Web Services Guide

Operations Center 5.6

June 2015

Legal Notices

Contents

About This Guide 5

1 Getting Started 7

1.1 Accessing WSAPI 1.1 . . . 7
1.2 Installing the Reference Client . . . 9
1.2.1 Installing and Starting MosWsClient . . . 9
1.2.2 Viewing a List of Available Services . . . 10
1.2.3 Logging In and Establishing a WSAPI Session . . . 10

2 Understanding WSAPI Functionality 11

2.1 High-Level Data Model . . . 11
2.1.1 Elements and their Relationships . . . 12
2.1.2 Alarms . . . 14
2.1.3 Series Data . . . 15
2.1.4 SLA Data . . . 16
2.2 Session Management . . . 16
2.3 Queries . . . 17
2.3.1 Throttle Mechanism . . . 17
2.3.2 Element Queries . . . 18
2.3.3 Alarm Queries . . . 21
2.3.4 Series Data Queries . . . 21
2.3.5 Health Data Queries . . . 23
2.3.6 SLA Queries . . . 24
2.4 Miscellaneous Services . . . 24
2.5 Error Reporting . . . 25

3 Working with Elements 27

3.1 Writing Elements . . . 27
3.2 Changing Element Relationships . . . 28
3.3 Changing Element Property Page Data . . . 29
3.3.1 Condition Property Page . . . 30
3.3.2 Elements Property Page . . . 32
3.3.3 Status Property Page . . . 33

4 Web Services Client Operation 35

4.1 Using a Properties File . . . 35
4.2 Meta-Commands . . . 35
4.2.1 Simple Meta-Commands . . . 36
4.2.2 Post Processor Meta-Commands . . . 36
4.3 Record/Playback . . . 38

5 Extending and Embedding the Web Services Client 41

5.1 ClientContext . . . 42
5.2 MosWsClient Class . . . 43
5.3 ServiceFactory and ServiceCall . . . 44

5.4 MetacommandProcessor and PostProcessor 45

A WSAPI 1.1 **47**

About This Guide

The *Web Services Guide* provides information on the Web Services Application Programmer Interface (WSAPI), which is an integration point for customer or third-party applications to interact with the Operations Center Server. The API provides a number of services that allow remote applications to query data warehoused in the Operations Center Server. You can also use the API to create, update, or remove a limited set of elements.

- Chapter 1, “Getting Started,” on page 7
- Chapter 2, “Understanding WSAPI Functionality,” on page 11
- Chapter 3, “Working with Elements,” on page 27
- Chapter 4, “Web Services Client Operation,” on page 35
- Chapter 5, “Extending and Embedding the Web Services Client,” on page 41
- Appendix A, “WSAPI 1.1,” on page 47

Audience

This document is for developers who want to use the Web Services API to integrate their applications with the Operations Center Server.

Feedback

We want to hear your comments and suggestions about this manual and the other documentation included with this product. Please use the *User Comments* feature at the bottom of each page of the online documentation.

Additional Documentation & Documentation Updates

This guide is part of the Operations Center documentation set. For the most recent version of the *Web Services Guide* and a complete list of publications supporting Operations Center, visit our Online Documentation Web Site at Operations Center 5.6 online documentation.

The Operations Center documentation set is also available as PDF files on the installation CD or ISO; and is delivered as part of the online help accessible from multiple locations in Operations Center depending on the product component.

Additional Resources

We encourage you to use the following additional resources on the Web:

- NetIQ User Community (https://www.netiq.com/communities/): A Web-based community with a variety of discussion topics.
- NetIQ Support Knowledgebase (https://www.netiq.com/support/kb/?product%5B%5D=Operations_Center): A collection of in-depth technical articles.
- NetIQ Support Forums (https://forums.netiq.com/forumdisplay.php?26-Operations-Center): A Web location where product users can discuss NetIQ product functionality and advice with other product users.

Technical Support

You can learn more about the policies and procedures of NetIQ Technical Support by accessing its Technical Support Guide (https://www.netiq.com/Support/process.asp#_Maintenance_Programs_and).

Use these resources for support specific to Operations Center:

- Telephone in Canada and the United States: 1-800-858-4000
- Telephone outside the United States: 1-801-861-4000
- E-mail: support@netiq.com (support@netiq.com)
- Submit a Service Request: http://support.novell.com/contact/ (http://support.novell.com/contact/)

Documentation Conventions

A greater-than symbol (>) is used to separate actions within a step and items in a cross-reference path. The > symbol is also used to connect consecutive links in an element tree structure where you can either click a plus symbol (+) or double-click the elements to expand them.

When a single pathname can be written with a backslash for some platforms or a forward slash for other platforms, the pathname is presented with a forward slash to preserve case considerations in the UNIX* or Linux* operating systems.

A trademark symbol (®, ™, etc.) denotes a NetIQ trademark. An asterisk (*) denotes a third-party trademark.

1 Getting Started

All server components of WSAPI are included in the default installation of the Operations Center Server. For prior releases, check the Download Web site (https://dl.netiq.com/index.jsp) for patch availability.

A license separate from Operations Center is required to activate the WSAPI services. For licensing information, contact Technical Support.

The original version of WSAPI was 1.0. Version 1.1 was released in November 2008. WSAPI 1.1 adds three new service calls, as listed in Table 1-1, designed to facilitate the management of large services models through Web services.

Table 1-1 *New WSAPI Service Calls*

Command	Description
createElements(session:WsSession, transactionBlockSize:int, data:ElementCreateData[]):ElementsWriteResponse	Creates new elements on the Operations Center Server.
updateElements(session:WsSession, , transactionBlockSize:int, data:ElementUpdateData[]):ElementsWriteResponse	Updates elements on the Operations Center Server.
removeElement(session:WsSession, , transactionBlockSize:int, elementDName:string[]):void	Removes elements from the Operations Center Server.

The Web services reference client has been updated to automatically include the 1.1 service calls when a 1.1 connection point is accessed.

For more information, see Appendix A, “WSAPI 1.1,” on page 47.

Review the following sections for how to access WSAPI and to install the reference client:

- Section 1.1, “Accessing WSAPI 1.1,” on page 7
- Section 1.2, “Installing the Reference Client,” on page 9

1.1 Accessing WSAPI 1.1

To verify that WSAPI is available on a Operations Center Server:

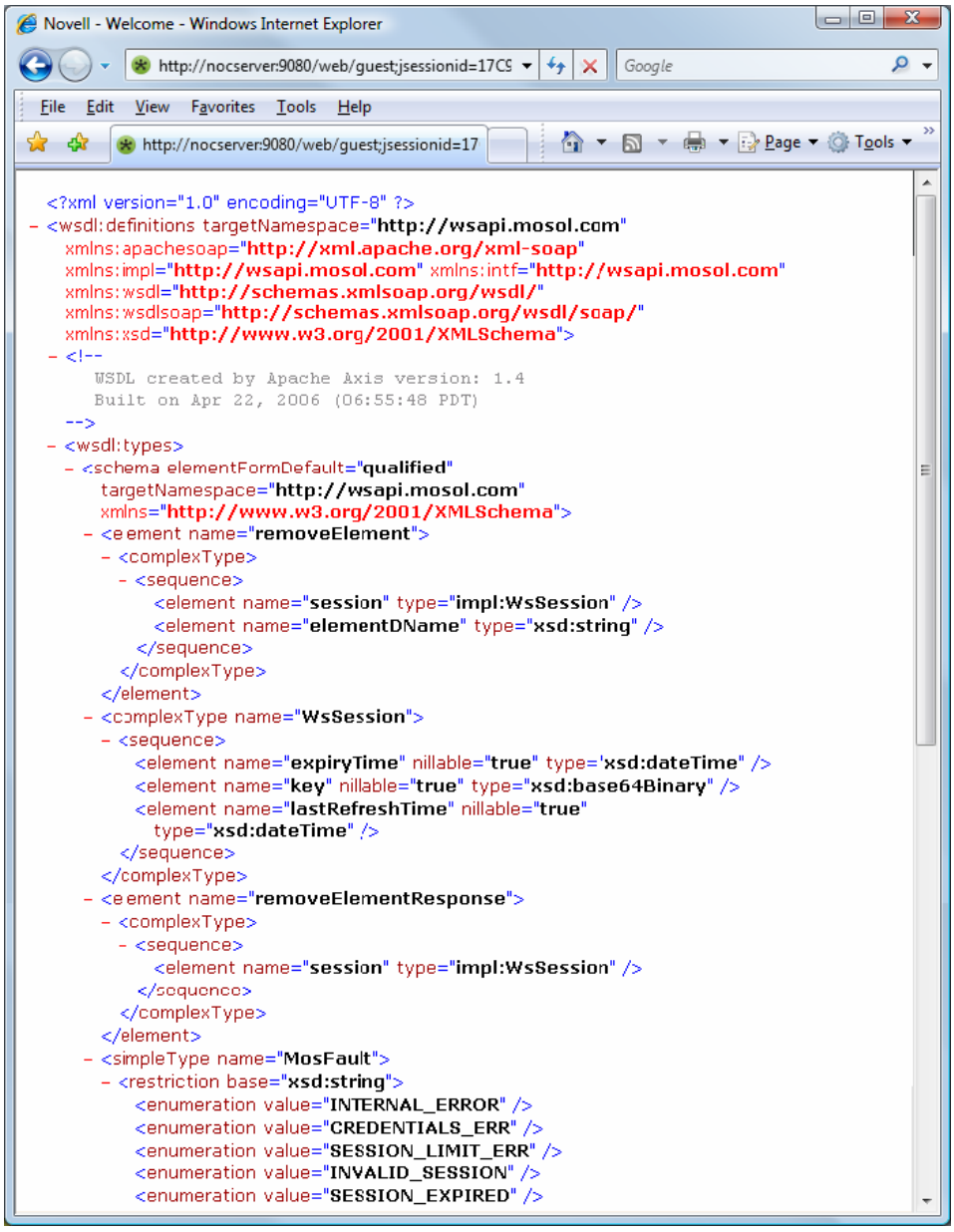
1 In a Web browser, navigate to the WSDL:

```
http://OperationsCenterServerAddress:WebServicePort/wsapi/services/
Moswsapi_1_0?wsdl
```

The interface point for WSAPI 1.1 itself is at:

```
http://OperationsCenterServerAddress:WebServicePort/wsapi/services/
Moswsapi_1_1
```

The WSDL Web page displays, indicating that WSAPI is available on your server:



1.2 Installing the Reference Client

WSAPI includes a fully operational reference client implementation named Web Services Client (MosWsClient). This reference client includes an interactive mode that allows you to invoke any of the services through a command line interface. For more information, see Chapter 4, “Web Services Client Operation,” on page 35.

MosWsClient is not distributed with Operations Center software. To obtain the client, contact Technical Support.

Do the following to establish the reference client:

- Section 1.2.1, “Installing and Starting MosWsClient,” on page 9
- Section 1.2.2, “Viewing a List of Available Services,” on page 10
- Section 1.2.3, “Logging In and Establishing a WSAPI Session,” on page 10

1.2.1 Installing and Starting MosWsClient

1. Unpack the `moswsclient.zip` file.
2. Edit the following properties in `wsclient.properties`:
   - Set `wsapi.servers=http://`*`OperationsCenterServerAddress`*`:`*`WebServicePort`*`/wsapi/services/Moswsapi_1_0`
   - Set `wsapi.default.username=`*`yourUserName`*
   - (Optional) Set `wsapi.default.password=`*`yourPassword`*
3. Start the client by executing `moswsclient.bat` (or `moswsclient.sh` on UNIX systems).

   This starts the client in interactive mode. A prompt displays, indicating that a connection to your server was established, followed by a menu of services:

```
Type /h for help
Established connection to http://localhost:9080/wsapi/services/Moswsapi_1_0
1-login
2-logout
3-getServerInfo
4-createElement
5-updateElement
6-removeElement
7-getRootElements
8-getElement
9-getAssociatedElements
10-findElements
11-nextElements
12-getAllSlas
13-nextSlas
14-getSlaHealthData
15-nextHealthData
16-getAlarms
17-getHistoricalAlarms
18-getBreaches
19-getOutages
20-nextAlarms
21-getSeriesData
22-nextSeriesData
23-getIcon
24-getIcons

Service: Select from menu (1 needed):
```

4. Continue with Section 1.2.2, “Viewing a List of Available Services,” on page 10.

1.2.2 Viewing a List of Available Services

1 Type `/h` and then press Enter.

The following list of available commands displays:

```
Available Commands:
    /c = Disable cache updates (CacheBuilderPostProc)
    /d = Enable display of SOAP messages (ShowSoapPostProc)
    /f = Enable prompting to save SOAP Messages (SaveSoapPostProc)
    /h = List available meta-commands (CommandHelpProcessor)
    /p = Play a recorded Test Script (PlaybackProcessor)
    /r = Start recording a new test script (TestScriptRecorder)
    /x = Cancel current service/activity (CancelProcessor)
Press [ENTER] to continue:
```

2 Enter `/d` to enable the display of SOAP messages.

This causes the SOAP message exchange for each invoked service to display on the screen. (At any time, you can enter `/d` to disable the feature.)

3 Continue with Section 1.2.3, “Logging In and Establishing a WSAPI Session,” on page 10.

1.2.3 Logging In and Establishing a WSAPI Session

1 Type `1` and press Enter.

```
Service: Select from menu (1 needed): 1
Enter user name [admin]:
```

Default values display in square brackets in the prompt, such as the user name shown above.

2 Type a new value, or just press Enter to accept the default.

3 When prompted for your password, enter your user password.

A confirmation displays:

```
Successfully logged in user admin

SOAP Request: <?xml version="1.0" encoding="UTF-8"?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"><soapenv:Body><login xmlns="http://
wsapi.mosol.com"><userName>admin</
userName><passwordHash>Nb4sHPLMSJvbFrZ4zXPNeA==</passwordHash><hashType>MD5</
hashType></login></soapenv:Body></soapenv:Envelope>
SOAP Response: <?xml version="1.0" encoding="utf-8"?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"><soapenv:Body><loginResponse xmlns="http://
wsapi.mosol.com"><loginReturn><expiryTime>2007-03-10T23:57:08.301Z</
expiryTime><key>W2FkbWluOjRdMTE3MzU3MDcyODMwMQ==</key><lastRefreshTime>2007-
03-10T23:52:08.301Z</lastRefreshTime></loginReturn></loginResponse></
soapenv:Body></soapenv:Envelope>
```

You have now established a WSAPI session with the server. The test client automatically includes this session in subsequent Web service requests.

2 Understanding WSAPI Functionality

This section describes the broad concepts of how WSAPI works, and what clients need to do to successfully interact with it. It also provides an overview of the data model exposed by WSAPI and each of the available services and some of their interdependencies.

Before implementing your own client, you must select a Web services framework if you do not already have one. There are many commercial and open-source toolkits available to facilitate integration to the published WSDL. The WSAPI implementation conforms to SOAP 1.1 and 1.2 specifications. WSAPI uses Wrapped Document Literal encoding; this style currently has the broadest platform support. You can use any Web services integration platform that supports these standards and can digest the published WSDL. One recommendation is the Axis toolkit, located at the Axis Web site. It is available for both Java and C++ environments.

When you begin development, refer to the Javadoc for answers to more detailed questions about the API.

The following sections provide information about the WSAPI functionality:

- Section 2.1, “High-Level Data Model,” on page 11
- Section 2.2, “Session Management,” on page 16
- Section 2.3, “Queries,” on page 17
- Section 2.4, “Miscellaneous Services,” on page 24
- Section 2.5, “Error Reporting,” on page 25

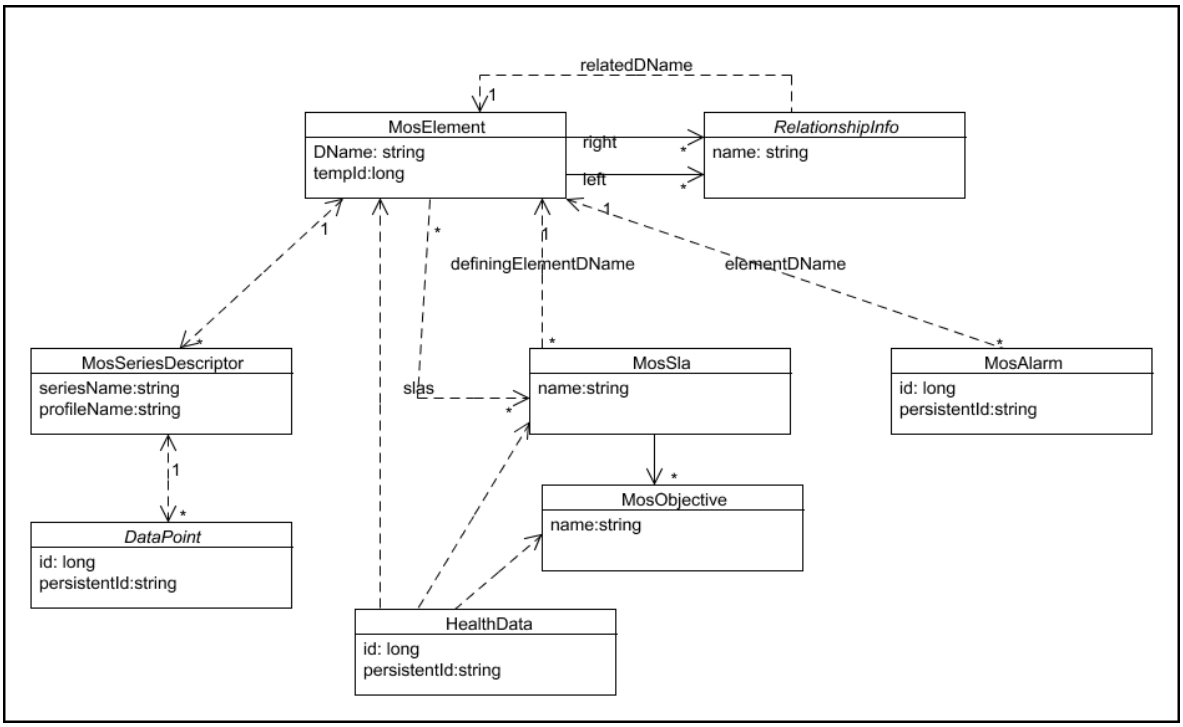
2.1 High-Level Data Model

This section describes some of the core entities exposed in the WSAPI. Understanding how these entities relate to each other and to the entities exposed in the client user interface (UI) is necessary to effectively use the WSAPI.

Figure 2-1 shows the high-level relationships among the following WSAPI entities:

- MosElement
- MosAlarm
- MosSeriesDescriptor, DataPoint
- MosSla, HealthData

Figure 2-1 *High-Level WSAPI Entity Relationships*



The following sections provide further explanation:

- Section 2.1.1, “Elements and their Relationships,” on page 12
- Section 2.1.2, “Alarms,” on page 14
- Section 2.1.3, “Series Data,” on page 15
- Section 2.1.4, “SLA Data,” on page 16

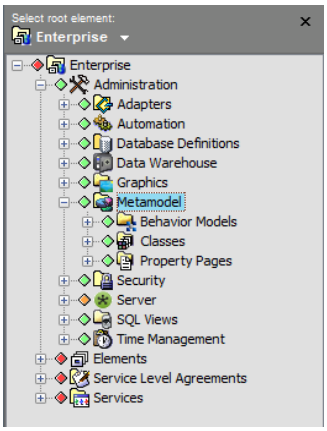
2.1.1 Elements and their Relationships

A MosElement is the WSAPI representation of an element in the Operations Center Server. Elements are the core structural entity in the Operations Center Server. An element can represent any of the following:

- A physical system or a service in the customer environment
- A service internal to the Operations Center Server
- Logical groupings of services and/or systems

In the Explorer pane of the Operations Center console, each node in the tree structure is an element.

Figure 2-2 *MosElement Identity Fields*



The fields in Table 2-1 identify a MosElement.

Table 2-1 *MosElement Identity Field Descriptions*

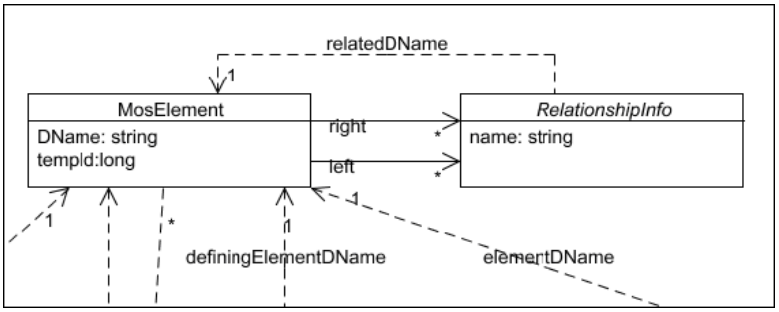
Field	Description
DName (string)	Elements are identified by a distinguished name (DName) field composed of the name and class of the element and all of its parents in the tree hierarchy. Moving or renaming an element changes the element's DName as well as the DNames of its children.
tempId (long)	A unique numeric identifier assigned to an element by the Operations Center Server during server startup or during element creation, if the element is created after server initialization. Unlike the DName, the tempId does not change when elements are moved or renamed. Use the serverStartTime field of MosServerInfo to determine if a tempId value is still valid.

The element tree structure in the Operations Center console can reflect both physical and logical associations among the displayed elements. A physical association is a name relationship as reflected in the element DNames. Each nonroot element has exactly one parent and can have zero to many children.

Logical associations are created to provide various views of element data. An example of a logical relationship is the ability to configure elements as links that contribute to the state of a service model. Although these elements are not physical children of the service model, they can display under the service model in the tree hierarchy.

When relationship information is included in a MosElement, it is represented as either a “left” or “right” RelationshipInfo object embedded in a MosElement. In most cases, “right” relationships can be viewed as associating child or source elements and “left” relationships as associating parent or destination elements:

Figure 2-3 *MosElement Relationship Information*



2.1.2 Alarms

A MosAlarm is the WSAPI representation of an alarm in the Operations Center Server. Alarms represent recorded events that are captured by an adapter (such as a T/EC event) or by an internal configuration (such as an SLA breach). Although each alarm is tied to a single element (sometimes called the home element), elements can be configured to share the alarms of associated elements. Alarms can be current or historical.

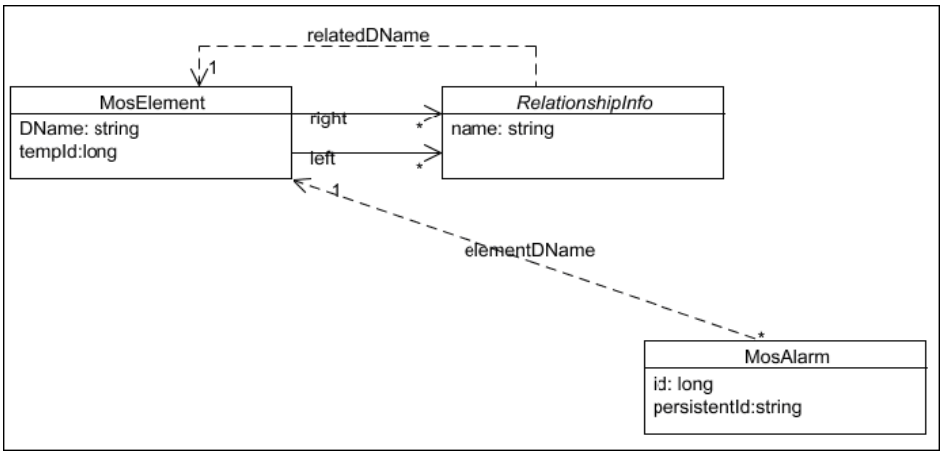
Table 2-2 shows MosAlarm fields.

Table 2-2 *MosAlarm Identity Fields*

Field	Description
elementDName (string)	The DName of the element to which the alarm is attached; the alarm’s home element. The other IDs in the alarm should be scoped to the home element.
id (long)	Similar to MosElement.tempId, this value is assigned when the alarm is created and remains constant for the life of the alarm. This value might not survive a server restart.
persistentId (string)	Use this ID for alarms that are persistent across a server restart.

The element DName ties MosAlarm to the element to which the alarm is attached as shown in Figure 2-4 on page 15.

Figure 2-4 *The DName Element Field for the MosAlarm Element*



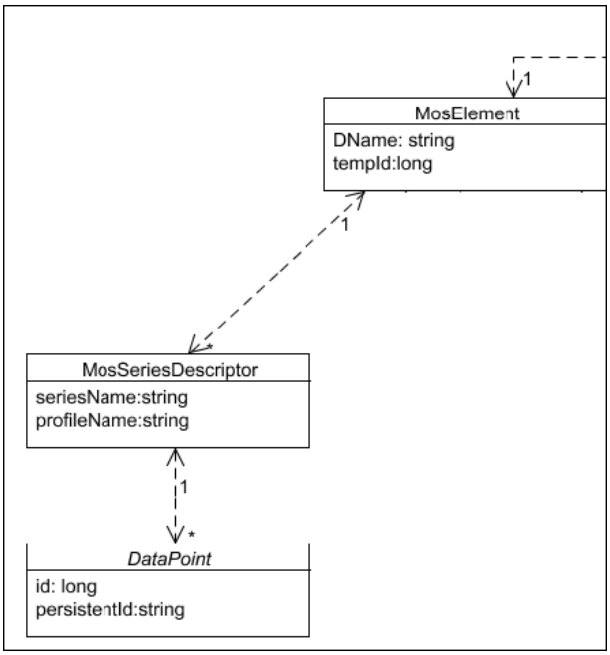
2.1.3 Series Data

Series data refers to element data that is tracked over time. In the Operations Center console, series data is usually displayed as charts in the Performance view for the element.

The WSAPI exposes series data by using several types of DataPoint objects, each representing a single value at a specified point in time, and a MosSeriesDescriptor object to describe a series of related DataPoints.

A MosSeriesDescriptor is scoped to a single MosElement and is distinguished within an element with the combination of the profileName and expressionName fields. DataPoints within a series are distinguished by time stamp:

Figure 2-5 *A MosSeriesDescriptor Object*

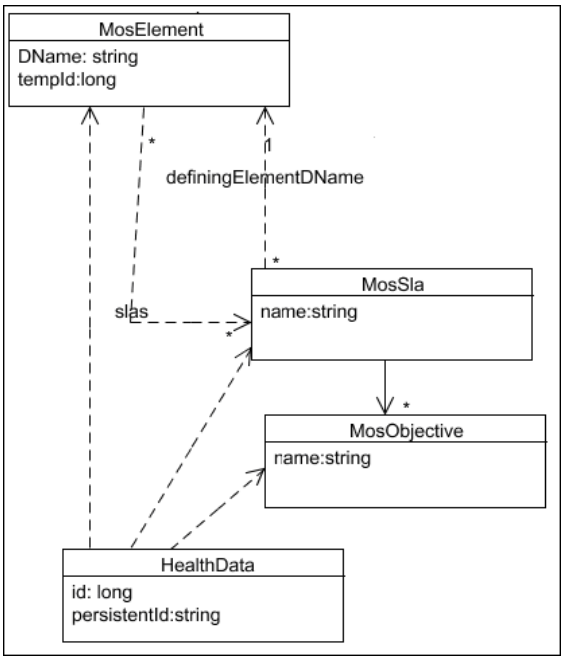


2.1.4 SLA Data

A MosSla is the WSAPI representation of a Service Level Agreement (SLA) in the Operations Center server. An SLA is a collection of objectives (MosObjective) and other configuration data that are used to evaluate the performance of elements. Each SLA is associated with a specific element and is distinguished within that element by the SLA name. An SLA can be shared with child or source elements in much the same way that alarms can be shared with parent or destination elements. The performance of an SLA over time is exposed as a set of HealthData objects.

HealthData objects are similar to series data DataPoint objects, except that they represent a period of time instead of an instance and they are calculated dynamically based on specified request criteria:

Figure 2-6 *A MosSla Element*



2.2 Session Management

WSAPI sessions are established by passing a valid user ID and password (MD5 hashed) to the WSAPI login service. A successful login returns a session object that must be included in each subsequent service call. All Web service activities are subject to the ACL restrictions of the logged-in user. WSAPI sessions are tied to the originating client IP, but not to any particular connection or HTTP session.

WSAPI sessions remain active until the session object is sent to the logout service, or the session times out because of inactivity. (The WSAPI session timeout is specified by the wsapi.sessionTimeout property in the `formula.properties` file.)

2.3 Queries

The WSAPI includes a number of queries that allow callers to retrieve various types of data from the Operations Center server.

- Section 2.3.1, “Throttle Mechanism,” on page 17
- Section 2.3.2, “Element Queries,” on page 18
- Section 2.3.3, “Alarm Queries,” on page 21
- Section 2.3.4, “Series Data Queries,” on page 21
- Section 2.3.5, “Health Data Queries,” on page 23
- Section 2.3.6, “SLA Queries,” on page 24

2.3.1 Throttle Mechanism

To prevent query responses from becoming unmanageably large, many queries support a response throttling mechanism. To limit the response size, the caller specifies a maximum record count, which truncates the response and provides a cursor for retrieving the remaining records.

For example, the following alarms request limits the number of alarms in the response to 50:

```
<getAlarms xmlns="http://wsapi.mosols.com">
<session>

   . . .
</session>
<elementDName>device_port=6789/server_host=devtower18.mosol.com/
computer_minicomputer=Monitor+Distribution/
BEM+Root+Element=Adapter%3A+Operations+Center+Experience+Manager/root=Elements</
elementDName>
<channelName>REALTIME</channelName>
<throttle>
<maxRecords>50</maxRecords>
</throttle>
</getAlarms>
```

If the target element has more than 50 alarms, the response includes the first 50 alarms, a cursor to access the remaining alarms, and a count of the remaining records in the query response:

```
<getAlarmsReturn>
<alarms>
   . . . <!-- 50 Alarm records here -->
</alarms>
<cursor>1172681096613</cursor>
<remainingAlarmsCount>31</remainingAlarmsCount>
</getAlarmsReturn>
```

The caller can then pass the provided cursor value to the nextAlarms service to retrieve the remaining 31 alarms in the query response:

```
<< insert sample nextAlarms here >>>
```

Response throttling is available on Alarm, Element, Series Data, SLA, and SLA Health Data queries. Cursors returned in a throttled query are valid for the duration of the user session.

2.3.2 Element Queries

The API includes the element query services listed in Table 2-3.

Table 2-3 *Element Query Services*

Name	Description
`getElement(session:WsSession, dName:string, inclusionSpec:ElementInclusionSpec, contentSpec:ElementContentSpec, throttle:ResponseThrottle):ElementResultSet`	Retrieves an element and optionally, associated elements.
`getAssociatedElements(session:WsSession, dName:string, inclusionSpec:ElementInclusionSpec, contentSpec:ElementContentSpec, throttle:ResponseThrottle):ElementResultSet`	Retrieves elements associated with the specified element.
`getRootElements(session:WsSession, inclusionSpec:ElementInclusionSpec, contentSpec:ElementContentSpec, throttle:ResponseThrottle):ElementResultSet`	Retrieves elements starting at the Enterprise root.
`findElements(session:WsSession, baseElemDName:string, searchSpec:ElementSearchSpec, contentSpec:ElementContentSpec, throttle:ResponseThrottle):ElementResultSet`	Searches for child elements of a specified element using search strings.
`nextElements(session:WsSession,cursor:string, throttle:ResponseThrottle):ElementResultSet`	Retrieves additional records from a previous query.

Element queries have three basic components:

- **A starting point.** For all queries except getRootElements, the starting point is specified in the call with a DName argument. The getRootElements service assumes the Enterprise root element is the starting point.
- **A specification for the elements to return.** This specification is accomplished by using an `<inclusionSpec>` argument for the `get` services and a `<searchSpec>` for findElements.
- **A specification for the data to include in each returned element.** A `<contentSpec>` argument specifies which data to include in each returned element for each of the service calls.

All element queries are capable of retrieving multiple element records and support a throttle (for more information, see Section 2.3.1, “Throttle Mechanism,” on page 17).

To retrieve the undeliverable portion of an element query response, use the nextElements service.

The following sections describes services used to find and retrieve elements and their contents:

- “<inclusionSpec>” on page 19
- “<searchSpec>” on page 20
- “<contentSpec>” on page 20

<inclusionSpec>

An <inclusionSpec> (WSDL type: <ElementInclusionSpec>) is used to traverse the physical and logical associations of the query's base element to retrieve associated elements (see Section 2.1.1, "Elements and their Relationships," on page 12 for more information on element relationships).

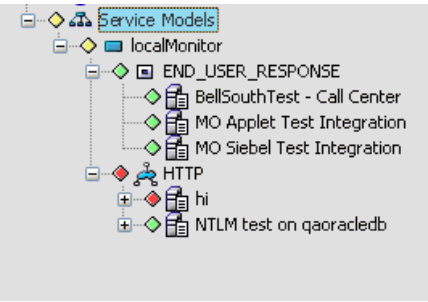
The following is a sample <inclusionSpec>:

```
<inclusionSpec>
  <childDepth>2</childDepth>
  <includeLeftElements>true</includeLeftElements>
  <includeRightElements>true</includeRightElements>
  <forceDiscovery>true</forceDiscovery>
</inclusionSpec>
```

When processing this <inclusionSpec>, the childDepth field first retrieves the physical (name) children of the query's base element, up to the specified depth of 2. Then, for the base element and each included child, any element that is associated through a "left" or "right" relationship is included in the result set.

Each element in a result set (that was created with an <inclusionSpec>) includes lists of associated element DNames in the same result set. These DName lists are called includedChildDNames, includedLeftDNames, and includedRightDNames to reflect the type of relationship between the elements. For example, the following element view shows the Service Models root with one physical child named localMonitor. localMonitor has two source elements named END_USER_RESPONSE and HTTP.

Figure 2-7 *Operations Center Explorer Pane*



If the above <inclusionSpec> is used to query the Service Models element, the result set would include the following:

```
<elements>
  <item>
    <DName>root=Organizations</DName>
    <includedChildDNames>
      <item>org=localMonitor/...</item>
    </includedChildDNames>
    <includedLeftDNames />
    <includedRightDNames>
      <item>org=localMonitor/...</item>
    </includedRightDNames>
    :    :    :
  </item>
  <item>
    <DName>org=localMonitor/...</DName>
    <includedChildDNames />
    <includedLeftDNames>
      <item>root=Organizations</item>
    </includedLeftDNames>
    <includedRightDNames>
```

```
      <item>END_USER_RESPONSE=END_USER_RESPONSE/...</item>
      <item>HTTP=HTTP/...</item>
    </includedRightDNames>
    :    :    :
  </item>
  <item>
    <DName>END_USER_RESPONSE=END_USER_RESPONSE/...</DName>
    <includedChildDNames />
    <includedLeftDNames>
      <item>org=localMonitor/...</item>
    </includedLeftDNames>
    <includedRightDNames />
    :    :    :
  </item>
  <item>
    <DName>HTTP=HTTP/...</DName>
    <includedChildDNames />
    <includedLeftDNames>
      <item>org=localMonitor/...</item>
    </includedLeftDNames>
    <includedRightDNames />
    :    :    :
  </item>
</elements>
```

<searchSpec>

A `<searchSpec>` (WSDL type: `<ElementSearchSpec>`) is used in the findElements query in place of an `<inclusionSpec>`. With a `<searchSpec>`, a searchString field specifies which children of the query's base element should be returned. (Refer to "Using Find to Search for Elements" in the *Operations Center 5.6 User Guide* for instructions on creating a search string using the `Find` command's *Advanced* tab.) Thus, a findElements result set can include elements in any part of the base element's downward hierarchy, without including the elements in between. Queries using a `<searchSpec>` also do not populate the included *nnn* lists in the returned elements.

Another important distinction between a `<searchSpec>` and an `<inclusionSpec>` is that a `<searchSpec>` does not distinguish between logical and physical relationships. With a `<searchSpec>`, right-relation (source) elements are treated as if they are physical (name) children, for the purpose of scoping the search. With a `<searchSpec>`, the caller can retrieve the name children of a source element.

A sample `<searchSpec>`:

```
<searchSpec>
  <maxDepth>10</maxDepth>
  <maxRecords>50</maxRecords>
  <maxWaitSeconds>60</maxWaitSeconds>
  <searchString>(objectClass=HTTP)</searchString>
</searchSpec>
```

By using this `<searchSpec>` against the Service Models root shown in the previous section, the caller can retrieve all of the HTTP test elements downward in its hierarchy.

<contentSpec>

For all element queries, a `<contentSpec>` (WSDL type: `<ElementContentSpec>`) specifies the optional contents of the included elements that should be populated in the result set. Specific parts of an element have been made optional either because they were deemed unnecessary for many Web services clients (for example, icon descriptors), or because there is an implicit overhead cost to retrieving the information that should not be incurred unless the data is actually needed (for example, certain attribute values). Overhead for retrieved data is most severe when querying elements from a Operations Center InterConnection adapter.

2.3.3 Alarm Queries

The API includes the alarm query services listed in Table 2-4.

Table 2-4 *Alarm Query Services*

Name	Description
`getAlarms(session:WsSession, elementDName:string, channelName:String, throttle:ResponseThrottle):AlarmResultSet`	Retrieves alarms for an element in a real-time channel.
`getHistoricalAlarms(session:WsSession, elementDName:string, channelName:String, from:Date, to:Date, throttle:ResponseThrottle): AlarmResultSet`	Retrieves alarms for an element in a historical channel.
`getBreaches(session:WsSession, elementDName:string, slaName:String, objectiveName:String, from:Date, to:Date, throttle:ResponseThrottle): AlarmResultSet`	Retrieves SLA breach alarms for an element.
`getOutages(session:WsSession, elementDName:string, slaName:String, objectiveName:String, from:Date, to:Date, throttle:ResponseThrottle): AlarmResultSet`	Retrieves SLA outage alarms for an element.
`nextAlarms(session:WsSession,cursor:string, throttle:ResponseThrottle):AlarmResultSet`	Retrieves additional records from a previous query.

All alarm queries are scoped to a single element, specified by the elementDName argument. The channelName argument specifies the type of alarm to be retrieved. Valid channel names for a Operations Center server should be retrieved by using the getServerInfo service.

Date arguments are always required.

The getOutages and getBreaches services include optional slaName and objectiveName arguments. These can limit the alarms query to a specified SLA, or to a specified objective name within an SLA. If slaName is not specified, objectiveName is ignored.

All alarm queries are capable of retrieving multiple alarm records and support a throttle (see Section 2.3.1, “Throttle Mechanism,” on page 17 for more information).

To retrieve the undeliverable portion of an alarm query response, use the nextAlarms service.

2.3.4 Series Data Queries

The API includes a single service, listed in Table 2-5, for querying series data.

Table 2-5 *Series Data Queries*

Name	Description
`getSeriesData(session:WsSession, elementDName:string, seriesDescriptor:MosSeriesDescriptor, throttle:ResponseThrottle):SeriesDataResultSet`	Retrieves the specified series data for an element.

A series data query requires an elementDName and a valid <seriesDescriptor> (WSDL type: <MosSeriesDescriptor>) for that element. Series descriptors for an element are part of the optional data that can be returned with an element by setting the includeSeriesInfo flag in the <contentSpec> argument of an element query. Here are some sample series descriptors as returned by an element query:

```
<seriesDescriptors>
  <item>
   <expressionName>devtower18.mosol.com:6789.NTLM test on qaoracledb.HTTP.001-main
page qaoracledb.TransactionTime</expressionName>
    <from>1970-01-01T00:00:00.000Z</from>
    <profileName>ElementProfile</profileName>
    <to>2007-03-05T22:24:05.580Z</to>
    <type>TYPE_SIMPLE_FLOAT</type>
  </item>
  <item>
    <expressionName>Element Condition Change</expressionName>
    <from>2006-12-12T23:04:32.257Z</from>
    <profileName>Service Levels</profileName>
    <to>2007-03-05T22:24:05.580Z</to>
    <type>TYPE_CONDITION</type>
  </item>  :    :    :
```

When a series descriptor is embedded in an element, the From and To time stamps reflect the available data range for series. When returning a series descriptor to the server in a series data query, clients should update these date fields to specify the desired portion of the series.

The type field of the series descriptor is important for processing the response data. The primary payload of a series data result set is an array of DataPoint objects. Close examination of the WSDL reveals that DataPoint is an abstract type:

Figure 2-8 *The DataPoint Abstract Type*

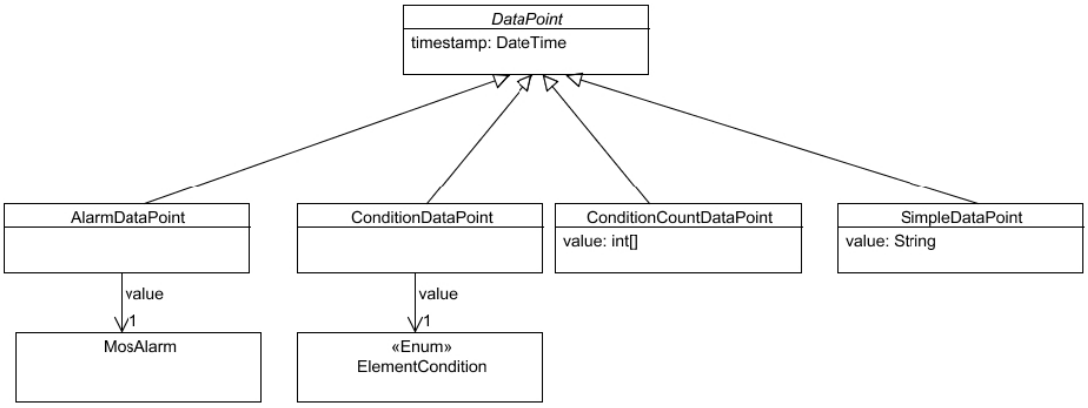


Table 2-6 lists the correlation between the type field in the series descriptor and the type of DataPoint objects returned in a series data query.

Table 2-6 *Correlation between Field and DataPoint Types*

Type	DataPoint Type
TYPE_SIMPLE_STRING	SimpleDataPoint
TYPE_SIMPLE_NUMERIC	SimpleDataPoint
TYPE_SIMPLE_FLOAT	SimpleDataPoint
TYPE_ALARM	AlarmDataPoint
TYPE_CONDITION	ConditionDataPoint
TYPE_CONDITION_COUNT	ConditionCountDataPoint

The series data query can retrieve multiple records and supports a throttle (for more information, see Section 2.3.1, “Throttle Mechanism,” on page 17).

To retrieve the undeliverable portion of a series data query response, use the nextSeriesData service as described in Table 2-7.

Table 2-7 *nextSeriesData Service*

Name	Description
`nextSeriesData(session:WsSession,cursor:string, throttle:ResponseThrottle):SeriesDataResultSet`	Retrieves additional records from a previous query.

2.3.5 Health Data Queries

The API includes a single service for querying SLA health data, as shown in Table 2-8.

Table 2-8 *Health Data Queries*

Name	Description
`getSlaHealthData(session:WsSession, elementDName:string, slaName:string, objectiveName:string, from:dateTime, to:dateTime, blockSize:IntervalBlock, throttle:ResponseThrottle):HealthDataResultSet`	Retrieves the SLA health data for an element.

The health data query is similar to the series data query and some of the alarm queries. The primary difference is that health data responses represent a state over a period of time. The length of time represented by each health data record in the result set is specified by blockSize argument, ranging from a minute to a month.

As with the getBreaches and getOutages queries, the slaName and objectiveName arguments are optional in this query. If they are not specified, the results represent the health of the element across all SLAs in which it participates.

The health data query can retrieve multiple records and supports a throttle (see Section 2.3.1, “Throttle Mechanism,” on page 17 for more information).

To retrieve the undeliverable portion of a health data query response, use the nextHealthData service as described in Table 2-9.

Table 2-9 *nextHeathData Service*

Name	Description
`nextHealthData(session:WsSession,cursor:string, throttle:ResponseThrottle):HealthDataResultSet`	Retrieves additional records from a previous query.

2.3.6 SLA Queries

The API includes a single service for querying SLAs, as shown in Table 2-10.

Table 2-10 *SLA Queries*

Name	Description
`getAllSlas(session:WsSession, throttle:ResponseThrottle):SlaResultSet`	Retrieves all SLAs defined on the server.

As the name implies, this query simply retrieves all defined SLAs. Throttling is available to manage response size. To retrieve subsequent portions of a SLA query, use nextSlas as shown in Table 2-11.

Table 2-11 *nextSlas*

Name	Description
`nextSlas(session:WsSession,cursor:string, throttle:ResponseThrottle):SlaResultSet`	Retrieves additional records from a previous query.

2.4 Miscellaneous Services

The API includes additional services that do not fit into any other category, as shown in Table 2-12.

Table 2-12 *Miscellaneous Services*

Named	Description
`getIcon (session:WsSession, iconDescriptor:MosIconDescriptor):MosIcon`	Retrieves a single icon.
`getIcons (session:WsSession, iconDescriptors:MosIconDescriptor[]):MosIcon[]`	Retrieves a list of icons.
`getServerInfo(session:WsSession):MosServerInfo`	Retrieves general server data.

An icon descriptor can be returned with elements and alarms. To retrieve the actual icon image, use one of the icon retrieval services. The icon descriptor includes the name, the file extension for example, `.gif`) and the size category (WSDL type: MosIconType) of the icon. Returned icons include the descriptor and a byte-stream representation of the image.

The getServerInfo service returns general information about the server, including start time and status, as well as other useful information like the internal names of alarm channels, and algorithms.

2.5 Error Reporting

The Operations Center server returns a MosFaultException for any fatal processing error. Table 2-13 describes the fault codes returned and their meaning.

Table 2-13 *Server Error Messages*

Name	Description
INTERNAL_ERROR	Indicates a problem with server configuration or code that prevented processing of the request. Check the formula.trc file for more information.
CREDENTIALS_ERR	Invalid user name or password in a login request.
SESSION_LIMIT_ERR	Too many active sessions already established for the user in this login attempt.
INVALID_SESSION	Session used in the request was not found. If it was previously active, it has already been closed and reaped.
SESSION_EXPIRED	Session used in the request has timed out.
NOT_FOUND	A required item to fulfill the request was not found; for example, the DName used for a query did not identify a known element.
NOT_ALLOWED	The session user does not have sufficient permissions to perform the requested action.
ALREADY_EXISTS	An attempt was made to create an item that already exists; for example, an attempt to create an element that already exists.
INVALID_REQUEST	Not used at this time.
NOT_IMPLEMENTED	Not used at this time.

3 Working with Elements

This section focuses on the API features that allow clients to write and remove elements, change element relationships, and modify element properties and attributes.

- Section 3.1, “Writing Elements,” on page 27
- Section 3.2, “Changing Element Relationships,” on page 28
- Section 3.3, “Changing Element Property Page Data,” on page 29

3.1 Writing Elements

The API provides services that allow clients to add, modify, and remove elements on the Operations Center server, as listed in Table 3-1.

Table 3-1 *Services for Writing Elements*

Name	Description
`createElement(session:WsSession, data:ElementCreateData):ElementWriteResponse`	Creates a new element on the Operations Center server.
`updateElement(session:WsSession, data:ElementUpdateData):ElementWriteResponse`	Updates an element on the Operations Center server.
`removeElement(session:WsSession, elementDName:string):void`	Removes an element from the Operations Center server.

Element write services are restricted from operating on any element with a DName ending with:

- `root=Administration`
- `root=Elements`
- `root=Service+Level+Agreements`

In addition, you cannot use removeElement on any root element.

The createElement and updateElement services allow the client to manipulate an element’s attributes, relationships, and property page data. These services are not transactional, so it is possible for some parts of the service request to succeed, while others fail. If either of these services returns an exception, then no updates are made. However, if any part of the operation succeeds, then an ElementWriteResponse object is returned.

An ElementWriteResponse contains an updated copy to the element, as well as a listing of any nonfatal errors that occurred. Each listed error includes an error code (WSDL type: MosFault), the type of operation that failed (WSDL type: ElementWriteType), and a qualifier identifying the name of the attribute, property, or relationship that could not be updated.

In the following example, the client attempts to create an element called “test” with an invalid property page called “foo”:

```
<createElement xmlns="http://wsapi.mosol.com">
  <session>
    :    :    :
  </session>
  <data>
    <DName>org=test/root=Organizations</DName>
    <addRightRelationships />
    <attributes />
    <propertyPageNames>
      <item>foo</item>
    </propertyPageNames>
    <propertyValues>
      <item>
        <fieldName>bar</fieldName>
        <fieldValue>someValue</fieldValue>
        <pageName>foo</pageName>
      </item>
    </propertyValues>
  </data>
</createElement>
```

The response to this request indicates that the element was created (because the response is not an exception). It also lists failures for adding the property page and setting the property value because no property page name “foo” could be found:

```
<createElementReturn>
  <element>
    <DName>org=test/root=Organizations</DName>
      :    :    :
  </element>
  <failures>
    <item>      <faultCode>NOT_FOUND</faultCode>
      <itemQualifier>foo</itemQualifier>
      <typeCode>ADD_PROPERTY_PAGE</typeCode>
    </item>
    <item>
      <faultCode>NOT_FOUND</faultCode>
      <itemQualifier>foo:bar</itemQualifier>
      <typeCode>SET_PROPTERY_VALUE</typeCode>
  </item>
  </failures>
</createElementReturn>
```

3.2 Changing Element Relationships

Managing element relationships with Web services works much the same way as the comparable functionality in the Operations Center console. As in the Operations Center console, you can only manipulate an element’s association with source elements, or “right” relations, as exposed in WSAPI. Through WSAPI you can only add or remove relations; you cannot modify an existing relation.

WSAPI provides five types of relations that can be configured:

Type	Description
RelationshipInfoStatic	Each instance of this type of relationship corresponds to an entry in the Elements tab shown in Figure 3-2.

Type	Description
RelationshipInfoRegex	An instance of this type corresponds to an entry on the *Matches* tab (with *Use LDAP Syntax* deselected) of an element's property page. An element can only support one relation of this type and this type cannot be used in conjunction with a RelationshipInfoLdap instance.
RelationshipInfoLdap	An instance of this type corresponds to an entry on the *Matches* tab (with *Use LDAP Syntax* selected) in the element's property page. An element can only support one relation of this type and this type cannot be used in conjunction with a RelationshipInfoRegex instance.
RelationshipInfoScript	An instance of this type corresponds to an entry on the *Script* tab in the element's property page. An element can only support one relation of this type.
RelationshipInfoClass	An instance of this type corresponds to an entry on the *Class* tab in the element's property page. An element can only support one relation of this type.

3.3 Changing Element Property Page Data

WSAPI allows clients to add and remove associations between an element and predefined metamodel property page templates. For any template associated with an element, WSAPI can be used to retrieve and set values for fields on that property page.

Property page support is limited. There is currently no way to query or update property page templates through the API, so validation rules for a property page field are not available to WSAPI clients. Some prior knowledge of the data entry requirements for a given template is needed. An attempt to set a field to an invalid value is reported in the *Failures* list in the result set.

Element attributes are the core properties of an element that control most aspects of an element's behavior. Some attribute functionality, such as element relationships, is exposed in WSAPI as higher level objects. Many element attributes are not available for update through the API.

Each element class defines a set of attributes that are common to all elements of that class. Element queries always return the names of the attributes supported by the element. Queries only return attribute objects (WSDL type: ElementAttribute) when they are explicitly requested in the element query contentSpec. Attribute objects contain not only the name and value (usually) of an attribute, but also identify its type (WSDL type: ElementAttributeType), and whether the attribute can be updated through Web services. For example:

```
<attributes>
  <item>
    <id>Condition</id>
    <type>COND</type>
    <value>OK</value>
    <writeable>false</writeable>
  </item>
  <item>
    <id>DisplaySourceElements</id>
    <type>BOOL</type>
    <value>true</value>
    <writeable>true</writeable>
  </item>
  :    :    :
</attributes>
```

When updating an element's attributes, the best practice is to first retrieve the current attribute object and return it to the server with the updated value. When creating an element, retrieve the attributes of another element of the same class. Some attributes also have special formatting rules or a limited set of understood values. Valid values for an attribute can vary by element class, server configuration, and possibly server version. It might be necessary to query elements with the desired configuration to derive the correct settings.

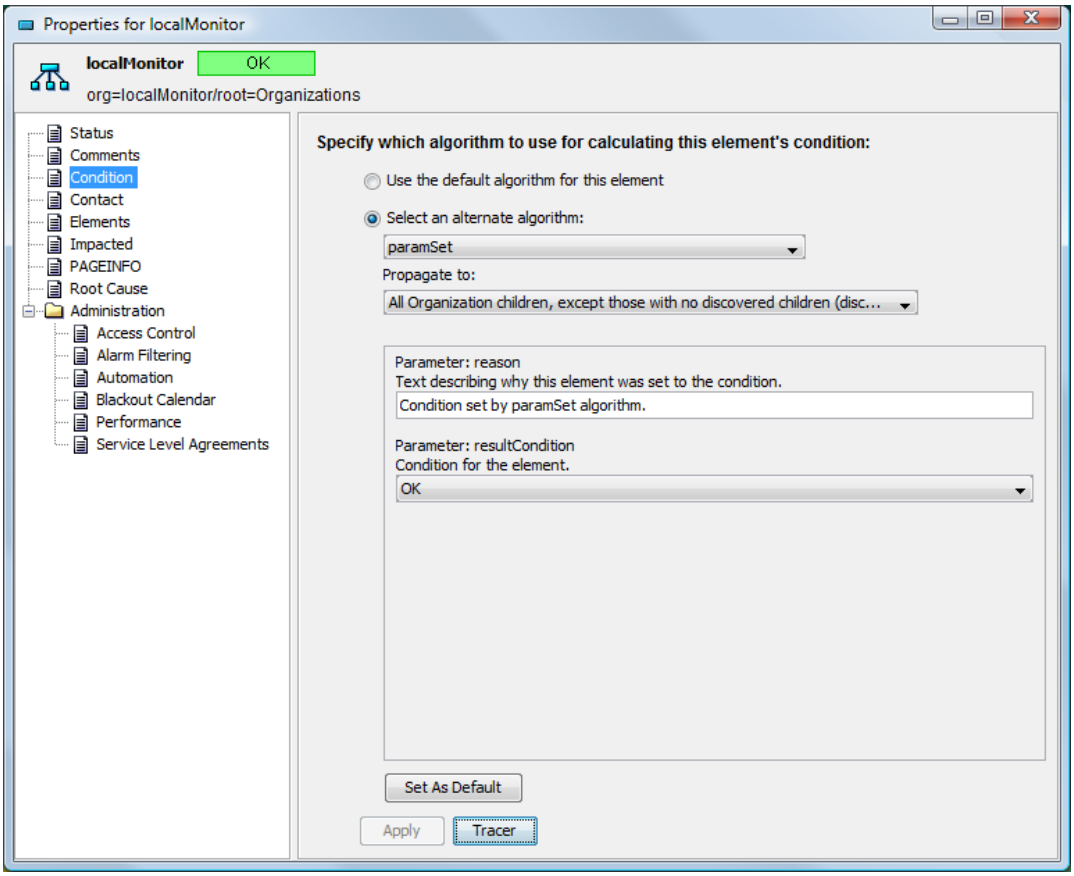
Some configuration options that can be manipulated by setting attributes through WSAPI display in an element's property pages, as shown in the following sections:

- Section 3.3.1, "Condition Property Page," on page 30
- Section 3.3.2, "Elements Property Page," on page 32
- Section 3.3.3, "Status Property Page," on page 33

3.3.1 Condition Property Page

In the Condition property page, specify an algorithm used to calculate an element's condition.

Figure 3-1 *Element Condition Property Page*



To specify the algorithm used to calculate an element's condition:

1. In th Explorer pane, right-click the element and select *Properties*.
2. Select *Condition*.
3. Select *Select an alternate algorithm*, then select the algorithm from the drop-down list. Then, set which objects the algorithm propagates to by selecting from the second drop-down list.

Setting	Description	Stored Attribute
Select an alternate algorithm	Uses a selected algorithm from a standard library.	Stored in the Algorithm attribute
Propagate to	Specifies how the algorithm is propagated: to no children, all children, all children except those with no children (might force discovery). or All children except those with no discovered children (discovery is not forced).	Stored in the *Algorithm Disseminates* attribute.
Parameter	Parameters vary by algorithm.	Stored in the *Algorithm Parameters* attribute.

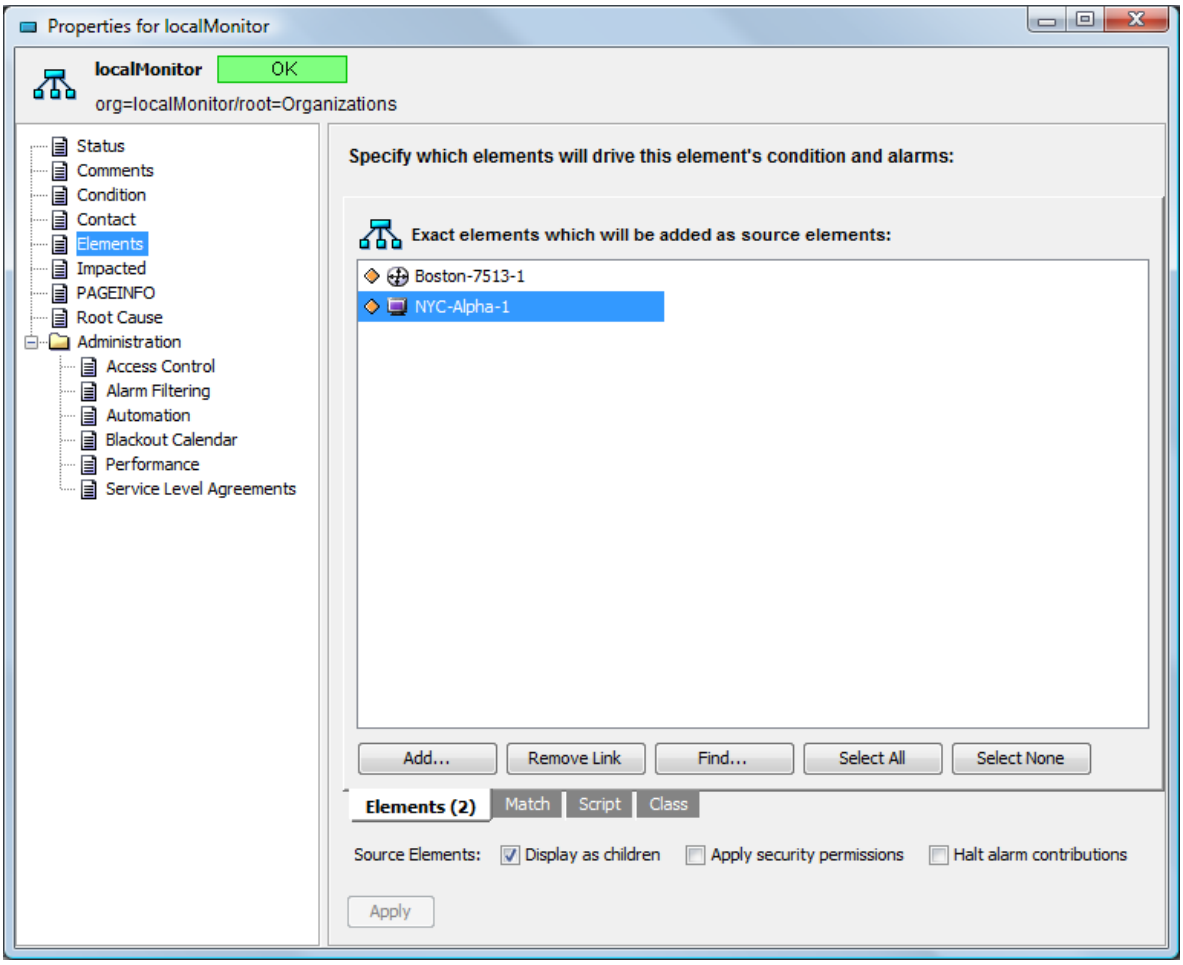
4 Set the algorithm parameters.

5 Click *Apply*.

3.3.2 Elements Property Page

The Elements property page determines state contribution or alarm propagation from a child service model.

Figure 3-2 *Element Elements Property Page*



To specify the elements to drive the element's condition and alarms:

1. In th Explorer pane, right-click the element and select *Properties*.
2. Select *Elements*.
3. Click *Add* to browse and select the elements to drive state and alarms.
4. Select from the following *Source Elements* options:

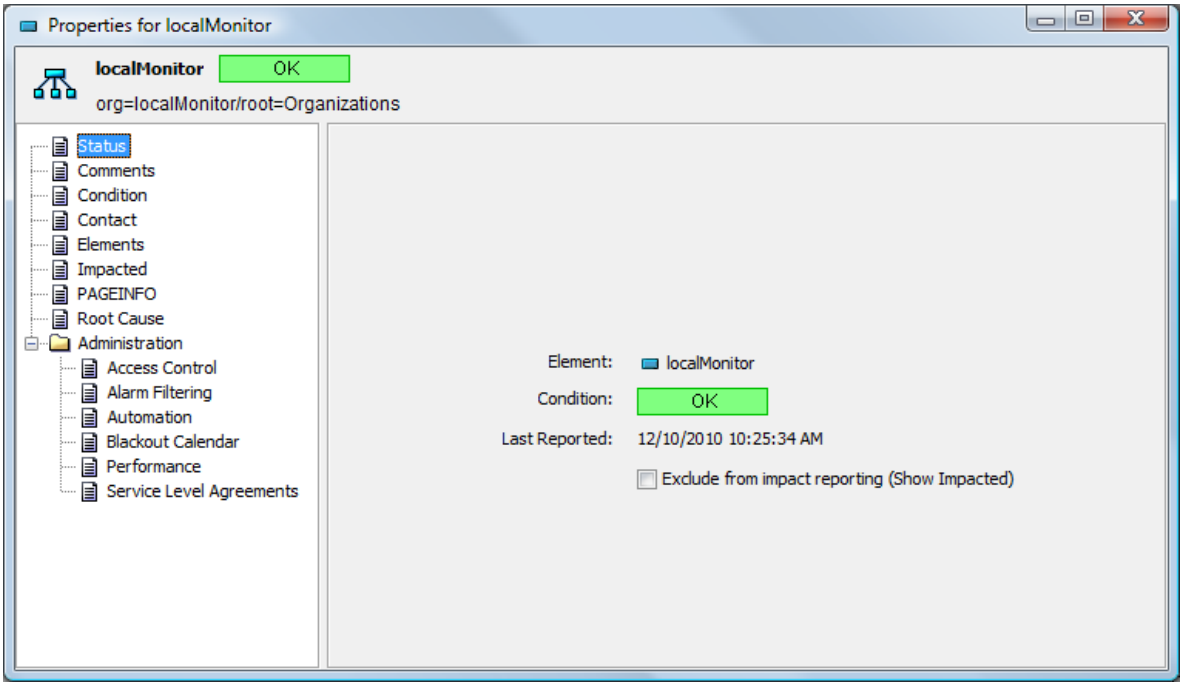
Setting	Description	Stored Attribute
Display as children	Displays the element created for the service model as a child of the source element used to create it.	Stored in the DisplaySourceElements attribute.
Apply security permissions	Applies the source element's security permissions to the element created for the service model.	Stored in the PropagateSecurity attribute.
Halt alarm contributions	Stops the propagation of alarms from the source element up to the service model.	Stored in the IgnoreChildAlarms attribute.

5 Click *Apply*.

3.3.3 Status Property Page

The *Status* property page displays the label, condition, and date and time of the last reported condition for the service model. Select the *Exclude from impact reporting (Show Impacted)* check box to exclude the element from impact reporting. This setting is stored in the *ExcludedFromImpact* attribute.

Figure 3-3 *Element Status Property Page*



4 Web Services Client Operation

The Operations Center Web Services Client (MosWsClient) is a command line utility that provides instant access to all available Web services capabilities. Developers who want to integrate their application with Operations Center can use this utility to become familiar with how Web services work, and to test various approaches to using Web services.

This section describes the advanced features included with the client (the basics of installing, configuring, and starting the client are covered in Chapter 1, “Getting Started,” on page 7.):

- Section 4.1, “Using a Properties File,” on page 35
- Section 4.2, “Meta-Commands,” on page 35
- Section 4.3, “Record/Playback,” on page 38

4.1 Using a Properties File

When MosWsClient is started, the default behavior is to look for the `wsclient.properties` file to establish the startup configuration. You can override this behavior by specifying an alternate properties file name as a command line argument. The client does not start if it cannot read a properties file.

4.2 Meta-Commands

Meta-commands are instructions to the client engine that can be entered at any data entry prompt. Meta-commands are client plug-ins that are configured in the properties file. Developers can add their own meta-commands or change the key sequences used to invoke meta-commands. The default configuration, as described in this document, uses a single character prefix followed by a single letter to specify a meta-command, but there is no restriction limiting which characters or how many characters are used to invoke a meta-command. Meta-command key sequences are not case sensitive.

Invocation of a meta-command is recognized whenever the data entry begins with the configured meta-command prefix (wsapi.metacommand.prefix). The default prefix is a forward slash (/). This document uses the default prefix in all meta-command descriptions. (UNIX users might want to change this default to avoid misinterpretation when entering file paths.)

With the exception of the CancelProcessor, all core meta-commands return control to the same prompt and state present when the command was invoked:

- Section 4.2.1, “Simple Meta-Commands,” on page 36
- Section 4.2.2, “Post Processor Meta-Commands,” on page 36

4.2.1 Simple Meta-Commands

Simple meta-commands are stateless operations that are always active and always behave in a consistent manner. The base implementation includes simple meta-commands, as listed in Table 4-1.

Table 4-1 *Simple Meta-Commands*

Command	Default Key Sequence	Function
`CommandHelpProcessor`	/h	Lists all current meta-commands with a description and class name for each. Post processor meta-commands (see Section 4.2.2, “Post Processor Meta-Commands,” on page 36) can be enabled and disabled, so the listed description changes based on the current state.
`CancelProcessor`	/x	Cancels the current activity. From the main menu (or any prompt before the main menu), this command exits the client. For any prompt from the main menu, this command cancels the activity and returns to the main menu.
`PlaybackProcessor`	/p	Plays a recorded script. Refer to Section 4.3, “Record/Playback,” on page 38.

4.2.2 Post Processor Meta-Commands

Post processors are used to analyze the results of a Web service invocation. They have a post processor interface that is named after each Web service invocation and a meta-command interface used to enable or disable the post-processors function.

Post processors are invoked in the order they are enabled. If you want to see the SOAP messages before deciding to save them, enable ShowSoapPostProc before enabling SaveSoapPostProc. More information about post processors is available in Section 5.4, “MetacommandProcessor and PostProcessor,” on page 45. The base implementation includes the post processor commands listed in Table 4-2.

Table 4-2 *Post Processor Meta-Commands*

Command	Default Key Sequence	Function
`ShowSoapPostProc`	/d	When enabled, displays SOAP request and reply messages.
`SaveSoapPostProc`	/f	When enabled, prompts users to save SOAP requests and replies to a file. It can be helpful to use an XML viewer to look the SOAP messages (for example, a Web browser).

Command	Default Key Sequence	Function
`CacheBuilderPostProc`	/c	When enabled, the cache builder examines the content of SOAP messages for data that can be used to populate options in the UI. For example, when you select the Update element service from the main menu and the first requested parameter is the DName of the element to update, the UI displays a list of all DNames that were cached from previous queries, allowing the user to select a DName instead of typing one. Cached data is context sensitive, so when you invoke the nextAlarms service, the UI displays a list of cursors from previous AlarmResultSets and does not display cursors for other types of queries. Disabling this post processor only stops updates to the cache; it does not disable or remove data already in the cache.
`TestScriptRecorder`	/r	When enabled, this processor first prompts the user for a file name in which it records subsequent SOAP exchanges. All request and reply data is stored in the specified file until the post processor is disabled. Terminating the client prior to disabling the recorder might result in an invalid recording. Refer to Section 4.3, “Record/Playback,” on page 38 for more information on this feature.

4.3 Record/Playback

The record/playback functionality allows you to record a series of Web service calls and play them back at a future time. To start the recorder, enter `/r` at any data entry prompt. The recorder prompts for a file name in which subsequent calls are recorded and then it returns to the data entry prompt.

Until the recorder is stopped, Web service request and reply data is stored as an XML stream to the specified file. You can manually edit these files, but you should be careful to avoid breaking the file structure. For example, this is a recording of a getElement request:

```
<object-stream>
  <com.netiq.wsapi.client.services.interactive.IAGetElementSvc>
    <__session>
      :    :    :
    </__session>
<__dname> root=Elements</__dname>
    <__inclusionSpec>
      <childDepth>0</childDepth>
      <forceDiscovery>false</forceDiscovery>
      <includeLeftElements>false</includeLeftElements>
      <includeRightElements>false</includeRightElements>
      <____hashCodeCalc>false</____hashCodeCalc>
    </__inclusionSpec>
    <__contentSpec>
      <includeIconInfo>false</includeIconInfo>
      <includeOperationInfo>false</includeOperationInfo>
      <includePropertyValues>false</includePropertyValues>
      <includeSeriesInfo>false</includeSeriesInfo>
      <includeSlas>false</includeSlas>
      <optionalAttributeNames/>
      <____hashCodeCalc>false</____hashCodeCalc>
    </__contentSpec>
    <__throttle>
      <maxRecords>0</maxRecords>
      <____hashCodeCalc>false</____hashCodeCalc>
    </__throttle>
```

You can safely change the content of a field such as `<__dname>` or `<maxRecords>`, but adding or removing other fields might cause deserialization errors during playback.

It is important to understand that the recorder looks only at Web service calls, not the user interactions that produced the call. If the recorder is running when you play back a script, the calls generated from the script player are also recorded in the current script file. You cannot play back a script that is currently being recorded.

After starting the script player, you are prompted to enter the script file name. If the player can find and read the specified file, the player issues the Web service calls recorded in the script file, sending each request (almost) verbatim. The script player does not use WsSession objects from the recording. Instead it uses session information from the current context.

When you play a recorded script, the script player inherits a copy of the current context (current session, if any, current post processor settings, etc.). Any changes to the context, such as opening a new user session, are only applied to the context copy and are lost when the playback is complete.

The only other case where a playback request differs from the recorded script is the cursor value when you play back a next-type request (nextAlarms, nextElements, etc.). Because cursor values are session-specific, sending a cursor value recorded in another session always fails. Instead, these services use the last cursor value returned for that type of query.

This feature depends on CacheBuilderPostProc being enabled. If it is disabled, the playback either returns the latest appropriate entry in the cache, or if there is none, it uses the recorded cursor value. In either case, the playback probably does not produce the expected results.

During playback, response data in the recording is compared with the playback response, and a detailed analysis is written to the display and a log file (*scriptFileName*.log). Here is a sample playback analysis of the script shown above:

```
Service: Select from menu (1 needed): /p
Enter file name: get.scr
= 1-getElement.elementResultSet.element[Elements(1)].dName:root=Elements
= 1-getElement.elementResultSet.element[Elements(1)].className:root
= 1-getElement.elementResultSet.element[Elements(1)].condition:CRITICAL
= 1-getElement.elementResultSet.element[Elements(1)].displayName:Elements
= 1-getElement.elementResultSet.element[Elements(1)].notes
= 1-getElement.elementResultSet.element[Elements(1)].parentDName
! 1-getElement.elementResultSet.element[Elements(1)].lastUpdate:74/15:1:0->74/
15:23:34
! 1-getElement.elementResultSet.element[Elements(1)].tempId:1173976147->1173990162
= 1-getElement.elementResultSet.element[Elements(1)].attrName[Element]
= 1-getElement.elementResultSet.element[Elements(1)].attrName[Condition]
= 1-getElement.elementResultSet.element[Elements(1)].attrName[Last Reported]
= 1-getElement.elementResultSet.element[Elements(1)].attrName[Algorithm]
= 1-getElement.elementResultSet.element[Elements(1)].attrName[Algorithm
Parameters]
= 1-getElement.elementResultSet.element[Elements(1)].attrName[Algorithm
Disseminates]
= 1-getElement.elementResultSet.element[Elements(1)].attrName[Root Cause]
= 1-getElement.elementResultSet.element[Elements(1)].attrName[Root Cause Tree]
= 1-getElement.elementResultSet.element[Elements(1)].attrName[Impacted]
- 1-getElement.elementResultSet.element[Elements(1)].
leftrelationshipInfo[STATIC:test(1)].relatedDName:org=test/
root=Generational+Models/root=Services
+ 1-getElement.elementResultSet.element[Elements(1)].leftrelationshipInfo
[STATIC:global(1)].relatedDName:org=global/root=Organizations
= 1-getElement.elementResultSet.cursor
= 1-getElement.elementResultSet.remainingElementCount:0
Playback of get.scr complete
```

Each line of the analysis corresponds to a single data value in the response. The first character of each line indicates the analysis results, as described in Table 4-3.

Table 4-3 *First Line Characters*

Character	Indicates…
=	The value has not changed.
!	The value has changed. When this indicator is used, both the recorded value and the playback value are shown separated by `->` (hyphen followed by greater than symbol).
+	A value in a list (an array) was present in the playback, but not in the recording.
-	The value in a list was present in the recording but not in the playback.

To analyze the meaning of the values in a line, look at the following example:

```
= 1-getElement.elementResultSet.element[Elements(1)].condition:CRITICAL
```

Table 4-4 describes the line characters used in this example.

Table 4-4 *Line Characters*

Sample Characters	Meaning
=	Indicates the analysis results. An equal sign = means the value is unchanged.
1	A number indicating sequence number of the message in the script being played. The number 1 identifies the first message in the script.
getElement.elementResultSet	The request type. This example looks at the result set from a getElement query.
.element[Elements(1)]	The response field. This example specifies the element field in the elementResultSet. Since element is actually an array of elements, the name is subscripted with square brackets [ ]. The value inside the brackets varies by the array type. For elements, it is the displayName of the element. However, because displayName is not unique (and DName is too long to produce readable output) the displayName is subscripted with parentheses ( ) containing the instance number of that displayName in this list.
.condition:	Specifies the field name within the element.
CRITICAL	Specifies the value of the field above.

5 Extending and Embedding the Web Services Client

The installed Operations Center Web Services Client (MosWsClient) serves as a reference client implementation as well as a test and learning tool. Full Java source code and (ANT) build scripts for the client are included in the client installation. You are free to use the reference client code as an example, embed the code directly into your client implementation, or add features to the client as needed.

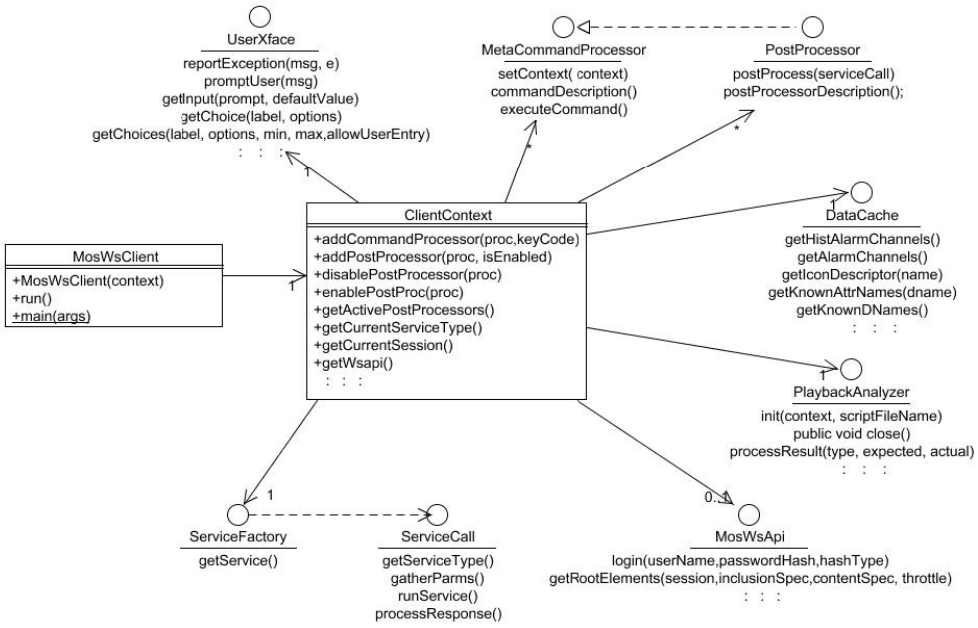
This section provides an overview of the client architecture and details some of the key extension points:

- Section 5.1, “ClientContext,” on page 42
- Section 5.2, “MosWsClient Class,” on page 43
- Section 5.3, “ServiceFactory and ServiceCall,” on page 44
- Section 5.4, “MetacommandProcessor and PostProcessor,” on page 45

5.1 ClientContext

At a high level, MosWsClient is comprised of several components defined by interfaces. A ClientContext object acts as a registry of active components and serves as the primary means by which all components interact. With a few exceptions, the individual components are loosely coupled, facilitating the quick addition of new implementations of various components by simply adding them to the context. The following figure illustrates the ClientContext object functionality.

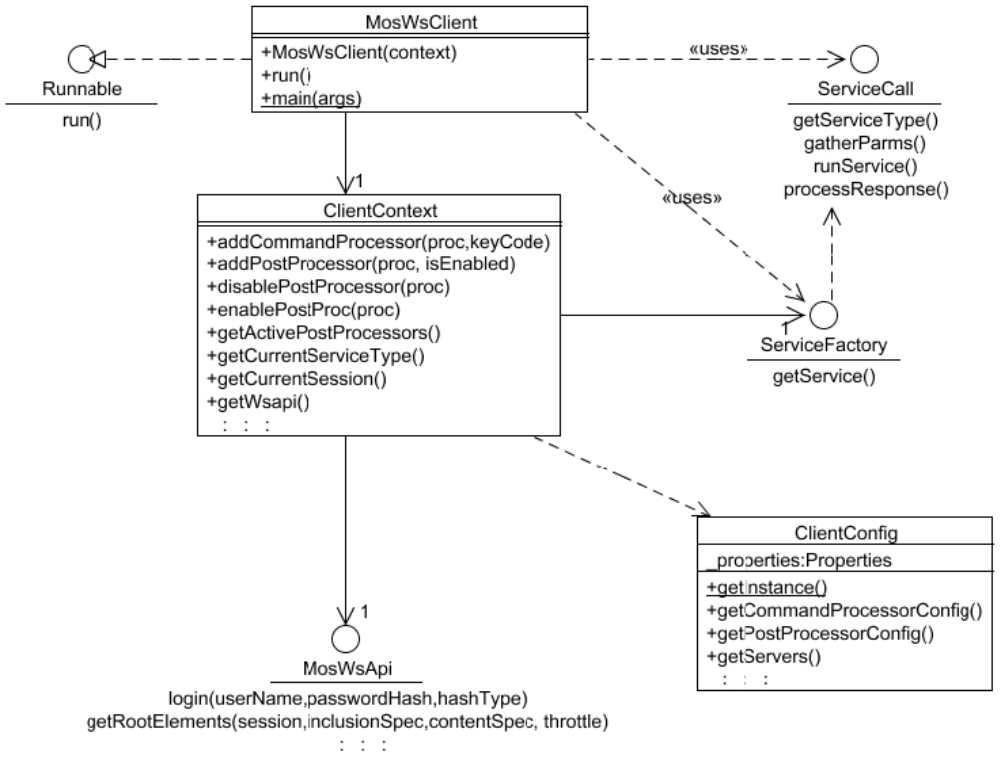
Figure 5-1 *ClientContext Object*



5.2 MosWsClient Class

The MosWsClient class is the primary entry point for the client. When invoked from the command line, it provides the main() method implementation to start up the client. When embedded in another application, MosWsClient operates as a Runnable and should be run in its own thread.

Figure 5-2 *MosWsClient Class*



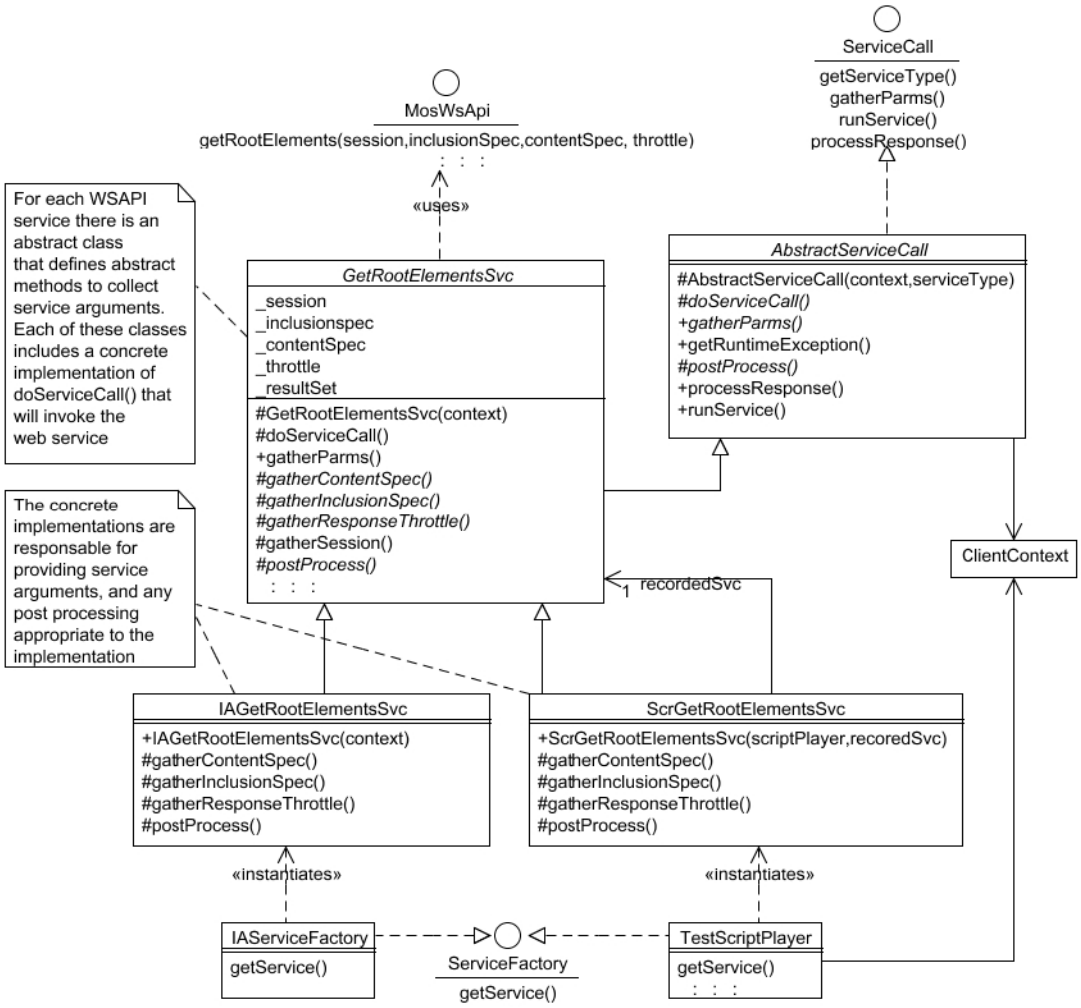
The run() method implements the primary run logic for the client as shown in the following code snippet:

```
          while (_running)
            {
                ServiceFactory svcFactory = _context.getServiceFactory();
                _context.setCurrentServiceType(null);

                ServiceCall service = svcFactory.getService();
                _context.setCurrentServiceType(service.getServiceType());

                try
                {
                    service.gatherParms();
                    service.runService();
                    service.processResponse();
                }
                catch (Abort e)
                {
                    _log.info("Aborted execution of " +
service.getServiceType().name());
                }
            }
```

5.3 ServiceFactory and ServiceCall

The following diagram shows the type hierarchy for two distinct implementations of service calls to invoke the getRootElements WSAPI service. IAGetRootElementsSvc is an interactive implementation, prompting the user to enter data from which it generates the appropriate arguments. ScrGetRootElements is the scripted implementation of the same service. It uses a reference to a previously executed GetRootElements service call to provide arguments to the new service call invocation. The abstract classes between these two implementation classes and the ServiceCall interface provide a common framework for ServiceCall implementations. The AbstractServiceCall class is the common base for all service call implementations. It provides access to the client context, and implements common functionality, such as invoking post-processors.

Figure 5-3 *ServiceFactory and ServiceCall*



The GetRootElementsSvc class is specific in the getRootElements WSAPI service. There are comparable classes for each WSAPI service (such as LoginSvc or CreateElementSvc). All of these service-specific classes follow the same model:

- They implement gatherParms() by calling abstract gatherArgument methods that collect the service arguments from the concrete class.

- They provide a concrete implementation of doServiceCall() that invokes the Web service and stores a reference to the response.
- They delegate argument collection and post processing to the concrete implementation.

The two ServiceFactory implementations shipped with MosWsClient are shown at the bottom of the diagram:

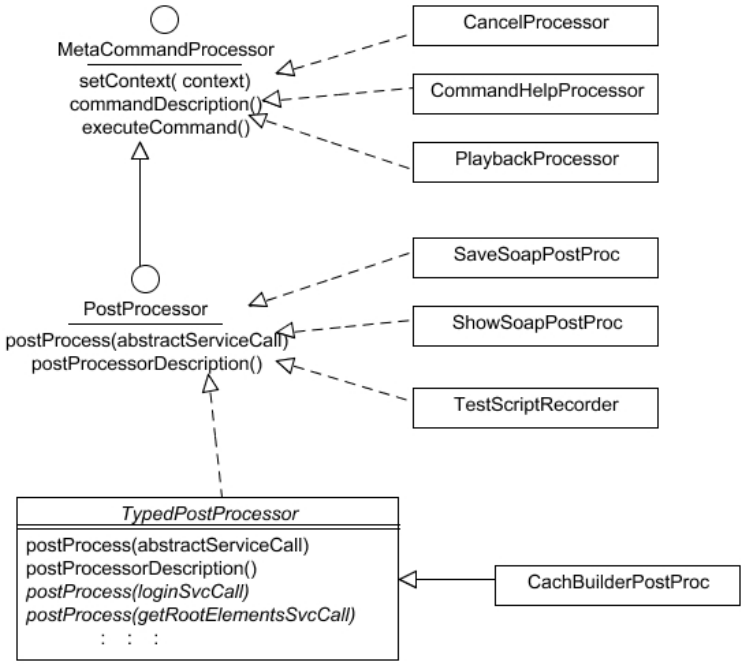
- **IAServiceFactory:** Used for interactive service invocations. The getService() method shows the main menu, and based on the user selection, it instantiates and returns the appropriate interactive ServiceCall implementation.
- **TestScriptPlayer:** Used for playing recorded scripts. The getService() method reads the next service invocation from the script file and based on its type, it instantiates the appropriate scripted ServiceCall implementation.

New ServiceCall and ServiceFactory implementations are needed to embed the client in another application. ServiceCall implementations should extend the appropriate service-specific class in order to be compatible with other features such as record/playback.

5.4 MetacommandProcessor and PostProcessor

Adding meta-commands and post processors is largely a matter of implementing the appropriate interface and making the context aware of the processor instance. This can be done by adding the processor in the properties file, or by adding it programmatically with the appropriate ClientContext method. PosProcessor.postProcess() takes an AbstractServiceCall argument instead of a ServiceCall. Also, the TypePostProcessor class delegates further to service-type specific methods.

Figure 5-4 *MetacommandProcessor and PostProcessor*



A WSAPI 1.1

The interface point for WSAPI 1.1 is:

`http://MOServername:WebServicePort/wsapi/services/Moswsapi_1_1`

The WSDL can be viewed at:

`http://MOServername:WebServicePort/wsapi/services/Moswsapi_1_1?wsdl`

The Web services reference client has been updated to automatically include the 1.1 service calls when a 1.1 connection point is accessed.

WSAPI 1.1 extends WSAPI 1.0 with three new service calls designed to facilitate the management of large services models through Web services, as described in Table A-1.

Table A-1 *New Service Calls*

Name	Description
`createElements(session:WsSession, transactionBlockSize:int, data:ElementCreateData[]):ElementsWriteResponse`	Creates new elements on the Operations Center server.
`updateElements(session:WsSession, , transactionBlockSize:int, data:ElementUpdateData[]):ElementsWriteResponse`	Updates elements on the Operations Center server.
`removeElement(session:WsSession, , transactionBlockSize:int, elementDName:string[]):void`	Removes elements from the Operations Center server.

These new service calls can dramatically improve the performance of large write operations by:

- Reducing the amount of communications overhead required when performing multiple write operations. In addition to allowing multiple write operations in a single Web service call, the returned ElementsWriteResponse does not include copies of the resulting elements.
- Reducing the number of round-trip operations to the Operations Center persistent store (also called the Config Store) by using the transactionBlockSize argument to wrap these operations in transactions. The transactionBlockSize argument specifies the number of element creates/updates/deletes to wrap in a single transaction.

  Specifying a transactionBlockSize of 0 disables the wrapping transaction and degrades performance. A transactionBlockSize of less than 0 (such as -1) wraps all write operations in the service call in a single transaction, potentially improving performance.

Although increasing the transactionBlockSize improves the performance of the write operations, it also incurs additional risk in the event of an error. When an error occurs, all write operations in the transaction are rolled back from the persistent store. However, this rollback does not currently impact the in-memory element model. Elements that were created in a rolled-back transaction still appear in the elements tree and must be manually removed, or removed when the server is restarted.