameter, you must restart Integration Server for the changes to take effect.

To set skipForMediator to true

1 In the Integration Server Administrator, click **Settings > Extended**.

2 Click **Show and Hide Keys**.

Look for the `watt.server.auth.skipForMediator` property and ensure it is set to true.

3 If the `watt.server.auth.skipForMediator` property is not present, add it as follows:

1. Click **Edit Extended Settings**.
2. Type `watt.server.auth.skipForMediator=true` on a separate line.
3. Click **Save**.
4. Restart Integration Server.

Action Evaluation Order and Dependencies

When you deploy a virtual service, CentraSite automatically validates the service's run-time policy (or policies) to ensure that:

- Any action that appears in a single policy multiple times is allowed to appear multiple times.

  For those actions that can appear in a policy only once (for example, Identify Consumer), Mediator will choose only one, which might cause problems or unintended results.
- All action dependencies are properly met. That is, some actions must be used in conjunction with another particular action.

CentraSite will inform you of any violation, and you will need to correct the violations before deploying the service.

Effective Policies

When you deploy a virtual service to Mediator, CentraSite combines the actions specified within the service's run-time policy (or policies) that apply to the virtual service, and generates what is called the *effective policy* for the virtual service. For example, suppose your virtual service is within the scope of two run-time policies: one policy that performs a logging action and another policy that performs a security action. When you deploy the virtual service, CentraSite automatically combines the two policies into one effective policy. The effective policy, which contains both the logging action and the security action, is the policy that CentraSite actually deploys to Mediator with the virtual service.

When CentraSite generates the effective policy, it validates the resulting action list to ensure that it contains no conflicting or incompatible actions. If the list contains conflicts or inconsistencies, CentraSite resolves them according to Policy Resolution Rules. For example, an action list can include only one Identify Consumer action. If the resulting action list contains multiple Identify Consumer actions, CentraSite resolves the conflict by including only one of the actions (selected according to a set of internal rules) in the effective policy and omitting the others.

The effective policy that CentraSite produces for a virtual service is contained in an object called a *virtual service definition* (VSD). The VSD is given to Mediator when you deploy the virtual service. After you deploy a virtual service, you can view its VSD (and thus examine the effective policy that CentraSite generated for it) from the CentraSite user interface or from the Mediator user interface.

The following table shows:

- The order in which Mediator evaluates the actions.
- Action dependencies (that is, whether an action must be used in conjunction with another particular action).
- Whether an action can be included multiple times in a single policy. If an action cannot be included multiple times in a single policy, Mediator selects just one for the effective policy, which may cause problems or unintended results.

Evaluation Order	Action	Dependency	Can include multiple times in a policy?
1	**Require SSL**	None.	If multiple actions appear, and one of them has its Client Certificate Required parameter set to Yes, only one occurrence of the action appears in the effective policy.
2	**Require HTTP Basic Authentication**	In Mediator versions below 9.0: None. In Mediator version 9.0 and above: Identify Consumer.	No. Mediator includes only one action in the effective policy.
3	**Require WSS Username Token**	Identify Consumer action.	No. Mediator includes only one action in the effective policy.
4	**Require WSS X.509 Token**	Identify Consumer action.	No. Mediator includes only one action in the effective policy.
5	**Require WSS SAML Token**	None.	No. Mediator includes only one action in the effective policy.
6	**Require Signing**	Identify Consumer action.	Yes. Mediator generates a UNION of all Require Signing actions for the effective policy.
7	**Require Encryption**	Identify Consumer action.	Yes. Mediator generates a UNION of all Require Encryption actions for the effective policy.
8	**Require Timestamps**	Require SSL, Require Signing *and* Require Encryption.	No. Mediator includes only one action in the effective policy.
9	**Identify Consumer**	If Identify Consumer's identifier field is set to: ■ HTTP Authentication Token, the action Require HTTP Basic Authentication is also required. ■ WS-Security Authentication Token, the action Require WSS Username Token is also required. ■ Consumer Certificate, the actions Require WSS X.509 Token or Require Signing are also required.	No. Mediator includes only one action in the effective policy.
10	**Authorize User**	Require HTTP Basic Authentication, Require WSS Username Token *or* Require WSS SAML Token.	No. Mediator includes only one action in the effective policy.

Evaluation Order	Action	Dependency	Can include multiple times in a policy?
11	**Authorize Against Registered Consumers**	Identify Consumer action.	No. Mediator includes only one action in the effective policy.
12	**Validate Schema**	None.	If at least one occurrence of the action is configured to validate requests, and at least one occurrence of the action is configured to validate responses, then Mediator includes in the effective policy an action to validate both requests and responses. Otherwise, an action is chosen which validates only requests or only responses (depending on the value of the Validate SOAP Messages parameter of the action).
13	**Log Invocation**	None.	No. Mediator includes only one action in the effective policy.
14	**Monitor Service Performance**	None.	Yes. Mediator includes all Monitor Service Performance actions in the effective policy.
15	**Monitor Service Level Agreement**	Identify Consumer action.	Yes. Mediator includes all Monitor Service Level Agreement actions in the effective policy.
16	**Throttling Traffic Optimization**	Identify Consumer (if the Limit Traffic for Applications option is selected).	Yes. Mediator includes all Throttling Traffic Optimization actions in the effective policy.

Usage Cases for Identifying/Authenticating Consumers

When deciding which type of identifier to use to identify a consumer application, consider the following points:

- Whatever identifier you choose to identify a consumer application, it must be unique to the application. Identifiers that represent user names are often not suitable because the identified users might submit requests for multiple applications.
- Identifying applications by IP address or host name is often a suitable choice, however, it does create a dependency on the network infrastructure. If a consumer application moves to a new machine, or its IP address changes, you must update the identifiers in the application asset.
- Using X.509 certificates or a custom token that is extracted from the SOAP message itself (using an XPATH expression), is often the most trouble-free way to identify a consumer application.

Following are some common combinations of actions used to authenticate/identify consumers.

- **Scenario 1: Identify consumers by IP address or host name**
  - The simplest way to identify consumers is to use the Identify Consumer action and set its `Identify User Using` parameter to specify either a host name or an IP address (or a range of IP addresses).
- **Scenario 2: Authenticate consumers by HTTP authentication token**
  Use the following actions:
  - Identify Consumer action, and set its `Identify User Using` parameter to HTTP Authentication Token (to identify consumers using the token derived from the HTTP header).
  - Require HTTP Basic Authentication.
  - Additionally, you can use one or both of the following:
    - Authorize User action (to authorize a list of users and/or groups registered in the Integration Server on which Mediator is running).
    - Authorize Against Registered Consumers action (to authorize consumer applications against all Application assets registered as consumers for a service in CentraSite).
- **Scenario 3: Authenticate consumers by WS-Security authentication token**
  Use the following actions:
  - Identify Consumer action, and set its `Identify User Using` parameter to WS-Security Authentication Token (to identify consumers using the token derived from the WSS Header).
  - Require WSS Username Token action.
  - Additionally, you can use one or both of the following:
    - Authorize User action (to authorize a list of users and/or groups registered in the Integration Server on which Mediator is running).
    - Authorize Against Registered Consumers action (to authorize consumer applications against all Application assets registered as consumers for a service in CentraSite).
- **Scenario 4: Authenticate consumers by WSS X.509 token**
  - Identify Consumer action, and set its `Identify User Using` parameter to Consumer Certificate (to identify consumers using the WSS X.509 token).
  - Require WSS X.509 Token action
  - Require SSL action.

Run-Time Actions Reference for Virtual Services

This section describes the following built-in run-time actions that you can include in run-time policies for virtual services:

Authorize Against Registered Consumers

Note: Dependency requirement: A policy that includes this action must also include the **Identify Consumer** action. However, if the **Identify Consumer** action is set to identify users via the **HTTP Authentication Token** option, then "Authorize Against Registered Consumers" should not be included in the policy.

Authorizes consumer applications against all consumer applications who are registered in CentraSite as consumers for the service.

Input Parameters

None.

Authorize User

Note: Dependency requirement: A policy that includes this action must also include *one* of the following: the **Require WSS SAML Token** action or the **Identify Consumer** action with one of the following options selected: "HTTP Authentication Token" or "WS-Security Authentication Token".

Authorizes consumers against a list of users and/or a list of groups registered in the Integration Server on which Mediator is running.

Input Parameters

`Perform authorization against list of users`	*Boolean* Authorizes consumers against a list of users who are registered in the Integration Server on which Mediator is running. Specify one or more users in the fields below this option.
`Perform authorization against list of groups`	*Boolean* Authorizes consumers against a list of groups who are registered in the Integration Server on which Mediator is running. Specify one or more groups in the fields below this option.

Note: By default, both of the input parameters are selected. If you de-select one of these parameters, the fields showing the list of users (or groups) is not displayed.

Identify Consumer

Mediator uses this action to identify consumer applications based on the kind of consumer identifier (IP address, HTTP authorization token, etc.) you specify. Alternatively, this action provides an option to allow anonymous users to access the assets.

Input Parameters

`Anonymous Usage Allowed`	*Boolean* Specifies whether to allow all users to access the asset, without restriction.	
	Value	**Description**
	`False`	*Default.* Allows only the users specified in the `Identify User Using` parameter to access the assets.
	`True`	Allow all users to access the asset. In this case, do not configure the `Identify User Using` parameter.
`Identify User Using`	*String* Specifies the kind of consumer identifier that the action will use to identify consumer applications.	
	Value	**Description**

	`IP Address`	Identifies one or more consumer applications based on their originating IP addresses.
	`Host Name`	Identifies consumer applications based on a host name.
	`HTTP ↩ Authentication ↩ Token`	Uses HTTP Basic authentication to verify the consumer's authentication credentials contained in the request's Authorization header. Mediator authorizes the credentials against the list of consumers available in the Integration Server on which Mediator is running. This type of consumer authentication is referred to as "preemptive authentication". If you want to use "preemptive authentication", you should also include the action **Require HTTP Basic Authentication** in the policy. If you choose to omit "Require HTTP Basic Authentication", the client will be presented with a security challenge. If the client successfully responds to the challenge, the user is authenticated. This type of consumer authentication is referred to as "non-preemptive authentication". For more information, see **Require HTTP Basic Authentication**. **Note:** If you select the value `HTTP Authentication Token`, do not include the `Authorize Against Registered Consumers` action in the policy. This is an invalid combination.
	`WS-Security ↩ Authentication ↩ Token`	Validate user names and passwords that are transmitted in the SOAP message header in the WSS Username Token. If you select this value, you should also include the action **Require WSS Username Token** in the policy.
	`Custom ↩ Identification`	Validates consumer applications based on an XML element (represented by an XPath expression).
	`Consumer ↩ Certificate`	Identifies consumer applications based on information in a WSS X.509 certificate. If you select this value, you should also include the action **Require WSS X.509 Token** or the action **Require Signing** in the policy.
	`Client ↩ Certificate for ↩ SSL Connectivity`	Validates the client's certificate that the consumer application submits to the asset in CentraSite. The client certificate that is used to identify the consumer is supplied by the client to the Mediator during the SSL handshake over the transport layer. In order to identify consumers by transport-level certificates, the run-time communication between the client and the Mediator must be over HTTPS and the client must pass a valid certificate. To use this option, the following prerequisites must be met: ■ In Integration Server, create a keystore and truststore, as described in *Securing Communications with the Server* in the *webMethods Integration Server Administrator's Guide*. ■ In Integration Server, create an HTTPS port, as described in *Configuring Ports* in the *webMethods Integration Server Administrator's Guide*.

		■ Configure Mediator by setting the IS Keystore and IS Truststore parameters, as described in *Configuring Mediator > Keystore Configuration* in the document *Administering webMethods Mediator*. ■ Configure Mediator by setting the HTTPS Ports Configuration parameter, as described in *Configuring Mediator > Ports Configuration* in the document *Administering webMethods Mediator*.

When deciding which type of identifier to use to identify a consumer application, consider the following points:

- Whatever identifier you choose to identify a consumer application, it must be unique to the application. Identifiers that represent user names are often not suitable because the identified users might submit requests for multiple applications.
- Identifying applications by IP address or host name is often a suitable choice, however, it does create a dependency on the network infrastructure. If a consumer application moves to a new machine, or its IP address changes, you must update the identifiers in the application asset.
- Using X.509 certificates or a custom token that is extracted from the SOAP or XML message itself (using an XPATH expression), is often the most trouble-free way to identify a consumer application.

Log Invocation

Logs request/response payloads. You can specify the log destination and the logging frequency. This action also logs other information about the requests/responses, such as the service name, operation name, the Integration Server user, a timestamp, and the response time.

Note: You can include this action multiple times in a policy.

Input Parameters

`Log the Following Payloads`	*String* Optional. Specifies whether to log all request payloads, all response payloads, or both.	
	Value	**Description**
	`Request`	Log all request payloads.
	`Response`	Log all response payloads.
`Log Generation Frequency`	*String* Specifies how frequently to log the payload.	
	Value	**Description**

	`Always`	Log all requests and/or responses.
	`On Success`	Log only the successful responses and/or requests.
	`On Failure`	Log only the failed requests and/or responses.
`Send Data To`	*String* Specifies where to log the payload. **Important:** Ensure that Mediator is configured to log the payloads to the destination(s) you specify here. For details, see *Alerts and Transaction Logging* in the document *Administering webMethods Mediator*.	
	Value	**Description**
	`CentraSite`	Logs the payloads in the virtual service's Events profile in CentraSite. Prerequisite: You must configure Mediator to communicate with CentraSite (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > CentraSite Communication**). For the procedure, see the section *Configuring Communication with CentraSite* in the document *Administering webMethods Mediator*.
	`Local Log`	Logs the payloads in the server log of the Integration Server on which Mediator is running. Also choose a value in the `Log Level` field: ■ Info: Logs error-level, warning-level, and informational-level alerts. ■ Warn: Logs error-level and warning-level alerts. ■ Error: Logs only error-level alerts. **Important:** The Integration Server Administrator's logging level for Mediator should match the logging level specified for this action (go to **Settings > Logging > Server Logger**).
	`SNMP`	Logs the payloads in CentraSite's SNMP server or a third-party SNMP server. Prerequisite: You must configure the SNMP server destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > SNMP**). For the procedure, see the section *SNMP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.

	Email	Sends the payloads to an SMTP email server, which sends them to the email address(es) you specify here. Mediator sends the payloads as email attachments that are compressed using gzip data compression. To specify multiple addresses, use the  button to add rows. Prerequisite: You must configure the SMTP server destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > Email**). For the procedure, see the section *SMTP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.
	Audit Log	Logs the payload to the Integration Server audit logger. For information, see the *webMethods Audit Logging Guide*. **Note:** If you expect a high volume of events in your system, it is recommended that you select the Audit Log destination for this action.

Monitor Service Performance

This action monitors a user-specified set of run-time performance conditions for a virtual service, and sends alerts to a specified destination when the performance conditions are violated. You can include this action multiple times in a single policy.

For the counter-based metrics (Total Request Count, Success Count, Fault Count), Mediator sends an alert as soon as the performance condition is violated, without having to wait until the end of the metrics tracking interval. You can choose whether to send an alert only once during the interval, or every time the violation occurs during the interval. (Mediator will send another alert the next time a condition is violated during a subsequent interval.) For information about the metrics tracking interval, see ***The Metrics Tracking Interval***.

For the aggregated metrics (Average Response Time, Minimum Response Time, Maximum Response Time), Mediator aggregates the response times at the end of the interval, and then sends an alert if the performance condition is violated.

This action does not include metrics for failed invocations.

Note: To enable Mediator to publish performance metrics, you must configure Mediator to communicate with CentraSite (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > CentraSite Communication**). For the procedure, see the section *Configuring Communication with CentraSite* in the document *Administering webMethods Mediator*.

Input Parameters

Action Configuration parameters	Specify one or more conditions to monitor. To do this, specify a metric, operator, and a value for each metric. To specify multiple conditions, use the button to add multiple rows. If multiple parameters are used, they are connected by the AND operator.	
Name	*String Array* The metrics to monitor.	
	Value	**Description**
	Availability	Indicates whether the service was available to the specified consumers in the current interval.
	Average Response Time	The average amount of time it took the service to complete all invocations in the current interval. Response time is measured from the moment Mediator receives the request until the moment it returns the response to the caller.

Monitor Service Level Agreement

Note: Dependency requirement: A policy that includes this action must also include the **Identify Consumer** action.

This action is similar to the Monitor Service Performance action. Both actions can monitor the same set of run-time performance conditions for a virtual service, and then send alerts when the performance conditions are violated. This action is different because it enables you to monitor run-time performance for *one or more specified consumers*. You can include this action multiple times in a single policy.

You can configure this action to define a *Service Level Agreement (SLA)*, which is a set of conditions that defines the level of performance that a consumer should expect from a service. You can use this action to identify whether a service's threshold rules are met or exceeded. For example, you might define an agreement with a particular consumer that sends an alert to the consumer if responses are not sent within a certain maximum response time. You can configure SLAs for each virtual service/consumer application combination.

For the counter-based metrics (Total Request Count, Success Count, Fault Count), Mediator sends an alert as soon as the performance condition is violated, without having to wait until the end of the metrics tracking interval. You can choose whether to send an alert only once during the interval, or every time the violation occurs during the interval. (Mediator will send another alert the next time a condition is violated during a subsequent interval.) For information about the metrics tracking interval, see *The Metrics Tracking Interval*.

For the aggregated metrics (Average Response Time, Minimum Response Time, Maximum Response Time), Mediator aggregates the response times at the end of the interval, and then sends an alert if the performance condition is violated.

This action does not include metrics for failed invocations.

Note: To enable Mediator to publish performance metrics, you must configure Mediator to communicate with CentraSite (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > CentraSite Communication**). For the procedure, see the section *Configuring Communication with CentraSite* in the document *Administering webMethods Mediator*.

Input Parameters

`Action Configuration parameters`	Specify one or more conditions to monitor. To do this, specify a metric, operator, and value for each metric. To specify multiple conditions, use the button to add multiple rows. If multiple parameters are used, they are connected by the AND operator.	
`Name`	*String Array* The metrics to monitor.	
	Value	**Description**
	`Availability`	Indicates whether the service was available to the specified consumers in the current interval.
	`Average Response Time`	The average amount of time it took the service to complete all invocations in the current interval. Response time is measured from the moment Mediator receives the request until the moment it returns the response to the caller.
	`Fault Count`	Indicates the number of faults returned in the current interval.
	`Maximum Response Time`	The maximum amount of time to respond to a request in the current interval.
	`Minimum Response Time`	The minimum amount of time to respond to a request in the current interval.
	`Successful Request Count`	The number of successful requests in the current interval.
	`Total Request Count`	The total number of requests (successful and unsuccessful) in the current interval.
	`Operator`	*String Array* Choose an appropriate operator.
`Value`	*String Array* Specify an appropriate value.	
`Alert for Consumer Applications`	*Object Array* Specify the Application asset(s) to which this Service Level Agreement will apply. To specify multiple Application assets, use the button to add multiple rows.	
`Alert parameters`	*Object* Specify the following parameters for the alerts that will report on the Service Level Agreement conditions:	
`Alert Interval`	*Number* The time period (in minutes) in which to monitor performance before sending an alert if a condition is violated. For information about the metrics tracking interval, see ***The Metrics Tracking Interval***.	

`Alert Frequency`	*String* Specifies how frequently to issue alerts for the counter-based metrics (Total Request Count, Success Count, Fault Count).	
	Value	**Description**
	`Every Time`	Issue an alert every time one of the specified conditions is violated.
	`Only Once`	Issue an alert only the first time one of the specified conditions is violated.

Require Encryption

Requires that a request's XML element (which is represented by an XPath expression) be encrypted. This action supports WS-SecurityPolicy 1.2 and cannot be used with REST services.

Prerequisites

1. Configure Integration Server: Set up keystores and truststores in Integration Server, as described in *Securing Communications with the Server* in the document *Administering webMethods Integration Server*.
2. Configure Mediator: In the Integration Server Administrator, navigate to **Solutions > Mediator > Administration > General** and complete the IS Keystore Name, IS Truststore Name and Alias (signing) fields, as described in *Keystore Configuration* in the document *Administering WebMethods Mediator*.

When this policy action is set for the virtual service, Mediator provides decryption of incoming requests and encryption of outgoing responses. Mediator can encrypt and decrypt only individual elements in the SOAP message body that are defined by the XPath expressions configured for the policy action. Mediator requires that requests contain the encrypted elements that match those in the XPath expression. You must encrypt the entire element, not just the data between the element tags. Mediator rejects requests if the element name is not encrypted.

Important: Do not encrypt the entire SOAP body because a SOAP request without an element will appear to Mediator to be malformed.

Mediator attempts to encrypt the response elements that match the XPath expressions with those defined for the policy. If the response does not have any elements that match the XPath expression, Mediator will not encrypt the response before sending. If the XPath expression resolves a portion of the response message, but Mediator cannot locate a certificate to encrypt the response, then Mediator sends a SOAP fault exception to the consumer and a Policy Violation event notification to CentraSite.

How Mediator Encrypts Responses

The Require Encryption action encrypts the response back to the client by dynamically setting a public key alias at run time. Mediator determines the public key alias as follows:

1. If Mediator can access the X.509 certificate of the client (based on the incoming request signature), it will use "useReqSigCert" as the public key alias.

   OR

2. If the Identify Consumer action is present in the policy (and it successfully identifies a consumer application), then Mediator will look for a public key alias with that consumer name in the "IS Keystore Name" property. The "IS Keystore Name" property is specified in the Integration Server Administrator, under Solutions > Mediator > Administration > General. This property should be set to an Integration Server keystore that Mediator will use.

   For an Identify Consumer action that allows for anonymous usage, Mediator does *not* require a consumer name in order to send encrypted responses. In this case, Mediator can use one of the following to encrypt the response in the following order, depending on what is present in the security element:

   - A signing certificate.
   - Consumer name.
   - WSS username, SAML token or X.509 certificate.
   - HTTP authorized user.

   OR

3. If Mediator can determine the current IS user from the request (i.e., if an Integration Server WS-Stack determined that Subject is present), then the first principal in that subject is used.

   OR

4. If the above steps all fail, then Mediator will use either the WS-Security username token or the HTTP Basic-Auth user name value. There should be a public key entry with the same name as the identified username.

Note: You can include this action multiple times in a single policy.

Input Parameters

`Namespace`	*String* Optional. Namespace of the element required to be encrypted. **Note:** Enter the namespace prefix in the following format: `xmlns:<prefix-name>`. For example: `xmlns:soapenv`. For more information, see the XML Namespaces specifications at **http://www.w3.org/TR/REC-xml-names/#ns-decl**. The generated XPath element in the policy should look similar to this:

	<sp:SignedElements ↩ xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"> <sp:XPath ↩ xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">//soapenv:Body</sp:XPath </sp:SignedElements>
Element Required to be Encrypted	*String* An XPath expression that represents the XML element that is required to be encrypted.

Require HTTP Basic Authentication

This action uses HTTP Basic authentication to verify the consumer's authentication credentials contained in the request's Authorization header. Mediator authorizes the credentials against the list of consumers available in the Integration Server on which Mediator is running. This type of consumer authentication is referred to as “preemptive authentication”. If you want to perform “preemptive authentication”, a policy that includes this action must also include the Identify Consumer action.

If the user/password value in the Authorization header cannot be authenticated as a valid Integration Server user (or if the Authorization header is not present in the request), a 500 SOAP fault is returned, and the client is presented with a security challenge. If the client successfully responds to the challenge, the user is authenticated. This type of consumer authentication is referred to as “non-preemptive authentication”. If the client does not successfully respond to the challenge, a 401 “WWW-Authenticate: Basic” response is returned and the invocation is not routed to the policy engine. As a result, no events are recorded for that invocation, and its key performance indicator (KPI) data are not included in the performance metrics.

If you choose to omit the “Require HTTP Basic Authentication” action (and regardless of whether an Authorization header is present in the request or not), then:

- Mediator forwards the request to the native service, without attempting to authenticate the request.
- The native service returns a 401 “WWW-Authenticate: Basic” response, which Mediator will forward to the client; the client is presented with a security challenge. If the client successfully responds to the challenge, the user is authenticated.

In the case where a consumer sends a request with transport credentials (HTTP Basic authentication) and message credentials (WSS Username or WSS X.509 token), the message credentials take precedence over the transport credentials when Integration Server determines which credentials it should use for the session. For more information, see **Require WSS Username Token** and **Require WSS X.509 Token**. In addition, you must ensure that the service consumer that connects to the virtual service has an Integration Server user account.

Note: Do not include the "Require HTTP Basic Authentication" action in a virtual service's run-time policy if you selected the **OAuth2** option in the virtual service's Routing Protocol step.

Input Parameters

Note: This input parameter is not available in Mediator versions prior to 9.0.

`Authenticate Credentials`	Required. Authorizes consumers against the list of consumers available in the Integration Server on which Mediator is running.

Require Signing

This action requires that a request's XML element (which is represented by an XPath expression) be signed. This action supports WS-SecurityPolicy 1.2.

Prerequisites

1. Configure Integration Server: Set up keystores and truststores in Integration Server, as described in *Securing Communications with the Server* in the document *Administering webMethods Integration Server*.
2. Configure Mediator: In the Integration Server Administrator, navigate to **Solutions > Mediator > Administration > General** and complete the IS Keystore Name, IS Truststore Name and Alias (signing) fields, as described in *Keystore Configuration* in the document *Administering WebMethods Mediator*. Mediator uses the signing alias specified in the Alias (signing) field to sign the response.

When this action is set for the virtual service, Mediator validates that the requests are properly signed, and provides signing for responses. Mediator provides support both for signing an entire SOAP message body or individual elements of the SOAP message body.

Mediator uses a digital signature element in the security header to verify that all elements matching the XPath expression were signed. If the request contains elements that were not signed or no signature is present, then Mediator rejects the request.

Notes:

1. You must map the public certificate of the key used to sign the request to an Integration Server user. If the certificate is not mapped, Mediator returns a SOAP fault to the caller.
2. You can include this action multiple times in a policy.

Input Parameters

`Namespace`	*String* Optional. Namespace of the element required to be signed. **Note:** Enter the namespace prefix in the following format: `xmlns:<prefix-name>`. For example: `xmlns:soapenv`. For more information, see the XML Namespaces specifications at **http://www.w3.org/TR/REC-xml-names/#ns-decl**. The generated XPath element in the policy should look similar to this: `<sp:SignedElements ↩ xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"> <sp:XPath ↩ xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">//soapenv:Body</sp:XPath </sp:SignedElements>`
`Element Required to be Signed`	*String* An XPath expression that represents the XML element that is required to be signed.

Require SSL

Requires that requests be sent via SSL client certificates. This action supports WS-SecurityPolicy 1.2 and can be used for both SOAP and REST services.

When this action is set for the virtual service, Mediator ensures that requests are sent to the server using the HTTPS protocol (SSL). The action also specifies whether the client certificate is required. This allows Mediator to verify the client sending the request. If the policy requires the client certificate, but it is not presented, Mediator rejects the message.

When a client certificate is required, the Integration Server HTTPS port should be configured to request or require a client certificate.

Input Parameters

`Client Certificate Required`	*Boolean* Specifies whether client certificates are required for the purposes of: ■ Verifying the signature of signed SOAP requests or decrypting encrypted SOAP requests. ■ Signing SOAP responses or encrypting SOAP responses.	
	Value	**Description**

	`Yes`	Require client certificates.
	`No`	Default. Do not require client certificates.

Require Timestamps

Note: Dependency requirement: A policy that includes this action must also include *all* of the following actions: **Require SSL**, **Require Signing**, **Require Encryption**.

When this policy action is set for the virtual service, Mediator requires that timestamps be included in the request header. Mediator checks the timestamp value against the current time to ensure that the request is not an old message. This serves to protect your system against attempts at message tampering, such as replay attacks. This action supports WS-SecurityPolicy 1.2 and cannot be used with REST services.

Mediator rejects the request if either of the following happens:

- Mediator receives a timestamp that exceeds the time defined by the timestamp element.
- A timestamp element is not included in the request.

Input Parameters

None.

Require WSS SAML Token

When this action is set for a virtual service, Mediator uses a WSS Security Assertion Markup Language (SAML) assertion token to validate service consumers. This action supports WS-SecurityPolicy 1.2 and cannot be used with REST services.

For more information about configuring your system for SAML token processing, see *SAML Support in Mediator* in the document *Administering webMethods Mediator*.

Input Parameters

`SAML Subject Confirmation`	*String* Select one of the following SAML subject confirmation methods:	
	Value	**Description**
		Default. Select this option if consumers use the SAML V1.1 or V2.0 Holder-of-Key Web Browser SSO Profile, which allows for transport of holder-of-key assertions. In this scenario, the consumer presents a holder-of-key SAML assertion acquired from its preferred identity provider to access a web-based resource at a service provider. If you select `Holder of Key`, Mediator also implicitly selects the "timestamp" and "signing" assertions to the virtual service definition

	`Holder of Key`	(VSD). Thus, you should not add the ""Require Timestamps" and "Require Signing" policy actions to a virtual service if the "Require WSS SAML Token" action is already applied.
	`Bearer`	Select this option if consumers use SAML V1.1 Bearer token authentication, in which a Bearer token mechanism relies upon bearer semantics as a means by which the consumer conveys to Mediator the sender's identity. If you select `Bearer`, the "timestamp" and "signing" assertions will be added to the virtual service definition (VSD). **Note:** If consumers use SAML 2.0 Sender-Vouches tokens, configure your system as described in *SAML Support in Mediator* in the document *Administering WebMethods Mediator*.
`SAML Version`	*String* Specifies the WSS SAML Token version to use: 1.1 or 2.0.	

Require WSS Username Token

Note: Dependency requirement: A policy that includes this action must also include the Identify Consumer action.

When this policy action is set for the virtual service, Mediator uses WS-SecurityPolicy authentication to validate user names and passwords that are transmitted in the SOAP message header for the WSS Username token. This action supports WS-SecurityPolicy 1.2 and cannot be used with REST services.

In the case where a consumer is sending a request with both transport credentials (HTTP basic authentication) and message credentials (WSS Username or X.509 token), the message credentials take precedent over the transport credentials when Integration Server is determining which credentials it should use for the session. For more information, see **Require HTTP Basic Authentication**.

Mediator rejects requests that do not include the username token and password of an Integration Server user. Mediator only supports clear text passwords with this kind of authentication

Input Parameters

None.

Require WSS X.509 Token

Note: Dependency requirement: A policy that includes this action must also include the Identify Consumer action.

Identifies consumers based on a WSS X.509 token. This action supports WS-SecurityPolicy 1.2 and cannot be used with REST services.

In the case where a consumer is sending a request with both transport credentials (HTTP Basic authentication) and message credentials (WSS X.509 token or WSS Username), the message credentials take precedence over the transport credentials when Integration Server is determining which credentials it should use for the session. For more information, see **Require HTTP Basic Authentication**. In addition, you must ensure that the service consumer that connects to the virtual service has an Integration Server user account.

Input Parameters

None.

Throttling Traffic Optimization

Notes:

1. This action is not available in Mediator versions below 9.0.
2. Dependency requirement: A policy that includes this action must also include the **Identify Consumer** action if the `Limit Traffic for Applications` option is selected.

This action limits the number of service invocations during a specified time interval, and sends alerts to a specified destination when the performance conditions are violated.

Reasons for limiting the service invocation traffic include:

- To avoid overloading the back-end services and their infrastructure.
- To limit specific consumers in terms of resource usage (that is, you can use the "Monitor Service Level Agreement" action to monitor performance conditions for a particular consumer, together with "Throttling Traffic Optimization" to limit the resource usage).
- To shield vulnerable servers, services, and even specific operations.
- For service consumption metering (billable pay-per-use services).

Note: To enable Mediator to publish performance metrics, you must configure Mediator to communicate with CentraSite (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > CentraSite Communication**). For the procedure, see the section *Configuring Communication with CentraSite* in the document *Administering webMethods Mediator*.

Input Parameters

`Soft Limit`	*Number* Optional. Specifies the maximum number of invocations allowed per `Interval` before issuing an alert. Reaching the soft limit will not affect further processing of requests (until the `Hard Limit` is reached). **Note:** The limit is reached when the total number of invocations coming from all *all* the consumer applications (specified in the `Limit Traffic for Applications` field) reaches the limit. Soft Limit is computed in an asynchronous manner; thus when multiple requests are made at the same time, it may be possible that the Soft Limit alert will not be strictly accurate.	
`Hard Limit`	*Number* Required. Specifies the maximum number of invocations allowed per alert interval before stopping the processing of further requests and issuing an alert. Typically, this number should be higher than the soft limit. **Note:** The limit is reached when the total number of invocations coming from all *all* the consumer applications (specified in the `Limit Traffic for Applications` field) reaches the limit. Hard Limit is computed in an asynchronous manner; thus when multiple requests are made at the same time, it may be possible that the Hard Limit alert will not be strictly accurate.	
`Limit Traffic for Applications`	*String* Specifies the consumer application(s) that this action applies to. To specify multiple consumer applications, use the button to add rows, or select **Any Consumer** to apply this action to any consumer application.	
`Interval`	*Number* Specifies the amount of time for the soft limit and hard limit to be reached.	
`Frequency`	*String* Specifies how frequently to issue alerts.	
	Value	**Description**
	`Every Time`	Issue an alert every time the specified condition is violated.
	`Only Once`	Issue an alert only the first time the specified condition is violated.
`Reply To Destination`	*String* Optional. Specifies where to log the alerts. **Important:** Ensure that Mediator is configured to send event notifications to the destination(s) you specify here. For details, see *Alerts and Transaction Logging* in the document *Administering webMethods Mediator*.	
	Value	**Description**
		Sends the alerts to the virtual service's Events profile in CentraSite. Prerequisite: You must configure Mediator to communicate with CentraSite (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > CentraSite Communication**). For the procedure, see the section *Configuring Communication*

	`CentraSite`	*with CentraSite* in the document *Administering webMethods Mediator*.
	`Local Log`	Sends the alerts to the server log of the Integration Server on which Mediator is running. Also choose a value in the `Log Level` field: ■ Info: Logs error-level, warning-level, and informational-level alerts. ■ Warn: Logs error-level and warning-level alerts. ■ Error: Logs only error-level alerts. **Important:** The Integration Server Administrator's logging level for Mediator should match the logging level specified for this action (go to **Settings > Logging > Server Logger**).
	`SNMP`	Sends the alerts to CentraSite's SNMP server or a third-party SNMP server. Prerequisite: You must configure the SNMP server destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > Email**). For the procedure, see the section *SNMP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.
	`Email`	Sends the alerts to an SMTP email server, which sends them to the email address(es) you specify here. To specify multiple addresses, use the  button to add rows. Prerequisite: You must configure the SMTP server destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > Email**). For the procedure, see the section *SMTP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.
`Alert Message for Soft Limit`	*String* Optional. Specify a text message to include in the soft limit alert.	
`Alert Message for Hard Limit`	*String* Optional. Specify a text message to include in the hard limit alert.	

Validate Schema

This action validates all XML request and/or response messages against an XML schema referenced in the WSDL.

Mediator can enforce this policy action for messages sent between services. When this policy is set for the virtual service, Mediator validates XML request messages, response messages, or both, against the XML schema referenced in the WSDL.

Input Parameters

<table>
<tr><td rowspan="4">Validate SOAP Message(s)</td><td colspan="2">Object Validates request and/or response messages. You may select both Request and Response.</td></tr>
<tr><td>Value</td><td>Description</td></tr>
<tr><td>Request</td><td>Validate all requests.</td></tr>
<tr><td>Response</td><td>Validate all responses.</td></tr>
</table>

Important: Be aware that Mediator does not remove `wsu:Id` attributes that may have been added to a request by a consumer as a result of security operations against request elements (i.e., signatures and encryptions). In this case, to avoid schema validation failures you would have to add a Request Handling step to the virtual service so that the requests are passed to an XSL transformation file that removes the `wsu:Id` attribute. For details about the Request Handling step, see the section *Virtual Services in CentraSite Control* .

3 Built-In Run-Time Actions Reference for APIs

This section describes the built-in run-time actions that you can include in run-time governance rules for APIs. You use these actions only when you are using the CentraSite Business UI to create run-time policies for APIs. The content is organized under the following sections:

Instructions throughout the remainder of this guide use the term "API" when referring to the Virtual Services, Virtual XML Services and Virtual REST Services; and the term "client" when referring to the Consumer Applications in general.

Summary of the Run-Time Actions

You can include the following kinds of built-in run-time actions in the run-time governance rules for APIs:

Request Handling Actions

Request Handling is the process of receiving and transforming the incoming message from a client into the custom format as expected by the native API.

Require HTTP / HTTPS	Specifies the protocol (HTTP or HTTPS), SOAP format (for a SOAP-based API), and the HTTP methods (for a REST-based API) to be used to accept and process the requests.
Require JMS	Specifies the JMS protocol to be used for the API to accept and process the requests.
Request Transformation	Invokes an XSL transformation in the SOAP request before it is submitted to the native API.
Invoke webMethods Integration Server	Invokes a webMethods Integration Server service to pre-process the request before it is submitted to the native API.

Policy Enforcement Actions

Policy Enforcement is the process of enforcing the adherence to real-time policy compliance identifying/authenticating, monitoring, auditing, and measuring and collecting result statistics for an API.

Mediator provides the following categories of policy enforcement actions:

Authentication Actions

Authentication actions verify that the API client has the proper credentials to access an API.

HTTP Basic Authentication	Uses HTTP basic authentication to verify the client's authentication credentials contained in the request's Authorization header against the Integration Server's user account.
NTLM Authentication	Uses NTLM authentication to verify the client's authentication credentials contained in the request's Authorization header against the Integration Server's user account.
OAuth2 Authentication	Uses OAuth2 authentication to verify the client's authentication credentials contained in the request's Authorization header against the Integration Server's user account.

JMS Routing Actions

JMS Routing actions route the incoming message to an API over JMS. For example, to a JMS queue where an API can then retrieve the message asynchronously.

JMS Routing Rule	Specifies a JMS queue to which the Mediator is to submit the request, and the destination to which the native API is to return the response.
Set Message Properties	Specifies JMS message properties to authenticate client requests before submitting to the native APIs.
Set JMS Headers	Specifies JMS headers to authenticate client requests before submitting to the native APIs.

Logging and Monitoring Actions

Logging and Monitoring actions monitor and collect information about the number of messages that were processed successfully or failed, the average execution time of message processing, and the number of alerts associated with an API.

Log Invocation	Logs request/response payloads to a destination you specify.
Monitor Service Level Agreement	Specifies a Service Level Agreement (SLA), which is set of conditions that define the level of performance that a specified client should expect from an API.
Monitor Service Performance	This action provides the same functionality as *Monitor Service Level Agreement* but this action is different because it enables you to monitor the API's run-time performance for all clients. This action monitors a user-specified set of run-time performance conditions for an API, and sends alerts to a specified destination when these performance conditions are violated.

Routing Actions

Routing actions route the incoming message (e.g., directly to the API, or routed according to the routing rules, or routed to a pool of servers for the purpose of load balancing and failover handling).

Straight Through Routing	Routes the requests directly to a native endpoint that you specify.
Context Based Routing	Route requests to different endpoints based on specific values that appear in the request message.
Content Based Routing	Route requests to different endpoints based on specific criteria that you specify.
Load Balancing and Failover Routing	Routes the requests across multiple endpoints.
Set Custom Headers	Specifies the HTTP headers to process the requests.

Security Actions

Security actions provide client validation (through WSS X.509 certificates, WSS username tokens etc.), confidentiality (through encryption) and integrity (through signatures) for request and response messages.

For the client validation, Mediator maintains a list of consumer applications specified in CentraSite that are authorized to access the API published to Mediator. Mediator synchronizes this list of consumer applications through a manual process initiated from CentraSite.

Generally speaking there are two different lists of consumers in the Mediator:

- **List of Registered Consumers**

  List of users and consumer applications (represented as Application assets) who are registered as consumers for the API in CentraSite, and available in the Mediator.

For more information on how to register as consumer for an API, refer to the online documentation section *Registering as Consumer for an API* in the document *Virtualizing APIs Using the CentraSite Business UI*.

- **List of Global Consumers**

  List of all users and consumer applications (represented as consumers) available in the Mediator.

  For more information on how to create a consumer application asset, refer to the online documentation section *Managing Consumer Applications for an API > Creating a Consumer Application* in the document *Virtualizing APIs Using the CentraSite Business UI*.

  For more information on how to publish a consumer application to Mediator, refer to the online documentation section *Managing Consumer Applications for an API > Publishing a Consumer Application* in the document *Virtualizing APIs Using the CentraSite Business UI*.

Mediator provides "Evaluate" actions that you can include in a message flow to identify and/or validate clients, and then configure their parameters to suit your needs. You use these "Evaluate" actions to perform the following actions:

- Identify the clients who are trying to access the APIs (through IP address or hostname).
- Validate the client's credentials.

Evaluate Client Certificate for SSL Connectivity	Mediator validates the client's certificate that the client submits to the API in CentraSite. The client certificate that is used to identify the client is supplied by the client to the Mediator during the SSL handshake over the transport layer.
Evaluate Hostname	■ Mediator will try to identify the client against either the Registered Consumers list (the list of registered consumers in Mediator) or the Global Consumers list (the list of available consumers in Mediator). ■ Mediator will try to validate the client's hostname against the specified list of consumers in the Integration Server on which Mediator is running.
Evaluate HTTP Basic Authentication	■ Mediator will try to identify the client against either the Registered Consumers list (the list of registered consumers in Mediator) or the Global Consumers list (the list of available consumers in Mediator). ■ Mediator will try to validate the client's authentication credentials contained in the request's Authorization header against the specified list of consumers in the Integration Server on which Mediator is running.
Evaluate IP Address	■ Mediator will try to identify the client against either the Registered Consumers list (the list of registered consumers in Mediator) or the Global Consumers list (the list of available consumers in Mediator). ■ Mediator will try to validate the client's IP address against the specified list of consumers in the Integration Server on which Mediator is running.

Evaluate WSS Username Token	*Applicable only for SOAP APIs.* ■ Mediator will try to identify the client against either the Registered Consumers list (the list of registered consumers in Mediator) or the Global Consumers list (the list of available consumers in Mediator). ■ Mediator will try to validate the client's WSS username token against the specified list of consumers in the Integration Server on which Mediator is running.
Evaluate WSS X.509 Certificate	*Applicable only for SOAP APIs.* ■ Mediator will try to identify the client against either the Registered Consumers list (the list of registered consumers in Mediator) or the Global Consumers list (the list of available consumers in Mediator). ■ Mediator will try to validate the client's WSS X.509 token against the specified list of consumers in the Integration Server on which Mediator is running.
Evaluate XPath Expression	■ Mediator will try to identify the client against either the Registered Consumers list (the list of registered consumers in Mediator) or the Global Consumers list (the list of available consumers in Mediator). ■ Mediator will try to validate the client's XPath expression against the specified list of consumers in the Integration Server on which Mediator is running.
Require Encryption	*Applicable only for SOAP APIs.* Requires that a request's XML element (which is represented by an XPath expression) be encrypted.
Require Signing	*Applicable only for SOAP APIs.* Requires that a request's XML element (which is represented by an XPath expression) be signed.
Require SSL	*Applicable only for SOAP APIs.* Requires that requests be sent via SSL client certificates.
Require Timestamps	*Applicable only for SOAP APIs.* Requires that timestamps be included in the request header. Mediator checks the timestamp value against the current time to ensure that the request is not an old message. This serves to protect your system against attempts at message tampering, such as replay attacks.
Require WSS SAML Token	*Applicable only for SOAP APIs.* Uses a WSS Security Assertion Markup Language (SAML) assertion token to validate API clients.

Traffic Management Action

Throttling Traffic Optimization	Limits the number of service invocations during a specified time interval, and sends alerts to a specified destination when the performance conditions are violated. You can use this action to avoid overloading the back-end services and their infrastructure, to limit specific clients in terms of resource usage, etc.

Validation Action

Validate Schema	Validates all XML request and/or response messages against an XML schema referenced in the WSDL.

Response Handling Actions

Response Handling is the process of transforming the response message coming from the native API into the custom format as expected by the client.

Response Transformation	Invokes an XSL transformation in the response payloads from XML format to the format required by the client.
Invoke webMethods Integration Server	Invokes a webMethods Integration Server service to process the response from the native API before it is returned to the client.

Error Handling Action

Error Handling is the process of passing an exception message which has been issued as a result of a run-time error to take any necessary actions.

Custom SOAP Response Message	Returns a custom error message (and/or the native provider's service fault content) to the client when the native provider returns a service fault.

The watt.server.auth.skipForMediator Property

This property specifies whether Integration Server authenticates requests for Mediator. You must set this property to true.

No request to Mediator should be authenticated by Integration Server. Instead, authentication should be handled by Mediator. Thus, to enable Mediator to authenticate requests, you must set skipForMediator to true (by default it is false).

When this parameter is set to true, Integration Server skips authentication for Mediator requests and allows the user credentials (of any type) to pass through so that Mediator can authenticate

them. If you change the setting of this parameter, you must restart Integration Server for the changes to take effect.

To set skipForMediator to true

1 In the Integration Server Administrator, click **Settings > Extended**.

2 Click **Show and Hide Keys**.

Look for the `watt.server.auth.skipForMediator` property and ensure it is set to true.

3 If the `watt.server.auth.skipForMediator` property is not present, add it as follows:

1. Click **Edit Extended Settings**.
2. Type `watt.server.auth.skipForMediator=true` on a separate line.
3. Click **Save**.
4. Restart Integration Server.

Effective Policies

When you publish an API to Mediator, CentraSite automatically validates the API's policy enforcement workflow to ensure that:

CentraSite will inform you of any violation, and you will need to correct the violations before publishing the API.

When you publish an API to Mediator, CentraSite combines the actions specified within the proxy API's enforcement definition, and generates what is called the effective policy for the API. For example, suppose your API is configured with two run-time actions: one that performs a logging action and another that performs a security action. When you publish the API, CentraSite automatically combines the two actions into one effective policy. The effective policy, which contains both the logging action and the security action, is the policy that CentraSite actually publishes to Mediator with the API.

When CentraSite generates the effective policy, it validates the resulting action list to ensure that:

- Any action that appears in a single message flow multiple times is allowed to appear multiple times.

  For those actions that can appear in a message flow only once (for example, Evaluate IP Address), Mediator will choose only one, which might cause problems or unintended results.
- All action dependencies are properly met. That is, some actions must be used in conjunction with another particular action.

If the list contains conflicts or inconsistencies, CentraSite resolves them according to Policy Resolution Rules.

The effective policy that CentraSite produces for an API is contained in an object called a virtual service definition (VSD). The VSD is given to Mediator when you publish the API. After you publish an API to Mediator, you can view its VSD (and thus examine the effective policy that CentraSite generated for it) from the Mediator user interface.

The following table shows:

- Action is WS-Security Policy 1.2 compliant.
- Action dependencies (that is, whether an action must be used in conjunction with another particular action).
- Action exclusives (that is, whether an action cannot be used in conjunction with another particular action).
- Action occurrences (that is, whether an action can occur once or multiple times within a message flow stage).

Action	WS-Security Policy Compliant	Dependency Requirement	Mutually Exclusive	Can include multiple times in a policy if the selection criteria is combined using an *AND* operator (not *OR*)?
Require HTTP / HTTPS	No	None	Require JMS	Once
Require JMS	No	None	Require HTTP / HTTPS	Once
Request Transformation	No	None	None	Multiple
Invoke webMethods Integration Server	No	None	None	Multiple
Require SSL	Yes	None	None	Once
Require WSS SAML Token	Yes	None	None	Once
Require Signing	Yes	None	None	Once
Require Encryption	Yes	None	None	Once
Require Timestamps	Yes	At least one of the following actions: ■ Evaluate WSS Username Token	None	Once

Action	WS-Security Policy Compliant	Dependency Requirement	Mutually Exclusive	Can include multiple times in a policy if the selection criteria is combined using an *AND* operator (not *OR*)?
		■ Evaluate WSS X.509 Certificate ■ Require Signing ■ Require Encryption		
Evaluate OAuth2 Token	No	None	None	Once
Evaluate HTTP Basic Authentication	No	None	■ Evaluate OAuth2 Authentication ■ OAuth2 Authentication ■ NTLM Authentication	Once
Evaluate WSS Username Token	Yes	None	None	Once
Evaluate WSS X.509 Certificate	Yes	None	None	Once
Evaluate IP Address	No	None	None	Once
Evaluate XPath Expression	No	None	None	Once
Evaluate Hostname	No	None	None	Once
Evaluate Client Certificate for SSL Connectivity	No	None	None	Once
Log Invocations	No	None	None	Once
Monitor Service Level Agreement	No	At least one of the "Evaluate" actions, or the Require WSS SAML Token.	None	Multiple
Monitor Service Performance	No	At least one of the "Evaluate" actions, or the Require WSS SAML Token.	None	Multiple
Throttling Traffic Optimization	No	At least one of the "Evaluate" actions, or the Require WSS SAML Token, provided the **Alert for Consumer**	None	Multiple

Action	WS-Security Policy Compliant	Dependency Requirement	Mutually Exclusive	Can include multiple times in a policy if the selection criteria is combined using an *AND* operator (not *OR*)?
		Applications value is specified.		
Validate Schema	No	None	None	Once
HTTP Basic Authentication	No	At least one of the "Routing" actions.	■ NTLM Authentication ■ OAuth2 Authentication ■ JMS Routing Rule ■ Evaluate OAuth2 Authentication	Once
NTLM Authentication	No	At least one of the "Routing" actions.	■ HTTP Basic Authentication ■ OAuth2 Authentication ■ JMS Routing Rule ■ Evaluate HTTP Basic Authentication ■ Evaluate OAuth2 Authentication	Once
OAuth2 Authentication	No	At least one of the "Routing" actions.	■ HTTP Basic Authentication ■ NTLM Authentication ■ JMS Routing Rule ■ Evaluate HTTP Basic Authentication	Once
JMS Routing Rule	No	None	None of the "Routing" actions.	Once
Set Message Properties	No	JMS Routing Rule	None of the "Routing" actions.	Once
Set JMS Headers	No	JMS Routing Rule	None of the "Routing" actions.	Once
Straight Through Routing	No	None	None of the other "Routing" actions.	Once

Action	WS-Security Policy Compliant	Dependency Requirement	Mutually Exclusive	Can include multiple times in a policy if the selection criteria is combined using an *AND* operator (not *OR*)?
Content Based Routing	No	None	None of the other "Routing" actions.	Once
Load Balancing and Failover Routing	No	None	None of the other "Routing" actions.	Once
Context Based Routing	No	None	None of the other "Routing" actions.	Once
Set Custom Headers	No	At least one of the "Routing" actions.	None	Once
Response Transformation	No	None	None	Multiple
Invoke webMethods Integration Server	No	None	None	Multiple
Custom SOAP Response Message	No	None	None	Once

Effective Policies

Usage Cases for Identifying/Authenticating Clients

When deciding which type of identifier to use to identify a client, consider the following points:

- Whatever identifier you choose to identify a client, it must be unique to the application. Identifiers that represent user names are often not suitable because the identified users might submit requests for multiple APIs.
- Identifying applications by IP address or host name is often a suitable choice, however, it does create a dependency on the network infrastructure. If a client moves to a new machine, or its IP address changes, you must update the identifiers in the application asset.
- Using X.509 certificates or a custom token that is extracted from the SOAP message itself (using an XPATH expression), is often the most trouble-free way to identify a client.

Following are some common combinations of actions used to authenticate/identify clients.

- **Scenario 1: Identify clients by IP address or host name**
  - The simplest way to identify clients is to use the **Evaluate IP Address** action.
- **Scenario 2: Authenticate clients by HTTP authentication token**
  Use the following actions:
  - **Evaluate HTTP Basic Authentication** to identify clients using the token derived from the HTTP Header.
  - **HTTP Basic Authentication**.
- **Scenario 3: Authenticate clients by WS-Security authentication token**
  Use the following action:
  - **Evaluate WSS Username Token** action to identify clients using the token derived from the WSS Header.
- **Scenario 4: Authenticate clients by WSS X.509 certificate**
  - **Evaluate WSS X.509 Certificate** action to identify clients using the WSS X.509 certificate.
  - **Require SSL** action.

Run-Time Actions Reference

This section provides an alphabetic list of the built-in run-time actions you can include in the run-time governance rules for APIs:

- Response Transformation
- Request Transformation
- Require Encryption
- Require HTTP / HTTPS
- Require JMS
- Require Signing
- Require SSL
- Require Timestamps
- Require WSS SAML Token
- Set Custom Headers
- Set JMS Headers
- Set Message Properties
- Straight Through Routing
- Throttling Traffic Optimization
- Validate Schema

Content Based Routing

If you have a native API that is hosted at two or more endpoints, you can use the Content Based Routing to route specific types of messages to specific endpoints.

You can route messages to different endpoints based on specific values that appear in the request message.

When this action is configured for a proxy API, the requests are routed according to the routing rules you create. That is, they are routed based on the successful evaluation of one or more XPath expressions that are constructed utilizing the content of the request payload. For example, a routing rule might allow requests for half of the methods of a particular service to be routed to Endpoint A, and the remaining methods to be routed to Endpoint B.

Input Parameters

`Route To`	*URI. Mandatory.* Enter the URL of the native API endpoint to route the request to in case all routing rules evaluate to False. For example:
	
	`http://mycontainer/creditCheckService`
	
	Click the **Configure Endpoint Properties**  icon (next to the **Route To** field) if you want to configure a set of properties for the specified endpoint.
	
	Alternatively, Mediator offers "Local Optimization" capability if the native endpoint is hosted on the same Integration Server as webMethods Mediator. With local optimization, API invocation happens in-memory and not through a network hop.
	
	Specify the native API in either of the following forms:
	
	`local://<Service-full-path>`

<table>
<tr><td></td><td>OR

local://<server>:<port>/ws/<Service-full-path>

For example:

local://MyAPIFolder:MyLocalAPI

which points to the endpoint API MyLocalAPI which is present under the folder MyAPIFolder in Integration Server.</td></tr>
<tr><td>Add Routing Rule button</td><td>Click the Add Routing Rule button and complete the Add Routing Rule dialog box as follows.

1. In the XPath Expression field, specify an argument to evaluate the XPath expression contained in the request.
2. To add a custom Namespace, specify a name and value for the namespace in the Prefix and URI fields. If you need to add additional rows, use the plus button.
3. In the Route To field, specify the URL of the native API to route the request to, if the rule criteria are met.
4. Click the Configure Endpoint Properties icon (next to the Route To field) if you want to configure a set of properties for the specified endpoint individually.
5. Click OK.</td></tr>
<tr><td>Configure Endpoint Properties icon</td><td>Optional. This icon displays the Endpoint Properties dialog box that enables you to configure a set of properties for the Mediator to route incoming requests to the native API as follows:

SOAP Optimization Method

Only for SOAP-Based APIs. Mediator can use the following optimization methods to parse SOAP requests to the native API:

<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td>MTOM</td><td>Mediator will use the Message Transmission Optimization Mechanism (MTOM) to parse SOAP requests to the API.</td></tr>
<tr><td>SwA</td><td>Mediator will use the SOAP with Attachment (SwA) technique to parse SOAP requests to the API.</td></tr>
<tr><td>None</td><td>Default. Mediator will not use any optimization method to parse the SOAP requests to the API.</td></tr>
</table>
Note:

1. Bridging between SwA and MTOM is not supported. If a client sends a SwA request, Mediator can only forward SwA to the native API. The same is true for MTOM, and applies to responses received from the native API. That is, a SwA or MTOM response received by Mediator from a native API will be forwarded to the client using the same format it received.</td></tr>
</table>

		2. When sending SOAP requests that do not contain a MTOM or SWA attachment to a native API that returns an MTOM or SWA response, the request 'Accept' header must be set to 'multipart/related'. This is necessary so Mediator knows how to parse the response properly.
	`HTTP Connection Timeout`	*Number. Optional.* The time interval (in seconds) after which a connection attempt will timeout. If a value 0 is specified (or if the value is not specified), Mediator will use the value specified in the `Connection Timeout` field (go to **Integration Server Administrator > Settings > Extended**). *Default*: 30 seconds.
	`Read Timeout`	*Number. Optional.* The time interval (in seconds) after which a socket read attempt will timeout. If a value 0 is specified (or if the value is not specified), Mediator will use the value specified in the `Read Timeout` field (Open the Integration Server Administrator. Go to **> Settings > Extended**.). *Default*: 30 seconds.
	`SSL Configuration`	*Optional.* To enable SSL client authentication that Mediator will use to authenticate incoming requests for the native API, you must specify values for both the Client Certificate Alias field and the IS Keystore Alias field. If you specify a value for only one of these fields, a deployment error will occur. **Note:** SSL client authentication is optional; you may leave both fields blank. Prerequisite: You must set up the key alias and keystore properties in the Integration Server. For the procedure, see the section *Securing Communications with the Server* in the document*webMethods Integration Server Administrator's Guide*. You will use these properties to specify the following fields:
		Value / **Description**
		`Client Certificate Alias` — *Mandatory*. The client's private key to be used for performing SSL client authentication.
		`IS Keystore Alias` — *Mandatory*. The keystore alias of the instance of Integration Server on which Mediator is running. This value (along with the value of Client Certificate Alias) will be used for performing SSL client authentication.
	`WS Security Header`	Only for SOAP-Based APIs. Indicates whether Mediator should pass the WS-Security headers of the incoming requests to the native API.
		Value / **Description**

		`Remove processed security headers`	Removes the security header if it is processed by Mediator (i.e., if Mediator processes the header according to the API's security run-time action). Note that Mediator will not remove the security header if both of the following conditions are true: 1) Mediator did not process the security header, and 2) the mustUnderstand attribute of the security header is 0/false).
		`Pass all security headers`	*Default*. Passes the security header, even if it is processed by Mediator (i.e., even if Mediator processes the header according to the API's security action).

Context Based Routing

If you have a native API that is hosted at two or more endpoints, you can use the Context Based Routing to route specific types of messages to specific endpoints.

When this action is configured for a proxy API, the requests are routed according to the routing rules you create. A routing rule specifies where requests should be routed, and the criteria by which they should be routed there. For example, requests can be routed according to certain clients, certain dates/times, or according to requests that exceed/fall below a specified metric (Total Count, Success Count, Fault Count, etc.). You can create one or more rules.

Input Parameters

`Default Route To`	*URI. Mandatory.* Enter the URL of the native API endpoint to route the request to in case all routing rules evaluate to False. For example:
	
	`http://mycontainer/creditCheckService`
	
	Click the **Configure Endpoint Properties** icon (next to the **Route To** field) if you want to configure a set of properties for the specified endpoint.
	
	Alternatively, Mediator offers "Local Optimization" capability if the native endpoint is hosted on the same Integration Server as webMethods Mediator. With local optimization, API invocation happens in-memory and not through a network hop.
	
	Specify the native API in either of the following forms:
	
	`local://<Service-full-path>`
	
	OR
	
	`local://<server>:<port>/ws/<Service-full-path>`

	For example: `local://MyAPIFolder:MyLocalAPI` which points to the endpoint API `MyLocalAPI` which is present under the folder `MyAPIFolder` in Integration Server.
`Add Routing Rule` button	Click the **Add Routing Rule** button and complete the **Add Routing Rule** dialog box as follows. 1. In the **Name** field, specify a name for the routing rule. 2. In the **Condition** panel, specify the following as required: a. In the **Variable** column, select **Time**, **IP Address Range**, **Date**, **Consumer**, **Predefined Context Variable** or **Custom Context Variable**. For more information, see *Using Context Variables in APIs*. b. In the **Value** column, specify an applicable value. For **Date** choose **Before**, **After** or **Equal To** and enter a date. For **Time** choose **Before** or **After** and enter a time. For **IP Address**, enter numeric values for **Between** and **And**. For **Consumer**, enter a consumer application name in the text box. For **Predefined Context Variable** or **Custom Context Variable**, choose the **String** or **Integer** data type. Select a predefined variable name or custom variable name from the drop-down list. For **String,** choose **Equal To** or **Not Equal To** and enter a value. For **Integer**, choose **Greater Than**, **Less Than**, **Not Equal To**, **Equal To** or and enter a value. **Note:** i. For the list of the predefined context variables, see *Using Context Variables in APIs*. ii. The predefined context variable PROTOCOL_HEADER is not available in the drop-down list; to include PROTOCOL_HEADER in the rule, define the variable as Custom Context Variable. For more information, see *The API for Context Variables*. iii. If you define a custom context variable in the routing rule, you must write a *webMethods IS service* and invoke it in the API's Context Based Routing action. In this Integration Server service, use the API to get/set the custom context variable. For more information, see *The API for Context Variables*. If you need to specify multiple variables, use the plus button to add rows. c. If you have more than one routing rule, choose an operator for the expression: **AND** or **OR** (the default) . 3. In the **Route To** field, specify the URL of the native API endpoint to route the request to, if the rule criteria are met. 4. Click the **Configure Endpoint Properties**  icon (next to the **Route To** field) if you want to configure a set of properties for the specified endpoint individually. 5. Click **OK**.
`Configure Endpoint`	*Optional.* This icon displays the **Endpoint Properties** dialog box that enables you to configure a set of properties for the Mediator to route incoming requests to the native API as follows:

<table>
<tr><td rowspan="8">Properties  icon</td><td rowspan="8">SOAP Optimization Method</td><td colspan="2">Only for SOAP-Based APIs. Mediator can use the following optimization methods to parse SOAP requests to the native API:</td></tr>
<tr><td>Value</td><td>Description</td></tr>
<tr><td>MTOM</td><td>Mediator will use the Message Transmission Optimization Mechanism (MTOM) to parse SOAP requests to the API.</td></tr>
<tr><td>SwA</td><td>Mediator will use the SOAP with Attachment (SwA) technique to parse SOAP requests to the API.</td></tr>
<tr><td>None</td><td>Default. Mediator will not use any optimization method to parse the SOAP requests to the API.</td></tr>
<tr><td colspan="2">Note:

1. Bridging between SwA and MTOM is not supported. If a client sends a SwA request, Mediator can only forward SwA to the native API. The same is true for MTOM, and applies to responses received from the native API. That is, a SwA or MTOM response received by Mediator from a native API will be forwarded to the client using the same format it received.
2. When sending SOAP requests that do not contain a MTOM or SWA attachment to a native API that returns an MTOM or SWA response, the request 'Accept' header must be set to 'multipart/related'. This is necessary so Mediator knows how to parse the response properly.</td></tr>
<tr><td>HTTP Connection Timeout</td><td colspan="2">Number. Optional. The time interval (in seconds) after which a connection attempt will timeout. If a value 0 is specified (or if the value is not specified), Mediator will use the value specified in the Connection Timeout field (go to Integration Server Administrator > Settings > Extended). Default: 30 seconds.</td></tr>
<tr><td>Read Timeout</td><td colspan="2">Number. Optional. The time interval (in seconds) after which a socket read attempt will timeout. If a value 0 is specified (or if the value is not specified), Mediator will use the value specified in the Read Timeout field (Open the Integration Server Administrator. Go to > Settings > Extended.). Default: 30 seconds.</td></tr>
<tr><td>SSL Configuration</td><td colspan="2">Optional. To enable SSL client authentication that Mediator will use to authenticate incoming requests for the native API, you must specify values for both the Client Certificate Alias field and the IS Keystore Alias field. If you specify a value for only one of these fields, a deployment error will occur.

Note: SSL client authentication is optional; you may leave both fields blank.</td></tr>
</table>

<table>
<tr><td></td><td></td><td colspan="2">Prerequisite: You must set up the key alias and keystore properties in the Integration Server. For the procedure, see the section Securing Communications with the Server in the documentwebMethods Integration Server Administrator's Guide.

You will use these properties to specify the following fields:</td></tr>
<tr><td></td><td></td><td>Value</td><td>Description</td></tr>
<tr><td></td><td></td><td>Client Certificate Alias</td><td>Mandatory. The client's private key to be used for performing SSL client authentication.</td></tr>
<tr><td></td><td></td><td>IS Keystore Alias</td><td>Mandatory. The keystore alias of the instance of Integration Server on which Mediator is running. This value (along with the value of Client Certificate Alias) will be used for performing SSL client authentication.</td></tr>
<tr><td></td><td>WS Security Header</td><td colspan="2">Only for SOAP-Based APIs. Indicates whether Mediator should pass the WS-Security headers of the incoming requests to the native API.</td></tr>
<tr><td></td><td></td><td>Value</td><td>Description</td></tr>
<tr><td></td><td></td><td>Remove processed ↩ security headers</td><td>Default. Removes the security header if it is processed by Mediator (i.e., if Mediator processes the header according to the API's security run-time action). Note that Mediator will not remove the security header if both of the following conditions are true: 1) Mediator did not process the security header, and 2) the mustUnderstand attribute of the security header is 0/false).</td></tr>
<tr><td></td><td></td><td>Pass all security headers</td><td>Passes the security header, even if it is processed by Mediator (i.e., even if Mediator processes the header according to the API's security action).</td></tr>
</table>

Custom SOAP Response Message

This action returns a custom error response (and/or the native provider's service fault content) to the client when the native provider returns a service fault. Alternatively, you can configure global error responses for all APIs, using Mediator's Service Fault Configuration page (see *Configuring Global Service Fault Responses* in the document *Administering webMethods Mediator*).

Input Parameters

`Failure Message`	*String.* Specify the custom failure message to the client.
`Send Native SOAP Fault Message`	When the parameter is enabled, Mediator sends the native SOAP / REST failure message to the client. When you enable this parameter, the `Failure Message` is ignored when a fault is returned by the native API provider. (Faults returned by internal Mediator exceptions will still be handled by the `Failure Message`.)
`Pre-processing webMethods IS Service`	*String. Optional.* Invokes one or more webMethods IS services to manipulate the response message from the native API before it is returned to the consuming application. The IS service will have access to the response message context (the axis2 MessageContext instance) before it is updated with the custom error message. For example, you might want to send emails or perform custom alerts based on the response payload.
`Post-processing webMethods IS Service`	*String.* Optional. Invokes one or more webMethods IS services to manipulate the API fault after the Custom SOAP Response Message action is invoked. The IS service will have access to the entire API fault and the custom error message. You can make further changes to the fault message structure, if needed.

Failure Messages

The failure message is returned in both of the following cases:

- When a failure is returned by the native API provider.

  In this case, the `$ERROR_MESSAGE` variable in the failure message will contain the message produced by the provider's exception that caused the error. This is equivalent to the `getMessage` call on the Java Exception.
- When a failure is returned by internal Mediator exceptions (such as policy violation errors, timeouts, etc.).

  In this case, `$ERROR_MESSAGE` will contain the error message generated by Mediator.

Alternatively, you can configure global failure messages for *all* APIs, using Mediator's Service Fault Configuration page, as described in the document *Administering webMethods Mediator*.

Mediator returns the following failure message to the consuming application:

```
Mediator encountered an error:$ERROR_MESSAGE while executing operation:$OPERATION ↩
service:$SERVICE at time:$TIME on date:$DATE. The client ip was:$CLIENT_IP. The ↩
current user:$USER. The consumer application:$CONSUMER_APPLICATION".
```

The precedence of the Failure Message configurations is as follows:

- If you configure a Custom SOAP Response Message action for an API, the failure message configurations take precedence over any settings on the global Service Fault Configuration page.

- If you do not configure a Custom SOAP Response Message action for an API, the settings on the Service Fault Configuration page take precedence.

The default failure message contains predefined fault handler variables (`$ERROR_MESSAGE`, `$OPERATION`, etc.) which are described in the table below.

You can customize the default failure message using the following substitution variables, where Mediator replaces the variable reference with the real content at run time:

- The predefined context variables listed in *The Predefined Context Variables*.
- Custom context variables that you declare using Mediator's API (see *The API for Context Variables*).

Note: If you want to reference a custom context variable that you have already defined in a **Context Based Routing** action (as opposed to one you have declared using Mediator's API), then you must add the prefix `$mx` to the variable name in order to reference the variable. For example, if you defined the variable TAXID, you would reference it as `$mx:TAXID`.

The fault handler variables are described below.

Note: If no value is defined for a fault handler variable, then the returned value will be the literal string "null". For example, `$CONSUMER_APPLICATION` will always be "null" if the service's policy does not contain at least one of the "Evaluate" actions.

Fault Handler Variable	Description
`$ERROR_MESSAGE`	The error message produced by the exception that is causing the error. This is equivalent to the getMessage call on the Java Exception. This maps to the **faultString** element for SOAP 1.1 or the **Reason** element for SOAP 1.2 catch.
`$OPERATION`	The operation that was invoked when this error occurred.
`$SERVICE`	The service that was invoked when this error occurred.
`$TIME`	The time (as determined on the Container side) at which the error occurred.
`$DATE`	The date (as determined on the Container side) at which the error occurred.
`$CLIENT_IP`	The IP address of the client invoking the service. This might be available for only certain invoking protocols, such as HTTP(S).
`$USER`	The currently authenticated user. The user will be present only if the Transport/SOAP Message have user credentials.
`$CONSUMER_APPLICATION`	The currently identified consumer application (client).

Evaluate Client Certificate for SSL Connectivity

If you have a native API that requires to authenticate a client to the Integration Server using the Secure Sockets Layer (SSL) client authentication, you can use the Evaluate Client Certificate action to extract the client's identity certificate, and verify the client's identity (certificate-based authentication).

This form of authentication does not occur at the message layer using a user ID and password or tokens. This authentication occurs during the connection handshake using SSL certificates.

This action extracts the client identity certificate supplied by the client to the Mediator during the SSL handshake over the Transport layer. For example, when you have configured this action for a proxy API, the PEP extracts the certificate from the Transport layer. In order to identify clients by transport-level certificates, the run-time communication between the client and the Mediator must be over HTTPS and the client must pass a valid certificate.

To use this action, the following prerequisites must be met:

- In Integration Server, create a keystore and truststore, as described in *Securing Communications with the Server* in the *webMethods Integration Server Administrator's Guide*.
- In Integration Server, create an HTTPS port, as described in *Configuring Ports* in the *webMethods Integration Server Administrator's Guide*.
- Configure Mediator by setting the HTTPS Ports Configuration parameter, as described in *Configuring Mediator > Ports Configuration* in the document *Administering webMethods Mediator*.

Mediator rejects requests that do not include a client certificate during the SSL handshake over the Transport layer.

If Mediator cannot identify the client, Mediator fails the request and generates a Policy Violation event.

Input Parameters

<table>
<tr><td><code>Identify Consumer</code></td><td>String. The list of consumers against which the client certificate should be validated for identifying requests from a particular client.

<table><tr><th>Value</th><th>Description</th></tr><tr><td><code>Registered Consumers</code></td><td>Mediator will try to verify the client identify certificate against the list of consumer applications who are registered as consumers for the specified API.</td></tr></table></td></tr>
</table>

	`Global Consumers`	*Default.* Mediator will try to verify the client identify certificate against a list of all global consumers available in the Mediator.

Evaluate Hostname

If you have a native API that requires to authenticate a client to the Integration Server using the hostname, you can use the Evaluate Hostname action to extract the client's hostname from the HTTP request header, and verify the client's identity.

This action extracts the specified hostname from an incoming request and locates the client defined by that hostname. For example, when you have configured this action for an API, the PEP extracts the hostname from the request's HTTP header at run time and searches its list of consumers for the client defined by the hostname.

Mediator will evaluate the incoming request to identify and validate that the client's request originated from a particular host machine.

Mediator rejects requests that do not include the hostname of an Integration Server user.

If Mediator cannot identify the client, Mediator fails the request and generates a Policy Violation event.

Input Parameters

`Identify Consumer`	*String.* The list of consumers against which the hostname should be validated for identifying requests from a particular client.	
	Value	**Description**
	`Registered Consumers`	Mediator will try to verify the client's hostname against the list of consumer applications who are registered as consumers for the specified API.
	`Global Consumers`	*Default.* Mediator will try to verify the client's hostname against a list of all global consumers available in the Mediator.

Evaluate HTTP Basic Authentication

If you have a native API that requires to authenticate a client to the Integration Server using the HTTP Basic Authentication, you can use the Evaluate HTTP Basic Authentication action to extract the client's credentials (user ID and password) from the Authorization request header, and verify the client's identity.

This action uses HTTP Basic authentication to verify the client's authentication credentials contained in the request's Authorization header. When this action is configured for an API, Mediator validates the credentials against the list of consumers available in the Integration Server on which Mediator

is running. If you chosen the checkbox `Authenticate User` using the HTTP Basic Authentication, this type of client authentication is referred to as “preemptive authentication”.

If the user/password value in the Authorization header cannot be authenticated as a valid Integration Server user (or if the Authorization header is not present in the request), a 500 SOAP fault is returned, and the client is presented with a security challenge. If the client successfully responds to the challenge, the user is authenticated. This type of client authentication is referred to as “non-preemptive authentication”. If the client does not successfully respond to the challenge, a 401 “WWW-Authenticate: Basic” response is returned and the invocation is not routed to the policy engine.

If you choose to omit the `Authenticate User` parameter (and regardless of whether an Authorization header is present in the request or not), then Mediator forwards the request to the native API, without attempting to authenticate the request.

In the case where a client sends a request with transport credentials (HTTP Basic Authentication) and message credentials (WSS Username Token or WSS X.509 Token), the message credentials take precedence over the transport credentials when Integration Server determines which credentials it should use for the session. For more information, see **Evaluate WSS Username Token** and **Evaluate WSS X.509 Certificate**.

If Mediator cannot identify the client, Mediator fails the request and generates a Policy Violation event.

Input Parameters

<table>
<tr><td rowspan="4">Identify Consumer</td><td colspan="2">String. The list of consumers against which authentication credentials (user ID and password) should be validated for identifying requests from a particular client.</td></tr>
<tr><td>Value</td><td>Description</td></tr>
<tr><td>Registered Consumers</td><td>Mediator will try to verify the client's credentials against the list of consumer applications who are registered as consumers for the specified API.</td></tr>
<tr><td>Global Consumers</td><td>Default. Mediator will try to verify the client's credentials against a list of all global consumers available in the Mediator.</td></tr>
<tr><td></td><td>Do Not Identify</td><td>Mediator forwards the request to the native API, without attempting to verify client's credentials in incoming request.</td></tr>
<tr><td>Authenticate User</td><td colspan="2">Use this checkbox to specify the users who can access the APIs. If you select the checkbox, Mediator allows only the users specified in the Identify Consumer parameter to access the APIs. If you do not select the checkbox, Mediator allows all users to access the API. In this case, do not configure the Identify Consumer parameter.

Note: If you have selected theAuthenticate User option, the client that connects to the API must have an Integration Server user account.</td></tr>
</table>

Evaluate IP Address

If you have a native API that requires to authenticate a client to the Integration Server using the IP address, you can use the Evaluate IP Address action to extract the client's IP address from the HTTP request header, and verify the client's identity.

This action extracts the specified IP address from an incoming request and locates the client defined by that IP address. For example, when you have configured this action for a proxy API, the PEP extracts the IP address from the request's HTTP header at run time and searches its list of consumers for the client defined by the IP address.

Mediator will evaluate the incoming request to identify and validate that the client's request originated from a particular IP address.

Mediator rejects requests that do not include the IP address of an Integration Server user.

If Mediator cannot identify the client, Mediator fails the request and generates a Policy Violation event.

Input Parameters

`Identify Consumer`	*String.* The list of consumers against which the IP address should be validated for identifying requests from a particular client.	
	Value	**Description**
	`Registered Consumers`	Mediator will try to verify the client's credentials against the list of consumer applications who are registered as consumers for the specified API. Mediator will evaluate whether the request header contains the X-Forwarded-For, which is used for identifying the IP address of a client through an HTTP proxy.
	`Global Consumers`	*Default.* Mediator will try to verify the client's credentials against a list of all global consumers available in the Mediator.
	`Do Not Identify`	Mediator forwards the request to the native API, without attempting to verify client's credentials in incoming request.

Evaluate OAuth2 Token

This action extracts the specified OAuth access token from an incoming request and locates the client defined by that access token. For example, when you have configured this action for an API, the PEP extracts the OAuth access token from the request's HTTP header at run time and searches its list of consumers for the client that is defined by that access token.

Mediator will evaluate the incoming request to identify and validate that the client's access token.

Mediator rejects requests that do not include the OAuth access token of an Integration Server user.

Mediator supports OAuth2.0 using the grant type “Client Credentials”.

If Mediator cannot identify the client, Mediator fails the request and generates a Policy Violation event.

Input Parameters

`Identify User`	*String.* The list of consumers against which the OAuth token should be validated for identifying requests from a particular client.	
	Value	**Description**
	`Registered Consumers`	Mediator will try to verify the client's OAuth access token against the list of consumer applications who are registered as consumers for the specified API.
	`Global Consumers`	*Default.* Mediator will try to verify the client's OAuth access token against a list of all global consumers available in the Mediator.
`Validate Access Token`	*Boolean. Optional.* This option uses your resource server to verify clients. When Integration Server acts as a resource server, it receives requests from clients that include an access token. The resource server asks the authorization server to validate the access token and user. If the token is valid and the user has privileges to access the folders and services, the resource server executes the request. For more information about using Integration Server to act as a resource server, see the chapter “Configuring OAuth” in the *webMethods Integration Server Administrator’s Guide*.	
	Value	**Description**
	`True`	Default. Mediator will verify the client's OAuth access token against the list of consumers available in the Integration Server on which Mediator is running.
	`False`	Mediator will not verify the client's OAuth access token.

Evaluate WSS Username Token

If you have a native API that requires to authenticate a client to the Integration Server using the WS-Security authentication, you can use the Evaluate WSS Username Token action to extract the client's credentials (username token and password) from the WS-Security SOAP message header, and verify the client's identity.

This action extracts the username token and password supplied in the message header of the request and locates the client defined by that username token and password. For example, when you have configured this action for an API, the PEP extracts the username token and password from the SOAP header at run time and searches its list of consumers for the client that is defined by the credentials.

To use this action, the following prerequisites must be met:

- In Integration Server, create a keystore and truststore, as described in *Securing Communications with the Server* in the *webMethods Integration Server Administrator's Guide*.
- In Integration Server, create an HTTPS port, as described in *Configuring Ports* in the *webMethods Integration Server Administrator's Guide*.
- Configure Mediator by setting the HTTPS Ports Configuration parameter, as described in *Configuring Mediator > Ports Configuration* in the document *Administering webMethods Mediator*.

Mediator rejects requests that do not include the username token and password of an Integration Server user. Mediator only supports clear text passwords with this kind of authentication.

In the case where a client sends a request with transport credentials (HTTP Basic Authentication) and message credentials (WSS Username Token or WSS X.509 Certificate), the message credentials take precedence over the transport credentials when Integration Server determines which credentials it should use for the session. For more information, see **Evaluate HTTP Basic Authentication Action** and **Evaluate WSS X.509 Certificate Token**.

If Mediator cannot identify the client, Mediator fails the request and generates a Policy Violation event.

Input Parameters

`Identify Consumer`	*String.* The list of consumers against which the username token and password should be validated for identifying requests from a particular client.	
	Value	**Description**
	`Registered Consumers`	Mediator will try to verify the client's WSS username token against the list of consumer applications who are registered as consumers for the specified API.
	`Global Consumers`	*Default.* Mediator will try to verify the client's WSS username token against a list of all global consumers available in the Mediator.
	`Do Not Identify`	Mediator forwards the request to the native API, without attempting to verify the client's username token in incoming request.

Evaluate WSS X.509 Certificate

If you have a native API that requires to authenticate a client to the Integration Server using the WS-Security authentication, you can use the Evaluate WSS X.509 Certificate action to extract the client identity certificate from the WS-Security SOAP message header, and verify the client's identity.

This action extracts the certificate supplied in the header of an incoming SOAP request and locates the client defined by the information in that certificate. For example, when you have configured this action for an API, the PEP extracts the certificate from the SOAP header at run time and searches its list of consumers for the client that is defined by the certificate.

To use this action, the following prerequisites must be met:

- In Integration Server, create a keystore and truststore, as described in *Securing Communications with the Server* in the *webMethods Integration Server Administrator's Guide*.
- In Integration Server, create an HTTPS port, as described in *Configuring Ports* in the *webMethods Integration Server Administrator's Guide*.
- Configure Mediator by setting the HTTPS Ports Configuration parameter, as described in *Configuring Mediator > Ports Configuration* in the document *Administering webMethods Mediator*.

Mediator rejects requests that do not include the X.509 token of an Integration Server user.

In the case where a client sends a request with transport credentials (HTTP Basic Authentication) and message credentials (WSS Username Token or WSS X.509 Certificate), the message credentials take precedence over the transport credentials when Integration Server determines which credentials it should use for the session. For more information, see **Evaluate WSS Username Token** and **Evaluate HTTP Basic Authentication Action**.

If Mediator cannot identify the client, Mediator fails the request and generates a Policy Violation event.

Input Parameters

`Identify Consumer`	*String.* The list of consumers against which the X.509 certificate should be validated for identifying requests from a particular client.	
	Value	**Description**
	`Registered Consumers`	Mediator will try to verify the client's X.509 certificate against the list of consumer applications who are registered as consumers for the specified API.
	`Global Consumers`	*Default.* Mediator will try to verify the client's X.509 certificate against a list of all global consumers available in the Mediator.
	`Do Not Identify`	Mediator forwards the request to the native API, without attempting to verify client's certificate in incoming request.

Evaluate XPath Expression

Note: This action does not support JSON-based REST APIs.

If you have a native API that requires to authenticate a client to the Integration Server using the custom authentication, you can use the Evaluate XPath Expression action to extract the custom authentication credentials (tokens, or username and password token combination) from the request header, and verify the client's identity.

This action extracts the custom authentication credentials that is supplied in the request header (which is represented using an XPath expression) and locates the client defined by the credentials.

The custom authentication credentials can be in the form of tokens, or a username and password token combination. For example, when you have configured this action for an API, the PEP extracts the custom credentials from the request header (using an XPath expression) at run time and searches its list of consumers for the client defined by the credentials.

Mediator rejects requests that do not include the XPath expression of an Integration Server user.

If Mediator cannot identify the client, Mediator fails the request and generates a Policy Violation event.

Input Parameters

`Identify Consumer`	*String.* The list of consumers against which the XPath expression should be validated for identifying requests from a particular client.	
	Value	**Description**
	`Registered Consumers`	Mediator will try to verify the client's XPath expression against the list of consumer applications who are registered as consumers for the specified API.
	`Global Consumers`	*Default.* Mediator will try to verify the client's XPath expression against a list of all global consumers available in the Mediator.
	`Do Not Identify`	Mediator forwards the request to the native API, without attempting to verify client's XPath expression in incoming request.
`Namespace`	*String. Optional.* The namespace of the XPath expression to be validated.	
`XPath Expression`	*String. Mandatory.* An argument to evaluate the XPath expression contained in the request. See the sample below.	

Let's take a look at an example. For the following SOAP message:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
</soap:Header>
  <soap:Body>
    <catalog xmlns="http://www.store.com">
      <name>My Book</name>
      <author>ABC</author>
      <price>100</price>
    </catalog>
  </soap:Body>
</soap:Envelope>
```

The XPath expression appears as follows:

```
/soap:Envelope/soap:Body
```

HTTP Basic Authentication

This action uses the HTTP authentication mechanism to validate incoming requests from clients. Mediator authorizes the basic credentials (username and password) against a list of all global consumers available in the Mediator.

If the username/password value in the Authorization header cannot be authenticated as a valid Integration Server user (or if the Authorization header is not present in the request), a 500 SOAP fault is returned, and the client is presented with a security challenge. If the client successfully responds to the challenge, the user is authenticated. If the client does not successfully respond to the challenge, a 401 "WWW-Authenticate: Basic" response is returned and the invocation is not routed to the policy engine. As a result, no events are recorded for that invocation, and its key performance indicator (KPI) data are not included in the performance metrics.

If none of the authentication actions (**HTTP Basic Authentication**, **NTLM Authentication** or **OAuth2 Authentication**) is configured for a proxy API, Mediator forwards the request to the native API, without attempting to authenticate the request.

Input Parameters

<table>
<tr><td rowspan="6">Authenticate Using</td><td colspan="3">String. The user credentials for authenticating client requests to the native API.</td></tr>
<tr><td>Value</td><td colspan="2">Description</td></tr>
<tr><td>Existing ↩ Credentials</td><td colspan="2">Default. Mediator authenticates requests based on the credentials specified in the HTTP header. It passes the “Authorization” header present in the original client request to the native API.</td></tr>
<tr><td rowspan="4">Custom ↩ Credentials</td><td colspan="2">Mediator authenticates requests according to the values you specify in the User, Password and Domain fields.</td></tr>
<tr><td>Field</td><td>Description</td></tr>
<tr><td>Username</td><td>String. Mandatory. Account name of a consumer who is available in the Integration Server on which Mediator is running.</td></tr>
<tr><td>Password</td><td>String. Mandatory.A valid password of the consumer.</td></tr>
<tr><td></td><td>Domain</td><td>String. Optional. Domain used by the server to authenticate the consumer.</td></tr>
</table>

Invoke webMethods Integration Server

Specifically, you would need to configure the *Invoke webMethods Integration Server* action to:

- Pre-process the request messages into the format required by the native API, before Mediator sends the requests to the native APIs.
- Pre-process the native API's response messages into the format required by the clients, before Mediator returns the responses to the clients.

In some cases an API might need to process messages.

For example, you might need to accommodate differences between the message content that a client is capable of submitting and the message content that a native API expects. For example, if the client submits an order record using a slightly different structure than the structure expected by the native API, you can use this action to process the record submitted by the client to the structure required by the native API.

In the Request Handling sequence, this action invokes the webMethods IS service to pre-process the request received from the client and before it is submitted to the native API.

In the Response Processing sequence, this action invokes the webMethods IS service to process the response message received from the native API and before it is returned to the client.

Note: A webMethods IS service must be running on the same Integration Server as webMethods Mediator. It can call out a C++ or Java or .NET function. It can also call other Integration Server services to manipulate the message.

Input Parameters

`webMethods IS Service`	*String. Mandatory.* Enter a name for the webMethods IS Service. This service will be used to manipulate the request/response (the axis2 MessageContext instance). Mediator will pass to the invoked IS service the request message context (the axis2 MessageContext instance), which contains the request-specific information. Also, you can use the public IS services that accept MessageContext as input to manipulate the response contents.

JMS Routing Rule

This action allows you to specify a JMS queue to which the Mediator is to submit the request, and the destination to which the native API is to return the response.

To use the JMS Routing Rule action, you publish multiple APIs for a single native API. For example, to make a particular native API available to clients over both HTTP and JMS, you would create two APIs for the native API: one that accepts requests over HTTP and another that accepts requests over JMS. Both APIs would route requests to the same native API on the back end.

Note: To make it easier to manage APIs, consider adopting a naming convention like the one shown above. Doing so will make it easier to identify APIs and the native API with which they are associated. Keep in mind however, that unlike native APIs, the names of APIs cannot contain spaces or special characters (except _ and -). Consequently, if you adopt a convention that involves using the name of the native API as part of the API name, then the names of the native APIs themselves must not contain characters that are invalid in API names.

To use this action the following prerequisites must be met:

- Create an alias to a JNDI Provider (in the Integration Server Administrator, go to **Settings > Web Services**). For the procedure, see the section *Creating a JNDI Provider Alias* in the document *Administering webMethods Integration Server*.
- To establish an active connection between Integration Server and the JMS provider, you must configure Integration Server to use a JMS connection alias (in the Integration Server Administrator, go to **Settings > Messaging > JMS Settings**). For the procedure, see the section *Creating a JMS Connection Alias* in the document *Administering webMethods Integration Server*.
- Create a WS (Web Service) endpoint alias for provider Web Service Descriptor (WSD) that uses a JMS binder. In the Integration Server Administrator, navigate to **Settings > Web Services** and complete the Alias Name, Description, Descriptor Type, and Transport Type fields, as described in the section *Creating an Endpoint Alias for a Provider Web Service Descriptor for Use with JMS* in the document *Administering webMethods Integration Server*.
- Configure a WS (Web Service) endpoint trigger (in the Integration Server Administrator, go to **Settings > Messaging > JMS Trigger Management**). For the procedure, see the section *Editing WS Endpoint Triggers* in the document *Administering webMethods Integration Server*.
- Create a WS (Web Service) endpoint alias for consumer Web Service Descriptor (WSD) that has a JMS binder. In the Integration Server Administrator, navigate to **Settings > Web Services** and complete the Alias Name, Description, Descriptor Type, and Transport Type fields, as described in the section *Creating an Endpoint Alias for a Consumer Web Service Descriptor for Use with JMS* in the document *Administering webMethods Integration Server*.
- Additionally, in the proxy API's **Message Flow** area, make sure that you delete the predefined *Straight Through Routing* and *HTTP Basic Authentication* actions from the **Receive** stage. This is because, these actions are mutually exclusive.

Input Parameters

`Connection URL`	*String. Mandatory.* Specify a connection alias for connecting to the JMS provider (e.g., an Integration Server alias or a JNDI URL). For example, a JNDI URL of the form:
	
	`jms:queue:dynamicQueues/MyRequestQueue?`
	`wm-wsendpointalias=MediatorConsumer`
	`&targetService=vs-jms-in-echo`
	
	Note that the `wm-wsendpointalias` parameter is required for Integration Server/Mediator to look up the JMS consumer alias to send the request to the specified queue (e.g., `MyRequestQueue`), which is a dynamic queue in ActiveMQ. Also, the `targetService` parameter is required if sending to another API that uses JMS as the entry protocol.

`Reply to Destination`	*Optional.* Specify a queue name where a reply to a message should be sent.
`Priority`	Enter an integer that represents the priority of this JMS message with respect to other messages that are in the same queue. The priority value determines the order in which the messages are routed. The lower the **Priority** value, the higher the priority (i.e., 0 is the highest priority, and messages with this priority value are executed first). ■ Priority values 0 through 10. ■ The default priority for a JMS message is 0. For more information about priorities, see ***What Happens if a Queue Includes Multiple JMS Messages?***
`Time to Live`	*Optional.* A numeric value (in milliseconds) that specifies the expiration time of the JMS message. If the time-to-live is specified as zero, expiration is set to zero which indicates the message does not expire. The default value is 0.
`Delivery Mode`	*Optional.* The type of message delivery to the endpoint.

Value	Description
`Persistent`	The message is stored by the JMS server before delivering it to the client.
`Non-Persistent`	*Default.* The message is not stored before delivery.

What Happens if a Queue Includes Multiple JMS Messages?

To determine the order in which to execute the JMS messages in a queue, Mediator examines each message's Priority setting.

The Priority setting contains a non-negative integer that indicates the JMS message's priority. A priority value of 0 represents the highest possible priority.

Note: A JMS message's **Priority** property is used *only* when there are multiple JMS messages to route in the queue. If the queue has only one message to route, the **Priority** property is ignored entirely.

When a queue includes multiple JMS messages, Mediator routes the messages serially, in priority order from lowest to highest (that is, it routes with message the *lowest* priority value first). Each messages in the queue is routed to completion before the next one begins.

If two or more messages have the same priority value, their order is indeterminate. Mediator will route these messages in serial fashion after all lower priority messages and before any higher priority messages. However, you cannot predict their order

Example

If Mediator were given the following JMS messages to route for an API:

JMS Message	Priority
JMS Message A	11
JMS Message B	25
JMS Message C	11
JMS Message D	0

It would route the messages in the following order:

JMS Message	Priority
JMS Message D	0
JMS Message A then JMS Message C (or vice versa) The order of these two messages cannot be controlled or predicted because they have the same priority.	11
JMS Message B	25

Load Balancing and Failover Routing

If you have a native API that is hosted at two or more endpoints, you can use the Load Balancing and Failover Routing to distribute requests among the endpoints.

Requests are distributed across multiple endpoints. The requests are intelligently routed based on the "round-robin" execution strategy. The load for a service is balanced by directing requests to two or more services in a pool, until the optimum level is achieved. The application routes requests to services in the pool sequentially, starting from the first to the last service without considering the individual performance of the services. After the requests have been forwarded to all the services in the pool, the first service is chosen for the next loop of forwarding.

Load-balanced endpoints also have automatic Failover capability. If a load-balanced endpoint is unavailable (for example, if a connection is refused), then that endpoint is marked as "down" for

the number of seconds you specify in the `Timeout` field (during which the endpoint will not be used for sending the request), and the next configured endpoint is tried. If all the configured load-balanced endpoints are down, then a failure is sent back to the client. After the timeout expires, each endpoint marked will be available again to send the request.

Input Parameters

<table>
<tr><td><code>Route To</code></td><td colspan="2"><i>URI. Mandatory.</i> Enter the URLs of two or more endpoints in a pool to which the requests will be routed. The application routes the requests to endpoints in the pool sequentially, starting from the first to the last endpoint without considering the individual performance of the endpoints. After the requests have been forwarded to all the endpoints in the pool, the first endpoint is chosen for the next loop of forwarding.

Enter the URL of the endpoint to route the request to. For example:

<code>http://mycontainer/creditCheckService</code>

To specify additional endpoints, use the plus button next to the field to add rows.

Click the Configure Endpoint Properties icon (next to the field) if you want to configure a set of properties for the endpoints individually.

Alternatively, Mediator offers “Local Optimization” capability if the native API is hosted on the same Integration Server as webMethods Mediator. With local optimization, API invocation happens in-memory and not through a network hop.

<code>local://<<i>Service-full-path</i>></code>

OR

<code>local://<<i>server</i>>:<<i>port</i>>/ws/<<i>Service-full-path</i>></code>

For example:

<code>local://MyAPIFolder:MyLocalAPI</code>

which points to the endpoint API <code>MyLocalAPI</code> which is present under the folder <code>MyAPIFolder</code> in Integration Server.</td></tr>
<tr><td><code>Configure Endpoint Properties</code> icon</td><td colspan="2">Optional. This icon displays the Endpoint Properties dialog box that enables you to configure a set of properties for the Mediator to route incoming requests to the native API as follows:</td></tr>
<tr><td></td><td><code>SOAP Optimization Method</code></td><td>Only for SOAP-based APIs. Mediator can use the following optimization methods to parse SOAP requests to the native API:

Value | Description</td></tr>
</table>

		`MTOM`	Mediator will use the Message Transmission Optimization Mechanism (MTOM) to parse SOAP requests to the API.
		`SwA`	Mediator will use the SOAP with Attachment (SwA) technique to parse SOAP requests to the API.
		`None`	*Default*. Mediator will not use any optimization method to parse the SOAP requests to the API.
		Note: 1. Bridging between SwA and MTOM is not supported. If a client sends a SwA request, Mediator can only forward SwA to the native API. The same is true for MTOM, and applies to responses received from the native API. That is, a SwA or MTOM response received by Mediator from a native API will be forwarded to the client using the same format it received. 2. When sending SOAP requests that do not contain a MTOM or SWA attachment to a native API that returns an MTOM or SWA response, the request 'Accept' header must be set to 'multipart/related'. This is necessary so Mediator knows how to parse the response properly.	
	`HTTP Connection Timeout`	*Number. Optional*. The time interval (in seconds) after which a connection attempt will timeout. If a value 0 is specified (or if the value is not specified), Mediator will use the value specified in the `Connection Timeout` field (go to **Integration Server Administrator > Settings > Extended**). *Default*: 30 seconds.	
	`Read Timeout`	*Number. Optional.* The time interval (in seconds) after which a socket read attempt will timeout. If a value 0 is specified (or if the value is not specified), Mediator will use the value specified in the `Read Timeout` field (Open the Integration Server Administrator. Go to **> Settings > Extended**.). *Default*: 30 seconds.	
	`SSL Configuration`	*Optional.* To enable SSL client authentication that Mediator will use to authenticate incoming requests for the native API, you must specify values for both the Client Certificate Alias field and the IS Keystore Alias field. If you specify a value for only one of these fields, a deployment error will occur. **Note:** SSL client authentication is optional; you may leave both fields blank. Prerequisite: You must set up the key alias and keystore properties in the Integration Server. For the procedure, see the section *Securing Communications with the Server* in the document*webMethods Integration Server Administrator's Guide*.	

		You will use these properties to specify the following fields:	
		Value	**Description**
		`Client Certificate Alias`	*Mandatory*. The client's private key to be used for performing SSL client authentication.
		`IS Keystore Alias`	*Mandatory*. The keystore alias of the instance of Integration Server on which Mediator is running. This value (along with the value of Client Certificate Alias) will be used for performing SSL client authentication.
	`WS Security Header`	*Only for SOAP-based APIs.* Indicates whether Mediator should pass the WS-Security headers of the incoming requests to the native API.	
		Value	**Description**
		`Remove processed ↩ security headers`	*Default*. Removes the security header if it is processed by Mediator (i.e., if Mediator processes the header according to the API's security run-time policy). Note that Mediator will not remove the security header if both of the following conditions are true: 1) Mediator did not process the security header, and 2) the mustUnderstand attribute of the security header is 0/false).
		`Pass all security headers`	Passes the security header, even if it is processed by Mediator (i.e., even if Mediator processes the header according to the API's security action).

Log Invocation

This action logs request/response payloads. You can specify the log destination and the logging frequency. This action also logs other information about the requests/responses, such as the API name, operation name, the Integration Server user, a timestamp, and the response time.

Input Parameters

`Payloads`	*String.* Specify whether to log all request/response payloads.	
	Value	**Description**

	`Request`	Logs all request payloads.
	`Response`	Logs all response payloads.
`Log Generation Frequency`	*String*. Specify how frequently to log the payload.	
	Value	**Description**
	`Always`	Logs all requests and/or responses.
	`On Success`	Logs only the successful responses and/or requests.
	`On Failure`	Logs only the failed requests and/or responses.
`Send Data To`	*String*. Specify where to log the payload. **Important:** Ensure that Mediator is configured to log the payloads to the destination(s) you specify here. For details, see the section *Alerts and Transaction Logging* in the document *Administering webMethods Mediator*.	
	Value	**Description**
	`CentraSite`	Logs the payloads in the API's Events profile in CentraSite. Prerequisite: You must configure Mediator to communicate with CentraSite (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > CentraSite Communication**). For the procedure, see the section *Configuring Communication with CentraSite* in the document *Administering webMethods Mediator*.
	`Local Log`	Logs the payloads in the server log of the Integration Server on which Mediator is running. Also choose a value in the `Log Level` field: ■ Info: logs the error-level, warning-level, and informational-level alerts. ■ Warn: logs the error-level and warning-level alerts. ■ Error: logs only error-level alerts. **Important:** The Integration Server Administrator's logging level for Mediator should match the logging level specified for this action (go to **Settings > Logging > Server Logger**).
	`SNMP`	Logs the payloads in CentraSite's SNMP server or a third-party SNMP server. Prerequisite: You must configure the SNMP server destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > SNMP**). For the procedure, see the section *SNMP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.

	Email	Sends the payloads to an SMTP email server, which sends them to the email address(es) you specify here. Mediator sends the payloads as email attachments that are compressed using gzip data compression. To specify multiple addresses, use the  button to add rows. Prerequisite: You must configure the SMTP server destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > Email**). For the procedure, see the section *SMTP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.
	Audit Log	Logs the payload to the Integration Server audit logger. For information, see the *webMethods Audit Logging Guide*. **Note:** If you expect a high volume of events in your system, it is recommended that you select the Audit Log destination for this action.
	EDA	Mediator can use EDA to log the payloads to a database. Prerequisite: You must configure the EDA destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > EDA**). For the procedure, see the section *EDA Configuration for Publishing Run-Time Events and Metrics* in the document *Administering webMethods Mediator*.

Monitor Service Level Agreement

This action monitors a set of run-time performance conditions for an API, and sends alerts to a specified destination when the performance conditions are violated. This action enables you to monitor run-time performance for *one or more specified clients*.

You can configure this action to define a *Service Level Agreement (SLA)*, which is a set of conditions that defines the level of performance that a client should expect from an API. You can use this action to identify whether an API threshold rules are met or exceeded. For example, you might define an agreement with a particular client that sends an alert to the client (consumer application) if responses are not sent within a certain maximum response time. You can configure SLAs for each API/consumer application combination.

For the counter-based metrics (Total Request Count, Success Count, Fault Count), Mediator sends an alert as soon as the performance condition is violated, without having to wait until the end of the metrics tracking interval. You can choose whether to send an alert only once during the interval, or every time the violation occurs during the interval. (Mediator will send another alert the next time a condition is violated during a subsequent interval.) For information about the metrics tracking interval, see ***The Metrics Tracking Interval***.

For the aggregated metrics (Average Response Time, Minimum Response Time, Maximum Response Time), Mediator aggregates the response times at the end of the interval, and then sends an alert if the performance condition is violated.

This action does not include metrics for failed invocations.

Note: To enable Mediator to publish performance metrics, you must configure Mediator to communicate with CentraSite (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > CentraSite Communication**). For the procedure, see the section *Configuring Communication with CentraSite* in the document *Administering webMethods Mediator*.

Input Parameters

<table>
<tr><td>Action Configuration Parameters</td><td colspan="2">Specify one or more conditions to monitor. To do this, specify a metric, operator, and value for each metric. To specify multiple conditions, use the  button to add multiple rows. If multiple parameters are used, they are connected by the AND operator.</td></tr>
<tr><td rowspan="10">Name</td><td colspan="2">String. Array. The metrics to monitor.</td></tr>
<tr><td>Value</td><td>Description</td></tr>
<tr><td>Availability</td><td>Indicates whether the API was available to the specified clients in the current interval.</td></tr>
<tr><td>Average Response Time</td><td>The average amount of time it took the service to complete all invocations in the current interval. Response time is measured from the moment Mediator receives the request until the moment it returns the response to the caller.</td></tr>
<tr><td>Fault Count</td><td>Indicates the number of faults returned in the current interval.</td></tr>
<tr><td>Maximum Response Time</td><td>The maximum amount of time to respond to a request in the current interval.</td></tr>
<tr><td>Minimum Response Time</td><td>The minimum amount of time to respond to a request in the current interval.</td></tr>
<tr><td>Successful Request Count</td><td>The number of successful requests in the current interval.</td></tr>
<tr><td>Total Request Count</td><td>The total number of requests (successful and unsuccessful) in the current interval.</td></tr>
<tr><td>Operator</td><td>String. Array. Specifies an operator.</td></tr>
<tr><td>Value</td><td colspan="2">Integer. Array. Specifies an alert value.</td></tr>
</table>

`Alert for Consumer Applications`	*Object. Array.* Specify the Application asset(s) to which this Service Level Agreement will apply. To specify multiple consumer applications, use the  button to add multiple rows.	
`Alert Configuration Parameters`	Specifies the parameters for the alerts that will report on the Service Level Agreement conditions:	
`Alert Interval`	*Number.* The time period (in minutes) in which to monitor performance before sending an alert if a condition is violated. For information about the metrics tracking interval, see ***The Metrics Tracking Interval***.	
`Alert Frequency`	*String.* Specifies how frequently to issue alerts for the counter-based metrics (Total Request Count, Success Count, Fault Count).	

Value	Description
`Every Time`	Issue an alert every time one of the specified conditions is violated.
`Only Once`	Issue an alert only the first time one of the specified conditions is violated.

`Alert Destination`	*String.* Specifies where to log the alert. **Important:** Ensure that Mediator is configured to send event notifications to the destination(s) you specify here. For details, see *Alerts and Transaction Logging* in the document *Administering webMethods Mediator*.	

Value	Description
`CentraSite`	Sends the alerts to the API's Events profile in CentraSite. Prerequisite: You must configure Mediator to communicate with CentraSite (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > CentraSite Communication**). For the procedure, see the section *Configuring Communication with CentraSite* in the document *Administering webMethods Mediator*.

	Local Log	Sends the alerts to the server log of the Integration Server on which Mediator is running. Also choose a value in the Log Level field: ■ Info: Logs error-level, warning-level, and informational-level alerts. ■ Warn: Logs error-level and warning-level alerts. ■ Error: Logs only error-level alerts. **Important:** The Integration Server Administrator's logging level for Mediator should match the logging level specified for this action (go to **Settings > Logging > Server Logger**).
	SNMP	Sends the alerts to CentraSite's SNMP server or a third-party SNMP server. Prerequisite: You must configure the SNMP server destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > Email**). For the procedure, see the section *SNMP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.
	Email	Sends the alerts to an SMTP email server, which sends them to the email address(es) you specify here. To specify multiple addresses, use the  button to add rows. Prerequisite: You must configure the SMTP server destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > Email**). For the procedure, see the section *SMTP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.
	EDA	Mediator can use EDA to log the payloads to a database. Prerequisite: You must configure the EDA destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > EDA**). For the procedure, see the section *EDA Configuration for Publishing Run-Time Events and Metrics* in the document *Administering webMethods Mediator*.
Alert Message	*String. Optional.* Specify a text message to include in the alert.	

Monitor Service Performance

This action is similar to the **Monitor Service Level Agreement** action. Both actions can monitor the same set of run-time performance conditions for an API, and then send alerts when the performance conditions are violated. However, this action monitors run-time performance for *a specific client*.

For the counter-based metrics (Total Request Count, Success Count, Fault Count), Mediator sends an alert as soon as the performance condition is violated, without having to wait until the end of the metrics tracking interval. You can choose whether to send an alert only once during the interval, or every time the violation occurs during the interval. (Mediator will send another alert the next time a condition is violated during a subsequent interval.) For information about the metrics tracking interval, see ***The Metrics Tracking Interval***.

For the aggregated metrics (Average Response Time, Minimum Response Time, Maximum Response Time), Mediator aggregates the response times at the end of the interval, and then sends an alert if the performance condition is violated.

This action does not include metrics for failed invocations.

Note: To enable Mediator to publish performance metrics, you must configure Mediator to communicate with CentraSite (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > CentraSite Communication**). For the procedure, see the section *Configuring Communication with CentraSite* in the document *Administering webMethods Mediator*.

Input Parameters

`Action Configuration Parameters`	Specify one or more conditions to monitor. To do this, specify a metric, operator, and value for each metric. To specify multiple conditions, use the + button to add multiple rows. If multiple parameters are used, they are connected by the AND operator.	
`Name`	*String. Array.* The metrics to monitor.	
	Value	**Description**
	`Availability`	Indicates whether the service was available to the specified clients in the current interval.
	`Average Response Time`	The average amount of time it took the service to complete all invocations in the current interval. Response time is measured from the moment Mediator receives the request until the moment it returns the response to the caller.

	`Fault Count`	Indicates the number of faults returned in the current interval.
	`Maximum Response Time`	The maximum amount of time to respond to a request in the current interval.
	`Minimum Response Time`	The minimum amount of time to respond to a request in the current interval.
	`Successful Request Count`	The number of successful requests in the current interval.
	`Total Request Count`	The total number of requests (successful and unsuccessful) in the current interval.
	`Operator`	*String. Array.* Specify an operator.
`Value`	*Integer. Array.* Specify an alert value.	
`Alert Configuration Parameters`	Specify the parameters for the alerts that will report on the Service Level Agreement conditions:	
`Alert Interval`	*Number.* The time period (in minutes) in which to monitor performance before sending an alert if a condition is violated. For information about the metrics tracking interval, see ***The Metrics Tracking Interval***.	
`Alert Frequency`	*String.* Specify how frequently to issue alerts for the counter-based metrics (Total Request Count, Success Count, Fault Count).	
	Value	**Description**
	`Every Time`	Issue an alert every time one of the specified conditions is violated.
	`Only Once`	Issue an alert only the first time one of the specified conditions is violated.
`Alert Destination`	*String.* Specify where to log the alert. **Important:** Ensure that Mediator is configured to send event notifications to the destination(s) you specify here. For details, see *Alerts and Transaction Logging* in the document *Administering webMethods Mediator*.	
	Value	**Description**
	`CentraSite`	Sends the alerts to the API's Events profile in CentraSite. Prerequisite: You must configure Mediator to communicate with CentraSite (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > CentraSite Communication**). For the procedure, see the section *Configuring Communication with CentraSite* in the document *Administering webMethods Mediator*.

	`Local Log`	Sends the alerts to the server log of the Integration Server on which Mediator is running. Also choose a value in the `Log Level` field: ■ Info: Logs error-level, warning-level, and informational-level alerts. ■ Warn: Logs error-level and warning-level alerts. ■ Error: Logs only error-level alerts. **Important:** The Integration Server Administrator's logging level for Mediator should match the logging level specified for this action (go to **Settings > Logging > Server Logger**).
	`SNMP`	Sends the alerts to CentraSite's SNMP server or a third-party SNMP server. Prerequisite: You must configure the SNMP server destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > Email**). For the procedure, see the section *SNMP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.
	`Email`	Sends the alerts to an SMTP email server, which sends them to the email address(es) you specify here. To specify multiple addresses, use the  button to add rows. Prerequisite: You must configure the SMTP server destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > Email**). For the procedure, see the section *SMTP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.
	`EDA`	Mediator can use EDA to log the payloads to a database. Prerequisite: You must configure the EDA destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > EDA**). For the procedure, see the section *EDA Configuration for Publishing Run-Time Events and Metrics* in the document *Administering webMethods Mediator*.
`Alert Message`	*String. Optional.* Specify a text message to include in the alert.	

NTLM Authentication

This action uses the NTLM authentication to validate incoming requests from clients. Mediator authorizes the NTLM credentials (username and password) against a list of all global consumers available in the Mediator.

If the username/password value in the Authorization header cannot be authenticated as a valid Integration Server user (or if the Authorization header is not present in the request), a 500 SOAP fault is returned, and the client is presented with a security challenge. If the client successfully responds to the challenge, the user is authenticated. If the client does not successfully respond to the challenge, a 401 Unauthorized "WWW-Authenticate: NTLM" (for NTLM authentication) or "WWW-Authenticate: Negotiate" (for Kerberos authentication) is returned in the response header and the invocation is not routed to the policy engine. As a result, no events are recorded for that invocation, and its key performance indicator (KPI) data are not included in the performance metrics.

Note: Note that if Mediator is used to access a native API protected by NTLM (which is typically hosted in IIS), then the native API in IIS should be configured to use NTLM as the authentication scheme. If the authentication scheme is configured as “Windows”, then “NTLM” should be in its list.

If none of the authentication actions (**HTTP Basic Authentication**, **NTLM Authentication** or **OAuth2 Authentication**) is configured for a proxy API, Mediator forwards the request to the native API, without attempting to authenticate the request.

Input Parameters

<table>
<tr><td><code>Authenticate Using</code></td><td>String. Specifies the user credentials for authenticating client requests to the native API.

Note: Currently Windows only: If Mediator is used to access a native API protected by NTLM (which is typically hosted in IIS), then the native API in IIS should be configured to use NTLM as the authentication scheme. If the authentication scheme is configured as "Windows", then "NTLM" should be in its list. The "Negotiate" handshake will be supported in the near future. This note applies to all three of the following options for NTLM.
<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td><code>Existing ↩ Credentials</code></td><td>Default. Mediator uses the user credentials passed in the request header for an NTLM handshake with the server.</td></tr>
<tr><td></td><td>Mediator uses the values you specify in the User, Password and Domain fields for an NTLM handshake with the server.
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Username</code></td><td>String. Mandatory. Account name of a consumer who is available in the Integration Server on which Mediator is running.</td></tr>
</table></td></tr>
</table></td></tr>
</table>

	`Custom ↩ Credentials`	`Password`	*String. Mandatory.*A valid password of the consumer.
		`Domain`	*String. Optional.* Domain used by the server to authenticate the consumer.
	`Transparent`	Mediator will behave in "pass by" mode, allowing an NTLM handshake to occur between the client and server. **Note:** 1. If the client is a WCF application, then the client should be configured with clientCredentialType set to NTLM 2. If you configure for NTLM authentication scheme in transparent mode, Mediator will behave in "pass by" mode, allowing an NTLM handshake to occur between the client and server. This kind of a NTLM handshake becomes unreliable to authenticate the incoming client request. Mediator now supports Kerberos handshake in Transparent mode. If you choose to use the NTLM Transparent mode with Kerberos authentication, set the value of `watt.pg.disableNtlmAuthHandler` property to "true" in the extended settings for the Integration Server. For more information about configuring the extended settings, see the *Working with Extended Configuration Settings* section in the document *webMethods Integration Server Administrator's Guide*.	

OAuth2 Authentication

This action uses the OAuth 2.0 authentication to validate incoming requests from clients. Mediator authorizes the OAuth 2.0 credentials (access token) against a list of all global consumers available in the Mediator.

This action uses the NTLM authentication to validate incoming requests from clients. Mediator authorizes the credentials against a list of all global consumers available in the Mediator.

If the access token value in the Authorization header cannot be authenticated as a valid Integration Server user (or if the Authorization header is not present in the request), a 500 SOAP fault is returned, and the client is presented with a security challenge. If the client successfully responds to the challenge, the user is authenticated. If the client does not successfully respond to the challenge, a "WWW-Authenticate: OAuth" response is returned and the invocation is not routed to the policy engine. As a result, no events are recorded for that invocation, and its key performance indicator (KPI) data are not included in the performance metrics.

If none of the authentication actions (**HTTP Basic Authentication**, **NTLM Authentication** or **OAuth2 Authentication**) is configured for a proxy API, Mediator forwards the request to the native API, without attempting to authenticate the request.

Input Parameters

<table>
<tr><td>Authenticate Using</td><td colspan="3">String. Specifies the OAuth2 access token for authenticating client requests to the native API.</td></tr>
<tr><td></td><td>Value</td><td colspan="2">Description</td></tr>
<tr><td></td><td>Existing ↩ Token</td><td colspan="2">Default. Mediator uses the OAuth2 access token specified in the HTTP "Authorization" header to validate client requests for a native API.</td></tr>
<tr><td></td><td>Custom Token</td><td colspan="2">Mediator uses the access token you specify in the OAuth2 Token, field to validate client requests for a native API.</td></tr>
<tr><td></td><td></td><td>Field</td><td>Description</td></tr>
<tr><td></td><td></td><td>OAuth2 Token</td><td>String. Mandatory. Specifies an OAuth2 access token to be deployed by Mediator. The consumer need not pass the OAuth2 token during service invocation.</td></tr>
</table>

Response Transformation

The *Response Transformation* action specifies:

- The XSLT transformation file to transform response messages from native APIs into a format required by the client.

In some cases a message needs to be transformed prior to sending to the client.

For example, you might need to accommodate differences between the message content that a native API is capable of submitting and the message content that a client expects. For example, if the native API submits an order record using a slightly different structure than the structure expected by the client, you can use this action to transform the record submitted by the native API to the structure required by the client.

When this action is configured for a proxy API, the native API's response messages are transformed into the format required by the client, before Mediator returns the responses to the clients.

Input Parameters

Transformation File	*File. Mandatory.* Click **Choose**, select the XSL transformation file from your file system and click **OK**. When you virtualize an API, the transformation file is validated. If there are no validation errors, the XSLT file is displayed as a download link in the same dialog. If the transformation file is invalid (for example, non-XSLT file), this will be indicated by a warning icon. **Note:** If you make changes to the XSLT file in the future, you must republish the API.

	Important: The XSL file uploaded by the user should not contain the XML declaration in it (e.g., xml version="1.0" encoding="UTF-8"). This is because when the API is published to Mediator, Mediator embeds the XSL file in the virtual service definition (VSD), and since the VSD itself is in XML format, there cannot be an XML declaration line in the middle of it. This can lead to unexpected deployment issues which can be avoided by making sure the XSL file does not contain the declaration line.

Request Transformation

The *Request Transformation* action specifies:

- The XSLT Transformation File to transform request messages from clients into a format required by the native API.

In some cases a native API might need to transform messages.

For example, you might need to accommodate differences between the message content that a client is capable of submitting and the message content that a native API expects. For example, if the client submits an order record using a slightly different structure than the structure expected by the native API, you can use this action to transform the record submitted by the client to the structure required by the native API.

When this action is configured for a proxy API, the incoming requests from the clients are transformed into a format required by the native API, before Mediator sends the requests to the native APIs.

Input Parameters

`Transformation File`	*File. Mandatory.* Click **Choose**, select the XSL transformation file from your file system and click **OK**. When you virtualize an API, the transformation file is validated. If there are no validation errors, the XSLT file is displayed as a download link in the same dialog. If the transformation file is invalid (for example, non-XSLT file), this will be indicated by a warning icon. **Note:** If you make changes to the XSLT file in the future, you must republish the API. **Important:** The XSL file uploaded by the user should not contain the XML declaration in it (e.g., xml version="1.0" encoding="UTF-8"). This is because when the API is published to Mediator, Mediator embeds the XSL file in the virtual service definition (VSD), and since the VSD itself is in XML format, there cannot be an XML declaration line in the middle of it. This can lead to unexpected deployment issues which can be avoided by making sure the XSL file does not contain the declaration line.

Require Encryption

This action requires that a request's XML element (which is represented by an XPath expression) be encrypted.

To use this action, the following prerequisites must be met:

1. Configure Integration Server: Set up keystores and truststores in Integration Server, as described in *Securing Communications with the Server* in the document *Administering webMethods Integration Server*.
2. Configure Mediator: In the Integration Server Administrator, navigate to **Solutions > Mediator > Administration > General** and complete the IS Keystore Name, IS Truststore Name and Alias (signing) fields, as described in *Keystore Configuration* in the document *Administering WebMethods Mediator*.

When this action is configured for a proxy API, Mediator provides decryption of incoming requests and encryption of outgoing responses. Mediator can encrypt and decrypt only individual elements in the SOAP message body that are defined by the XPath expressions configured for the action. Mediator requires that requests contain the encrypted elements that match those in the XPath expression. You must encrypt the entire element, not just the data between the element tags. Mediator rejects requests if the element name is not encrypted.

Important: Do not encrypt the entire SOAP body because a SOAP request without an element will appear to Mediator to be malformed.

Mediator attempts to encrypt the response elements that match the XPath expressions with those defined for the action. If the response does not have any elements that match the XPath expression, Mediator will not encrypt the response before sending. If the XPath expression resolves a portion of the response message, but Mediator cannot locate a certificate to encrypt the response, then Mediator sends a SOAP fault exception to the client and a Policy Violation event notification to CentraSite.

How Mediator Encrypts Responses

The Require Encryption action encrypts the response back to the client by dynamically setting a public key alias at run time. Mediator determines the public key alias as follows:

1. If Mediator can access the X.509 certificate of the client (based on the incoming request signature), it will use "useReqSigCert" as the public key alias.

   OR

2. If an "Evaluate" action is present in the message flow (and it successfully identifies a client), then Mediator will look for a public key alias with that client name in the "IS Keystore Name" property. The "IS Keystore Name" property is specified in the Integration Server Administrator, under Solutions > Mediator > Administration > General. This property should be set to an Integration Server keystore that Mediator will use.

For an "Evaluate" action that allows for anonymous usage, Mediator does *not* require a client name in order to send encrypted responses. In this case, Mediator can use one of the following to encrypt the response in the following order, depending on what is present in the security element:

- A signing certificate.
- Client name.
- WSS username, SAML token or X.509 certificate.
- HTTP authorized user.

OR

3. If Mediator can determine the current IS user from the request (i.e., if an Integration Server WS-Stack determined that Subject is present), then the first principal in that subject is used.

   OR

4. If the above steps all fail, then Mediator will use either the WS-Security username token or the HTTP Basic-Auth username value. There should be a public key entry with the same name as the identified username.

Input Parameters

`Namespace`	*String. Mandatory.* Namespace of the element required to be encrypted.
	
	Note: Enter the namespace prefix in the following format: `xmlns:<prefix-name>`. For example: `xmlns:soapenv`. For more information, see the XML Namespaces specifications at **http://www.w3.org/TR/REC-xml-names/#ns-decl**.
	
	The generated XPath element in the policy should look similar to this:
	
	`<sp:SignedElements ↩`
	`xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">`
	`<sp:XPath ↩`
	`xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">//soapenv:Body</sp:XPath>`
	`</sp:SignedElements>`

`Element to be Encrypted`	*String. Mandatory.* An XPath expression that represents the XML element that is required to be encrypted. See the sample below.

Let's take a look at an example. For the following SOAP message:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
</soap:Header>
  <soap:Body>
    <catalog xmlns="http://www.store.com">
      <name>My Book</name>
      <author>ABC</author>
      <price>100</price>
    </catalog>
  </soap:Body>
</soap:Envelope>
```

The XPath expression appears as follows:

```
/soap:Envelope/soap:Body
```

Require HTTP / HTTPS

If you have a native API that requires clients to communicate with the server using the HTTP and/or HTTPS protocols, you can use the Require HTTP / HTTPS protocol action.

This action allows you to bridge the transport protocols between the client and the Mediator. For example, suppose you have a native API that is exposed over HTTPS and an API that receives requests over HTTP. In this situation, you can configure the API's Require HTTP / HTTPS action to accept HTTP requests and configure its Routing action to route the request to the native API using HTTPS.

Input Parameters

`Protocol`	*String*. Specifies the protocol over which the Mediator accepts requests from the client. **Note:** CentraSite supports HTTP version 1.1 only. **Important:** Before you deploy an API over HTTPS, ensure that the Integration Server on which the Mediator is running has been configured for SSL. In addition, make sure you specify an HTTPS port in the Mediator's Ports Configuration page. (In the Integration Server Administrator, go to **Solutions > Mediator > Administration > General** and specify the port in the **HTTPS Ports Configuration** field.) For details on the Port Configuration page, see the section *Configuring Mediator* in the document *Administering webMethods Mediator*.)	
	Value	**Description**

	`HTTP`	*Default.* Mediator will only accept requests that are sent using the HTTP protocol.
	`HTTPS`	Mediator will only accept requests that are sent using the HTTPS protocol.
	You can select *both* HTTP and HTTPS if needed.	
`SOAP Version`	*String. For SOAP-based APIs.* Specifies the SOAP version of the requests which the Mediator accepts from the client.	
	Value	**Description**
	`SOAP 1.1`	*Default.* Mediator will only accept requests that are in the SOAP 1.1 format.
	`SOAP 1.2`	Mediator will only accept requests that are in the SOAP 1.2 format.
`HTTP Methods`	*String. For REST-based APIs.* Specifies the HTTP methods in incoming requests which the Mediator accepts from the client.	
	Value	**Description**
	`GET`	Mediator will only accept GET invocations for the native API.
	`PUT`	Mediator will only accept PUT invocations for the native API.
	`POST`	Mediator will only accept POST invocations for the native API.
	`DELETE`	Mediator will only accept DELETE invocations for the native API.
	You can select *more than one* HTTP method if needed. **Note:** It is important to specify all the HTTP methods that are supported for the API. For example, if the API is deployed to Mediator and you selected only the GET method in its Straight Through Routing action, then Mediator will only permit GET invocations. In this case, POST requests will be rejected with a return of Status Code 405 even if the native API happens to support POSTs.	

Require JMS

If you have a native API that requires clients to communicate with the server using the JMS protocol, you can use the Require JMS protocol action.

This action allows you to bridge protocols between the client and the native API. For example, suppose you have a native API that is exposed over JMS and a client that submits SOAP requests over HTTP. In this situation, you can configure the API's Require JMS Protocol action to accept SOAP requests over Java Message Service (JMS) and configure its JMS Routing Rule action to route the request to the Web service using JMS.

When this action is configured for a proxy API, you can intentionally expose an API over a JMS protocol. For example, if you have a native API that is exposed over HTTP, you might expose the

API over JMS simply to gain the asynchronous-messaging and guaranteed-delivery benefits that one gains by using JMS as the message transport.

To use this action the following prerequisites must be met:

- Create an alias to a JNDI Provider (in the Integration Server Administrator, go to **Settings > Web Services**). For the procedure, see the section *Creating a JNDI Provider Alias* in the document *Administering webMethods Integration Server*.
- To establish an active connection between Integration Server and the JMS provider, you must configure Integration Server to use a JMS connection alias (in the Integration Server Administrator, go to **Settings > Messaging > JMS Settings**). For the procedure, see the section *Creating a JMS Connection Alias* in the document *Administering webMethods Integration Server*.
- Create a WS (Web Service) endpoint alias for provider Web Service Descriptor (WSD) that uses a JMS binder. In the Integration Server Administrator, navigate to **Settings > Web Services** and complete the Alias Name, Description, Descriptor Type, and Transport Type fields, as described in the section *Creating an Endpoint Alias for a Provider Web Service Descriptor for Use with JMS* in the document *Administering webMethods Integration Server*.
- Configure a WS (Web Service) endpoint trigger (in the Integration Server Administrator, go to **Settings > Messaging > JMS Trigger Management**). For the procedure, see the section *Editing WS Endpoint Triggers* in the document *Administering webMethods Integration Server*.
- Create a WS (Web Service) endpoint alias for consumer Web Service Descriptor (WSD) that has a JMS binder. In the Integration Server Administrator, navigate to **Settings > Web Services** and complete the Alias Name, Description, Descriptor Type, and Transport Type fields, as described in the section *Creating an Endpoint Alias for a Consumer Web Service Descriptor for Use with JMS* in the document *Administering webMethods Integration Server*.
- Additionally, in the API's Message Flow, make sure that you delete the predefined *Require HTTP / HTTPS Protocol* action from the Receive stage. This is because, these actions are mutually exclusive.

Input Parameters

`JMS Provider Alias`	*String. Mandatory.* Specify the name of Integration Server's JMS provider alias. The alias should include the JNDI destination name and the JMS connection factory.	
`SOAP Version`	*String.* Specify the SOAP version of the requests which the Mediator accepts from the client.	
	Value	**Description**

	`SOAP 1.1`	*Default.* Mediator will only accept requests that are in the SOAP 1.1 format.
	`SOAP 1.2`	Mediator will only accept requests that are in the SOAP 1.2 format.

Require Signing

This action requires that a request's XML element (which is represented by an XPath expression) be signed.

Prerequisites

1. Configure Integration Server: Set up keystores and truststores in Integration Server, as described in *Securing Communications with the Server* in the document *Administering webMethods Integration Server*.
2. Configure Mediator: In the Integration Server Administrator, navigate to **Solutions > Mediator > Administration > General** and complete the IS Keystore Name, IS Truststore Name and Alias (signing) fields, as described in *Keystore Configuration* in the document *Administering WebMethods Mediator*. Mediator uses the signing alias specified in the Alias (signing) field to sign the response.

When this action is configured for a proxy API, Mediator validates that the requests are properly signed, and provides signing for responses. Mediator provides support both for signing an entire SOAP message body or individual elements of the SOAP message body. Mediator uses a digital signature element in the security header to verify that all elements matching the XPath expression were signed. If the request contains elements that were not signed or no signature is present, then Mediator rejects the request.

Note: You must map the public certificate of the key used to sign the request to an Integration Server user. If the certificate is not mapped, Mediator returns a SOAP fault to the caller.

Input Parameters

`Namespace`	*String. Mandatory.* Namespace of the element required to be signed. **Note:** Enter the namespace prefix in the following format: `xmlns:<prefix-name>`. For example: `xmlns:soapenv`. For more information, see the XML Namespaces specifications at **http://www.w3.org/TR/REC-xml-names/#ns-decl**. The generated XPath element in the policy should look similar to this:

	<sp:SignedElements ↩ xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"> <sp:XPath ↩ xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">//soapenv:Body</sp:XPath </sp:SignedElements>
Element to be Signed	*String. Mandatory.* An XPath expression that represents the XML element that is required to be signed. See the sample below.

Let's take a look at an example. For the following SOAP message:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
</soap:Header>
  <soap:Body>
    <catalog xmlns="http://www.store.com">
      <name>My Book</name>
      <author>ABC</author>
      <price>100</price>
    </catalog>
  </soap:Body>
</soap:Envelope>
```

The XPath expression appears as follows:

```
/soap:Envelope/soap:Body
```

Require SSL

This action requires that requests be sent via SSL client certificates.

When this action is configured for a proxy API, Mediator ensures that requests are sent to the server using the HTTPS protocol (SSL). The action also specifies whether the client certificate is required. This allows Mediator to verify the client sending the request. If the action requires the client certificate, but it is not presented, Mediator rejects the message.

When a client certificate is required by the action, the Integration Server HTTPS port should be configured to request or require a client certificate.

Note: In Integration Server, create an HTTPS port, as described in *Configuring Ports* in the *webMethods Integration Server Administrator's Guide*.

Input Parameters

Client Certificate Required	Specifies whether client certificates are required for the purposes of: ■ Verifying the signature of signed SOAP requests or decrypting encrypted SOAP requests ■ Signing SOAP responses or encrypting SOAP responses

Require Timestamps

When this action is set for the API, Mediator requires that timestamps be included in the request header. Mediator checks the timestamp value against the current time to ensure that the request is not an old message. This serves to protect your system against attempts at message tampering, such as replay attacks.

Mediator rejects the request if either of the following happens:

- Mediator receives a timestamp that exceeds the time defined by the timestamp element.
- A timestamp element is not included in the request.

Note: This action has no input parameters; you simply drag and drop the action into the **Message Flow** area.

Input Parameters

None.

Require WSS SAML Token

When this action is configured for a proxy API, Mediator uses a WSS Security Assertion Markup Language (SAML) assertion token to validate clients for an API.

Note: For information about configuring your system for SAML token processing, see *SAML Support in Mediator* in the document *Administering webMethods Mediator*.

Input Parameters

SAML Subject Confirmation	*String* Specifies the SAML subject confirmation methods:	
	Value	**Description**

	Holder of Key	Default. Select this option if clients use the SAML V1.1 or V2.0 Holder-of-Key Web Browser SSO Profile, which allows for transport of holder-of-key assertions. In this scenario, the client presents a holder-of-key SAML assertion acquired from its preferred identity provider to access a web-based resource at an API provider. If you select Holder of Key, Mediator also implicitly selects the "timestamp" and "signing" assertions to the virtual service definition (VSD). Thus, you should not add the ""Require Timestamps" and "Require Signing" actions to the API if the "Require WSS SAML Token" action is already applied.
	Bearer	Select this option if clients use the SAML V1.1 Bearer token authentication, in which a Bearer token mechanism relies upon bearer semantics as a means by which the client conveys to Mediator the sender's identity. If you select Bearer, the "timestamp" and "signing" assertions will be added to the virtual service definition (VSD). **Note:** If clients use SAML 2.0 Sender-Vouches tokens, configure your system as described in *SAML Support in Mediator* in the document *Administering WebMethods Mediator*.
SAML Version	*String* Specifies the WSS SAML Token version to use: 1.1 or 2.0.	

Set Custom Headers

When this action is configured for a proxy API, Mediator includes custom HTTP headers to the client requests before submitting to the native APIs.

Input Parameters

Header	*String*. Mediator uses the HTTP headers that you specify in the **Name** and **Value** columns below. If you need to specify multiple headers, use the  button to add rows.	
	Value	**Description**
	Name	*String*. A name for the HTTP header field. The header field name ($field) is not case sensitive.
	Value	*String*. A value for the HTTP header field.

Sample

Let's imagine you have a Name field "Authorization". This will be encoded in Base64 scheme as follows: QXVOaG9yaXphdGlvbg==.

Set JMS Headers

Every JMS message includes message header properties that are always passed from provider to client. The purpose of the header properties is to convey extra information to the client outside the normal content of the message body.

When this action is configured for a proxy API, Mediator uses the JMS header properties to authenticate client requests before submitting to the native APIs.

To use this action the following prerequisites must be met:

- Create an alias to a JNDI Provider (in the Integration Server Administrator, go to **Settings > Web Services**). For the procedure, see the section *Creating a JNDI Provider Alias* in the document *Administering webMethods Integration Server*.
- To establish an active connection between Integration Server and the JMS provider, you must configure Integration Server to use a JMS connection alias (in the Integration Server Administrator, go to **Settings > Messaging > JMS Settings**). For the procedure, see the section *Creating a JMS Connection Alias* in the document *Administering webMethods Integration Server*.
- Create a WS (Web Service) endpoint alias for provider Web Service Descriptor (WSD) that uses a JMS binder. In the Integration Server Administrator, navigate to **Settings > Web Services** and complete the Alias Name, Description, Descriptor Type, and Transport Type fields, as described in the section *Creating an Endpoint Alias for a Provider Web Service Descriptor for Use with JMS* in the document *Administering webMethods Integration Server*.
- Configure a WS (Web Service) endpoint trigger (in the Integration Server Administrator, go to **Settings > Messaging > JMS Trigger Management**). For the procedure, see the section *Editing WS Endpoint Triggers* in the document *Administering webMethods Integration Server*.
- Create a WS (Web Service) endpoint alias for consumer Web Service Descriptor (WSD) that has a JMS binder. In the Integration Server Administrator, navigate to **Settings > Web Services** and complete the Alias Name, Description, Descriptor Type, and Transport Type fields, as described in the section *Creating an Endpoint Alias for a Consumer Web Service Descriptor for Use with JMS* in the document *Administering webMethods Integration Server*.

Input Parameters

`Header`	*String*. The JMS message headers that Mediator will use to authenticate incoming requests for the native API. To add additional rows, use the plus button.	
	Value	**Description**

	`Name`	*String.* A name for the JMS message header field. The header field name ($field) is not case sensitive.
	`Value`	*String.* A value for the JMS message header field.

Settable JMS Header Properties

Property Name	Property Type	Getter Method
Message ID	string	getJMSMessageID()
Priority	int	getJMSPriority()
Time To Live	long	getTimeToLive()
Delivery Mode	int	getJMSDeliveryMode()
Message Expiration	long	getJMSExpiration()
Correlation ID	string	getJMSCorralationID()
Redelivered	boolean	getJMSRedelivered()
Time Stamp	long	getJMSTimeStamp()
Type	string	getJMSType()

Set Message Properties

The message property fields are similar to header fields described previously in the *Set JMS Headers* action, except these fields are set exclusively by the consumer application. When a client receives a message, the properties are in read-only mode. If a client tries to modify any of the properties, a MessageNotWriteableException will be thrown.

The properties are standard Java name/value pairs. The property names must conform to the message selector syntax specifications defined in the Message interface.

Property fields are most often used for message selection and filtering. By using a property field, a message consumer can interrogate the property field and perform message filtering and selection.

When this action is configured for a proxy API, Mediator uses the message properties to authenticate client requests before submitting to the native APIs.

To use this action the following prerequisites must be met:

- Create an alias to a JNDI Provider (in the Integration Server Administrator, go to **Settings > Web Services**). For the procedure, see the section *Creating a JNDI Provider Alias* in the document *Administering webMethods Integration Server*.
- To establish an active connection between Integration Server and the JMS provider, you must configure Integration Server to use a JMS connection alias (in the Integration Server Administrator, go to **Settings > Messaging > JMS Settings**). For the procedure, see the section *Creating a JMS Connection Alias* in the document *Administering webMethods Integration Server*.

- Create a WS (Web Service) endpoint alias for provider Web Service Descriptor (WSD) that uses a JMS binder. In the Integration Server Administrator, navigate to **Settings > Web Services** and complete the Alias Name, Description, Descriptor Type, and Transport Type fields, as described in the section *Creating an Endpoint Alias for a Provider Web Service Descriptor for Use with JMS* in the document *Administering webMethods Integration Server*.
- Configure a WS (Web Service) endpoint trigger (in the Integration Server Administrator, go to **Settings > Messaging > JMS Trigger Management**). For the procedure, see the section *Editing WS Endpoint Triggers* in the document *Administering webMethods Integration Server*.
- Create a WS (Web Service) endpoint alias for consumer Web Service Descriptor (WSD) that has a JMS binder. In the Integration Server Administrator, navigate to **Settings > Web Services** and complete the Alias Name, Description, Descriptor Type, and Transport Type fields, as described in the section *Creating an Endpoint Alias for a Consumer Web Service Descriptor for Use with JMS* in the document *Administering webMethods Integration Server*.

Input Parameters

`Property`	*String.* The custom message properties Mediator will use to authenticate incoming requests for the native API. To add additional rows, use the plus button.	
	Value	**Description**
	`Name`	*String.* The name of the property.
	`Value`	*String.* The value of the property.

Straight Through Routing

This action routes the incoming requests to the Mediator directly to the native API.

1. Configure Integration Server: Set up keystores and truststores in Integration Server, as described in *Securing Communications with the Server* in the document *Administering webMethods Integration Server*.
2. Configure Mediator: In the Integration Server Administrator, navigate to **Solutions > Mediator > Administration > General** and complete the IS Keystore Name, IS Truststore Name and Alias (signing) fields, as described in *Keystore Configuration* in the document *Administering WebMethods Mediator*.

When this action is configured for an API, Mediator ensures that requests from the client are parsed directly to the native API you specify. This action also includes a set of configuration properties for the Mediator to process the incoming requests for the native API.

Input Parameters

<table>
<tr><th>Field</th><th colspan="3">Description</th></tr>
<tr><td>Route To</td><td colspan="3"><i>URI. Mandatory.</i> Enter the URL of the native API endpoint to route the request to in case all routing rules evaluate to False. For example:

http://mycontainer/creditCheckService

Click the Configure Endpoint Properties icon (next to the Route To field) if you want to configure a set of properties for the specified endpoint.

Alternatively, Mediator offers “Local Optimization” capability if the native endpoint is hosted on the same Integration Server as webMethods Mediator. With local optimization, API invocation happens in-memory and not through a network hop.

Specify the native API in either of the following forms:

local://<<i>Service-full-path</i>>

OR

local://<<i>server</i>>:<<i>port</i>>/ws/<<i>Service-full-path</i>>

For example:

local://MyAPIFolder:MyLocalAPI

which points to the endpoint API MyLocalAPI which is present under the folder MyAPIFolder in Integration Server.</td></tr>
<tr><td rowspan="4">Configure Endpoint Properties icon</td><td colspan="3">Optional. This icon displays the Endpoint Properties dialog box that enables you to configure a set of properties for the Mediator to route incoming requests to the native API as follows:</td></tr>
<tr><td>SOAP Optimization Method</td><td colspan="2">For SOAP-based APIs. Specifies the optimization methods that Mediator can use to parse SOAP requests to the native API.</td></tr>
<tr><td></td><td>Value</td><td>Description</td></tr>
<tr><td></td><td>MTOM</td><td>Mediator will use the Message Transmission Optimization Mechanism (MTOM) to parse SOAP requests to the API.</td></tr>
<tr><td></td><td></td><td>SwA</td><td>Mediator will use the SOAP with Attachment (SwA) technique to parse SOAP requests to the API.</td></tr>
</table>

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td></td><td></td><td>
<table>
<tr><td>None</td><td>Default. Mediator will not use any optimization method to parse the SOAP requests to the API.</td></tr>
<tr><td colspan="2">Note:

1. Bridging between SwA and MTOM is not supported. If a client sends a SwA request, Mediator can only forward SwA to the native API. The same is true for MTOM, and applies to responses received from the native API. That is, a SwA or MTOM response received by Mediator from a native API will be forwarded to the client using the same format it received.
2. When sending SOAP requests that do not contain a MTOM or SWA attachment to a native API that returns an MTOM or SWA response, the request 'Accept' header must be set to 'multipart/related'. This is necessary so Mediator knows how to parse the response properly.</td></tr>
</table>
</td></tr>
<tr><td></td><td>HTTP Connection Timeout</td><td>Number. Optional. Specifies the time interval (in seconds) after which a connection attempt will timeout. If a value 0 is specified (or if the value is not specified), Mediator will use the value specified in the Connection Timeout field (go to Integration Server Administrator > Settings > Extended). Default: 30 seconds.</td></tr>
<tr><td></td><td>Read Timeout</td><td>Number. Optional. Specifies the time interval (in seconds) after which a socket read attempt will timeout. If a value 0 is specified (or if the value is not specified), Mediator will use the value specified in the Read Timeout field (Open the Integration Server Administrator. Go to > Settings > Extended.). Default: 30 seconds.</td></tr>
<tr><td></td><td>SSL Configuration</td><td>
Optional. To enable SSL client authentication that Mediator will use to authenticate incoming requests for the native API, you must specify values for both the Client Certificate Alias field and the IS Keystore Alias field. If you specify a value for only one of these fields, a deployment error will occur.

Note: SSL client authentication is optional; you may leave both fields blank.

Prerequisite: You must set up the key alias and keystore properties in the Integration Server. For the procedure, see the section Securing Communications with the Server in the documentwebMethods Integration Server Administrator's Guide.

You will use these properties to specify the following fields:
<table>
<tr><th>Value</th><th>Description</th></tr>
</table>
</td></tr>
</table>

Field	Description		
		`Client Certificate Alias`	*Mandatory.* The client's private key to be used for performing SSL client authentication.
		`IS Keystore Alias`	*Mandatory.* The keystore alias of the instance of Integration Server on which Mediator is running. This value (along with the value of Client Certificate Alias) will be used for performing SSL client authentication.
	`WS Security Header`	*For SOAP-based APIs.* Indicates whether Mediator should pass the WS-Security headers of the incoming requests to the native API.	
		Value	**Description**
		`Remove processed ↩ security headers`	*Default*. Removes the security header if it is processed by Mediator (i.e., if Mediator processes the header according to the API's security run-time action). Note that Mediator will *not* remove the security header if *both* of the following conditions are true: 1) Mediator did not process the security header, and 2) the `mustUnderstand` attribute of the security header is 0/false).
		`Pass all security headers`	Passes the security header, even if it is processed by Mediator (i.e., even if Mediator processes the header according to the API's security action).

Throttling Traffic Optimization

This action limits the number of API invocations during a specified time interval, and sends alerts to a specified destination when the performance conditions are violated.

Reasons for limiting the API invocation traffic include:

- To avoid overloading the back-end services and their infrastructure.
- To limit specific clients in terms of resource usage (that is, you can use the "Monitor Service Level Agreement" action to monitor performance conditions for a particular client, together with "Throttle API Usage" to limit the resource usage).
- To shield vulnerable servers, services, and even specific operations.
- For API consumption metering (billable pay-per-use APIs).

Note: To enable Mediator to publish performance metrics, you must configure Mediator to communicate with CentraSite (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > CentraSite Communication**). For the procedure, see the section *Configuring Communication with CentraSite* in the document *Administering webMethods Mediator*.

Input Parameters

`Soft Limit`	*Number Optional.* The maximum number of invocations allowed per `Interval Value` before issuing an alert. Reaching the soft limit will not affect further processing of requests (until the `Hard Limit` is reached). **Note:** The limit is reached when the total number of invocations coming from all *all* the clients (specified in the `Limit Traffic for Applications` field) reaches the limit. Soft Limit is computed in an asynchronous manner; thus when multiple requests are made at the same time, it may be possible that the Soft Limit alert will not be strictly accurate.
`Alert Message for Soft Limit`	*String. Optional.* A text message to include in the soft limit alert.
`Hard Limit`	Required. The maximum number of invocations allowed per `Interval Value` before stopping the processing of further requests and issuing an alert. Typically, this number should be higher than the soft limit. **Note:** The limit is reached when the total number of invocations coming from all *all* the clients (specified in the `Limit Traffic for Consumers` field) reaches the limit. Hard Limit is computed in an asynchronous manner; thus when multiple requests are made at the same time, it may be possible that the Hard Limit alert will not be strictly accurate.
`Alert Message for Hard Limit`	*String. Optional.* A text message to include in the hard limit alert.
`Alert for Consumer Applications`	*String.* The consumer application(s) that this action applies to. To specify multiple consumers, use the  button to add rows, or select **Any Consumer** to apply this action to any consumer application.
`Alert Interval`	*String.* The amount and unit (Minutes, Hours, Days or Weeks) of time for the soft limit and hard limit to be reached.
`Alert Frequency`	*String.* Frequency to issue alerts. **Value** — **Description**

	`Every Time`	Default. Issues an alert every time the specified condition is violated.
	`Only Once`	Issues an alert only the first time the specified condition is violated.
`Alert Destination`	*String. Optional.* A place to log the alerts. **Important:** Ensure that Mediator is configured to send event notifications to the destination(s) you specify here. For details, see *Alerts and Transaction Logging* in the document *Administering webMethods Mediator*.	
	Value	**Description**
	`CentraSite`	Sends the alerts to the API's Events profile in CentraSite. Prerequisite: You must configure Mediator to communicate with CentraSite (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > CentraSite Communication**). For the procedure, see the section *Configuring Communication with CentraSite* in the document *Administering webMethods Mediator*.
	`Local Log`	Sends the alerts to the server log of the Integration Server on which Mediator is running. Also choose a value in the `Log Level` field: ■ Info: Logs error-level, warning-level, and informational-level alerts. ■ Warn: Logs error-level and warning-level alerts. ■ Error: Logs only error-level alerts. **Important:** The Integration Server Administrator's logging level for Mediator should match the logging level specified for this action (go to **Settings > Logging > Server Logger**).
	`SNMP`	Sends the alerts to CentraSite's SNMP server or a third-party SNMP server. Prerequisite: You must configure the SNMP server destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > Email**). For the procedure, see the section *SNMP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.

	`Email`	Sends the alerts to an SMTP email server, which sends them to the email address(es) you specify here. To specify multiple addresses, use the  button to add rows. Prerequisite: You must configure the SMTP server destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > Email**). For the procedure, see the section *SMTP Destinations for Run-Time Events* in the document *Administering webMethods Mediator*.
	`EDA`	Mediator can use EDA to log the payloads to a database. Prerequisite: You must configure the EDA destination (in the Integration Server Administrator, go to **Solutions > Mediator > Administration > EDA**). For the procedure, see the section *EDA Configuration for Publishing Run-Time Events and Metrics* in the document *Administering webMethods Mediator*.

Validate Schema

This action validates all XML request and/or response messages against an XML schema referenced in the WSDL.

Mediator can enforce this action for messages sent between APIs. When this action is configured for a proxy API, Mediator validates XML request messages, response messages, or both, against the XML schema referenced in the WSDL.

Input Parameters

`Validate SOAP Message(s)`	*Object.* Validates request and/or response messages. You may select both Request and Response.	
	Value	**Description**
	`Request`	Validate all requests.
	`Response`	Validate all responses.

Important: Be aware that Mediator does not remove `wsu:Id` attributes that may have been added to a request by a client as a result of security operations against request elements (i.e., signatures and encryptions). In this case, to avoid schema validation failures you would have to add a **Request Transformation** action or a **Response Transformation** action to the API so that the requests and responses are passed to an XSL transformation file that removes the `wsu:Id` attribute.

4 Computed Runtime Actions

CentraSite Business UI offers you the possibility to add computed runtime actions into the policy workflow; this gives you the option to define your own runtime action; which means that you can implement your own algorithms for representing the action's user interface.

Computed runtime actions let you create your own layout by using a UI Rendering Concept. You can also specify your own rendering logic to display the computed values. You could, for example, create a custom display of the attribute as a drop down or a radio button.

A computed runtime action can be implemented using the GWT framework. For a computed runtime action, you create an archive file that contains the plug-in definition, and you load the archive file in the CentraSite *CentraSiteBUIExtension* folder.

Writing Your Own Computed Runtime Action

A computed runtime action can be implemented as a plug-in. The prepared plug-in is a collection of files in a specific directory structure. After implementing the plug-in, the files are copied into the *CentraSiteBUIExtension* folder under:

```
<CentraSiteInstallDir>\demos
```

In the following sections, we demonstrate a sample framework named “MyComputedRuntimeAction” that illustrates how a custom computed runtime action may be set up.

You may use this sample as a guideline, adapting it and renaming it to suit your individual requirements. The sample indicates where customization is required.

The following topics are discussed in this document:

- The Build Environment
- Implementation Guidelines for Computed Runtime Action
- Setting up the Computed Action Plug-in
- Activating the Computed Action
- Sample Computed Runtime Action

The Build Environment

This section explains the build environment for generating the files that are used for the GUI and for compiling the necessary Java source files. It assumes the use of Ant, the Java-based build tool.

The following file system structure under the computed runtime action directory is assumed:

Name of File or Folder	Description
src	This folder that holds the Java source files.
lib	This folder contains the archive file, plug-in's executor class and the external libraries.
build.xml	The Ant input file for building the destination files

The *Ant* file,*build.xml* can be used to establish a custom computed profile.

The classpath for the build step must refer to all JAR files contained in the *redist* folder of the CentraSite installation. Add these JAR files to the build path of your java project also.

Implementation Guidelines for Computed Runtime Action

In order to create, install and use plug-ins, you must perform the following tasks:

- Implementation for Computed Action UI
- Implementation for Computed Action Parser

This section does not explain all the details of the Java source file; its purpose is to indicate the code that must be modified to suit your environment.

Implementation for Computed Action UI

src\com\softwareag\centrasite\bui\extension\client\runtime\action\MyComputedRuntimeActionWidget.java

```
public class MyComputedRuntimeActionWidget extends Composite {
 private PolicyActionJSO policyActionJso = null;
 private TextBox valueBox = null;
 private static final String WARNING_CSS = "loginTextBoxErrorBorder";

 public MyComputedRuntimeActionWidget(String policyActionJson) {
  FlowPanel container = new FlowPanel();
  initWidget(container);

  policyActionJso = getPolicyActionJso(policyActionJson);
  if (policyActionJso == null) {
   Label helloLabel = new Label("The JSON content is empty");
   container.add(helloLabel);
   return;
  }

  //Render widgets
  container.add(getParametersView(policyActionJso));
 }

 private Widget getParametersView(PolicyActionJSO policyActionJso) {
  FlowPanel parametersContainer = new FlowPanel();
```

```
 JsArray<ParameterJSO> parameters = policyActionJso.getParameters();
 if (parameters == null) {
  return parametersContainer;
 }

 for (int i = 0; i < parameters.length(); i++) {
  parametersContainer.add(getParameterView(parameters.get(i)));
 }

 return parametersContainer;
}

private Widget getParameterView(ParameterJSO parameterJso) {
 FlowPanel parameterContainer = new FlowPanel();
 Label nameLabel = new Label(parameterJso.getName());
 parameterContainer.add(nameLabel);

 valueBox = new TextBox();
 valueBox.setLayoutData(parameterJso.getId());

 String[] values = parameterJso.getValues();
 if (values != null && values.length > 0) {
  valueBox.setValue(values[0]);
 }

 parameterContainer.add(valueBox);
 return parameterContainer;
}

public static native PolicyActionJSO getPolicyActionJso(String json) /*-{
 return eval('(' + json + ')');
}-*/;

public String getJson() {
 JsArray<ParameterJSO> parameters = policyActionJso.getParameters();
 ParameterJSO parameterJso = null;
 if (parameters != null && parameters.length() > 0) {
  parameterJso = parameters.get(0);

  String[] values = {valueBox.getValue()};
  parameterJso.setValues(values);
 }

 return policyActionJso.toJSON();
}

public boolean isValid() {
 String value = valueBox.getValue();
 boolean isValid = (value != null && !"".equals(value));
 if (!isValid) {
  valueBox.addStyleName(WARNING_CSS);
 } else {
```

```
   valueBox.removeStyleName(WARNING_CSS);
  }

  return isValid;
 }
}
```

The `MyComputedRuntimeActionWidget` class extends the class `Composite`, which declares the basic rendering methods for the CentraSite Business user interface.

Implementations	Description
`MyComputedRuntimeActionWidget(String policyActionJson)`	Constructor dictates the user-defined rendering of the action's UI.
`getPolicyActionJson`	Returns the JSON object from the specified object.
`String getJson()`	Returns a JSON encoded string representing the action's parameters.
`boolean isValid()`	Enforces validation logic for the action's parameter values.

Implementation for Computed Action Parser

To implement your own computed runtime action with custom UI rendering, the parser (MyComputedRuntimeActionParser.java) must be located in the service directory. A parser is responsible for generating compressed JSON data from the given policy action instance, and creating a custom rendering of the action instnace using the JSON data.

Here is the frame of the computed runtime action parser implementation:

src\com\softwareag\centrasite\bui\extension\service\ MyComputedRuntimeActionParser.java

```
public class MyComputedRuntimeActionParser extends BasePolicyActionExtensionParser {


 public MyComputedRuntimeActionParser(CentraSiteSession centraSiteSession,
   CentraSitePolicyActionTemplate actionTemplate, CentraSitePolicyActionInstance actionInstance) {
  super(centraSiteSession, actionTemplate, actionInstance);
 }

 @Override
 public CentraSitePolicyActionInstance getActionInstance(String json) throws CLLException {
  Gson gson = new Gson();
  MyComputedRuntimeActionInfo = gson.fromJson(json, MyComputedRuntimeActionInfo.class);
  if (actionInfo == null) {
   return null;
```

```
  }

  CentraSitePolicyActionInstance policyActionInstance = null;
  CentraSiteObjectManager objectManager = ↩
getCentraSiteSession().getCentraSiteObjectManager();

  if (actionInfo.isActionInstance()) {
   policyActionInstance = objectManager.getPolicyActionInstance(actionInfo.getId());

  } else {
   policyActionInstance = ↩
objectManager.createPolicyActionInstance(actionInfo.getId());
  }

  if (policyActionInstance == null) {
   return null;
  }

  setParameterValues(policyActionInstance, actionInfo);
  return policyActionInstance;
 }

 private void setParameterValues(CentraSitePolicyActionInstance policyActionInstance,

   MyComputedRuntimeActionInfo actionInfo) throws CLLException {
  List<MyComputedRuntimeParameterInfo> parameters = actionInfo.getParameters();
  if (parameters == null || parameters.isEmpty()) {
   return;
  }

  MyComputedRuntimeParameterInfo parameterInfo = parameters.get(0);
  Collection<Object> convertedParameterValues = new ArrayList<Object>();
  convertedParameterValues.addAll(parameterInfo.getValues());
  policyActionInstance.setAttributeValue(parameterInfo.getId(), ↩
convertedParameterValues);
 }

 @Override
 public String getJson() throws CLLException {
  Gson gson = new Gson();

  MyComputedRuntimeActionInfo actionInfo = null;
  if (getActionInstance() != null) {
   CentraSitePolicyActionTemplate policyActionTemplate = ↩
getActionInstance().getCentraSitePolicyActionTemplate();
   actionInfo = new MyComputedRuntimeActionInfo(getActionInstance().getId(), ↩
policyActionTemplate.getName());
   actionInfo.setActionId(policyActionTemplate.getId());
   actionInfo.setIsActionInstance(true);
  } else if (getActionTemplate() != null) {
   actionInfo = new MyComputedRuntimeActionInfo(getActionTemplate().getId(), ↩
getActionTemplate().getName());
```

```
   actionInfo.setActionId(getActionTemplate().getId());
  }

  fillParameterInfos(getActionTemplate(), actionInfo);
  return (actionInfo != null ? gson.toJson(actionInfo) : null);
 }

 private void fillParameterInfos(CentraSitePolicyActionTemplate actionTemplate,
   MyComputedRuntimeActionInfo actionInfo) throws CLLException {
  if (actionTemplate == null) {
   return;
  }

  Collection<CentraSiteObjectAttribute> attributes = actionTemplate.getAttributes();

  if (attributes == null || attributes.isEmpty()) {
   return;
  }

  List<MyComputedRuntimeParameterInfo> parameters = new ↩
ArrayList<MyComputedRuntimeParameterInfo>(attributes.size());
  MyComputedRuntimeParameterInfo parameter = null;
  for (CentraSiteObjectAttribute attribute : attributes) {
   parameter = new MyComputedRuntimeParameterInfo(attribute.getName(), ↩
attribute.getDisplayName());
   parameters.add(parameter);
  }

  actionInfo.setParameters(parameters);
 }
}
```

Setting up the Computed Action Plug-in

The following diagram describes the main methods on each of the two Java source files *MyComputedRuntimeActionWidget.java* and *MyComputedRuntimeActionParser.java* and describes the type of functions that they serve.

#	Description
1	The `getPolicyActionJson` method returns the JavaScript Object (JSO) from the given JSON-formatted string.
2	The `getActionInstance` method returns a `CentraSitePolicyActionInstance` object from the given JSON-formatted string.
3	The `String getJson` method returns a JSON-formatted string from the existing policy action action instance.

#	Description
4	The `boolean isValid()` method enforces a validation logic for the user-defined rendering of the runtime action.

Assuming that you have set up all the Java files correctly in the directories, you should be able to build with the command:

```
ant -f build.xml jar all
```

Activating the Computed Action

After you define the computed action as a plug-in (extension point) with the above steps, enable the computed action in the Business UI configuration file *centrasite.xml* in order to display the action in the policy accordion.

Important: Remember that the action parameters defined in the configuration file are editable and cannot be protected.

To activate the plug-in

1 Open the *centrasite.xml* file.

The configuration file is located in the `cast\cswebapps\BusinessUI\system\conf` directory.

2 Navigate to the property lines <*UIProperties*> -> <*Extensions*> -> <*PolicyActions*>

3 Append the property statement for your custom computed runtime action ("My-ComputedRuntimeAction") as below:

```
<PolicyActions>
   <PolicyAction id="uddi:44e3e2de-064c-432f-b67a-8fbca0fb04d6" ↩
class="com.softwareag.centrasite.bui.extension.service.MyComputedRuntimeActionParser" ↩
/>
</PolicyActions>
```

wherein,

Parameter	Description
`id`	A unique identifier for the computed action. It uniquely distinguishes an action in the CentraSite registry. If you wish to reconfigure the action at a later stage, you identify the action using this id.
`class`	A parser implementation for the computed action.

4 Save and close the configuration file.

5 Restart Software AG Runtime.

Sample Computed Runtime Action

Your CentraSite installation contains a sample computed runtime action (which is contained in *demos* folder) that you can use to create an archive file for the custom runtime action specific to the CentraSite Business UI.

- *SampleComputedRuntimeAction*