SIENA XML INTERFACE

USER MANUAL

Table Of Contents

Table of Figures

Chapter 1

RULES

Creating a Rule

Rule definitions are stored in files on the local file system to the client. The files can contain any number of rules as long as they follow the basic structure of a rule. The basic structure of a rule consists of four parts: The declaration, the tag specifier, the condition, and the action.

rule: *tag-specifier condition action*

The declaration is simply the word "rule" followed by a colon. This indicates the beginning of a new rule. The tag specifier indicates the XML tag the rule is to be applied to. The condition specifies any constraints that the XML tag must meet before the rule is applied to it. The action indicates what is to be done with the XML tag. Each of these parts is discussed in further detail in the following sections.

Tag Specifiers

The tag specifier is an identifier followed by an optional attribute identifier. An identifier is a string that starts with a character or underscore and is followed by zero or more characters, numbers, or underscores. Here are some examples of a tag specifier:

- annotation: Matches only annotation XML tags.
- item@partNum: Matches item XML tags that have an attribute named partNum. If an item XML tag exists that does not have this attribute the rule would not be applied to it since it is not the target of the rule.
- any: Matches any tag.

- any@type: Matches any XML tag that has an attribute named type.
- any@any: Matches any attribute in any tag.

The word "any" can be used in the tag specifier to indicate that any XML tag or any attribute would work for the rule. As seen in the examples the "any" keyword can be used as the tag identifier, the attribute identifier, or as both.

Conditions

The condition is a test to be applied to the XML tag that matches the name specified in the tag specifier. The condition has a set of operators that are applied to the XML tag, such as numerical comparisons and tests to see if the XML tag or attribute is present or not. A list of the operators and a brief description of what they do is below. Only when all of the conditions are met is the rule applied to the XML tag.

= *value* – This condition compares the value of the XML node to *value*. Only if the XML Tag value and the value specified are equal does the action get applied to the XML tag. The value can be a quoted string, a number, or a tree axis followed by a tag specifier.

!= *value* - This condition compares the value of the XML node to *value*. Only if the XML Tag value and the value specified are not equal does the action get applied to the XML tag. The value can be a quoted string, a number, or a tree axis followed by a tag specifier.

[<,<=] *value* – This condition compares the value of the XML tag to *value*. If the tag value is less than or less than or equal to the *value* then action is applied to the XML tag. The *value* can only be some type of numerical value.

[>,>=] *value* - This condition compares the value of the XML tag to *value*. If the tag value is greater than or greater than or equal to the *value* then action is applied to the XML tag. The *value* can only be some type of numerical value.

present – This checks to see if the XML tag or attribute is present. If so the action of the rules is applied to the XML tag.

notpresent – This condition checks to see if the XML tag or attribute is not present. If it is not, then the rule is applied to the XML Tag. This condition really only works with attributes and checking to see if a XML tag does not have a particular attribute.

If the condition requests a test to be done on something that does not exist, i.e. testing the value of the name attribute except the current XML tag does not have a name attribute, the condition will always fail. This is shown in figure 1.

Conditions can be grouped together in a rule by using a Boolean operator. The Boolean operations supported are "and" and "or". When using a Boolean operator the conditions that it works upon must be enclosed in braces. All of the sub-conditions can only be applied to the attributes of the tag and nothing else. Therefore it is not possible to check to see if a XML tag has a specific value and if it has an attribute that contains another value. The "and" operator imposes the condition that all of the sub-conditions in the rule must be satisfied by the XML tag before the action is applied to it. The "or" condition simply states that one or more of the sub-conditions must be satisfied before the action is applied to the XML tag. Examples of the Boolean operators are given in figure 1.

```
rule: attribute and {
  @maxOccurs = 1,
  @minOccurs = 1,
  @fixed notpresent,
  @use = "required"
} unique:currentTag

This rule would apply to this tag:
<example minOccurs="1" maxOccurs="1" use="requied">
but not this one:
<example minOccurs="1" use="required">
because the attribute maxOccurs is not present and therefore untestable
by the rule.
```

```
rule: attribute and {
  @maxOccurs notpresent,
  @minOccurs notpresent,
  @fixed notpresent,
  @use = "required"
} unique:currentTag

rule: any@type or {
  @type = “xsd:string”
  @type = “xsd:date”
} IGNORE

The following XML tags would satisfy the rule:
<example type=”xsd:string”>
<example type=”xsd:date”>
This tag would not though
<example type=”myType”>
```

Figure 1: Example of rules using the Boolean operators.

Actions

The last part of a rule definition that must be specified is the action. An action is applied only when an XML tag satisfies all the conditions of the rule. The action states where to store the value of the XML tag, or attribute, in the SIENA notification or a processing instruction that guides the program when translating the XML notification into a SIENA notification. When specifying the name of a SIENA attribute a valid identifier is all that is needed. If multiple tags have the same action then the values are stored in a coma-separated list in the SIENA notification attribute. The following list contains all of the valid processing instructions along with a description of what they do.

- ignore this will ignore the tag and not include it in the notification.
- group:*name* this states that multiple instances of this tag should be grouped together under the attribute named *name*.
- path this action indicates that the XML tag should have its named added to a prefix that is used to generate the names of the SIENA attributes.

- unique:target this specifies that this tag or attribute can be used to uniquely identify the tag specified by the target. Target is a tag specifier.

When the program finds a rule that contains the processing instruction "ignore" it will skip over the XML tag any of its children tags. For attributes the program simply skips over the attribute and continues to process the remaining attributes.

```
rule: ignoredTag present ignore
<example>
<ignoredTag attr="nothing">
<child1/>
<child2/>
</ignoredTag>
<nexTagInLine/>
</example>
```

Figure 2: XML Example and ignore action

In figure 2 an example XML notification is given with a rule containing an ignore processing instruction. When the program applies the rule to the ignoredTag in the XML it will skip the processing of the attributes of the tag and its children and resume processing with the nexTagInLine XML tag. The XML tags child1 and child2 are not processed nor the attribute "attr" in the ignoredTag.

The group instruction specifies that the value in the XML tag should be grouped together with other values in the attribute with the name *name*. When there is more than one value to be stored in the SIENA attribute then each value is separated by a coma.

The path instruction will cause the program to append the name of the XML Tag that matches the rule to the name of the SIENA attribute. Figure 3 shows an example of an XML notification along with the rule containing the path processing instruction and the resulting SIENA notification.

```
RULE:
rule: pathTag present path
```

```
XML NOTIFICATION:
<example>
<pathTag>
<child1> child1 </child1>
<child2> child2 </child2>
</pathTag>
<otherTag>
<child1>child1</child1>
</otherTag>
</example>

SIENA NOTIFICATION:
pathTag/child1=”child1”
pathTag/child2=”child2”
child1=”child1”
```

Figure 3: path processing instruction example

The unique processing instruction informs the program how to uniquely identify an XML tag from its siblings that have the same tag name in the XML notification. The program will append the value of the tag, or attribute, to the SIENA attribute, similar to what the path processing instruction does. Figure 4 shows an example of the unique processing instruction and the resulting SIENA notification.

```
RULE:
rule: tag@uniqueAttr present unique:tag


XML NOTIFICATION:
<example>
<tag uniqueAttr=”u1”>
<child1> child1 </child1>
<child2> child2 </child2>
</tag>
<tag uniqueAttr=”u2”>
<child1> child1 </child1>
<child2> child2 </child2>
</tag>
</otherTag>
</example>



SIENA NOTIFICATION:
tag/u1/child1=”child1”
tag/u1/child2=”child2”
```

```
tag/u2/child1="child1"
tag/u2/child2="child2"
```
Figure 4: unique processing instruction example

In the action the keyword "currentTag" can be used to indicate to the program that whatever the current name of the XML tag it is examining should replace the word "currentTag" in the action. When processing an XSchema file the program treats the "currentTag" keyword a little different. During the processing of the file the program keeps track of the element and complex definitions so that when the "currentTag" keyword is used the XSchema tag is not used but instead the current XML tag being defined by the XSchema. An example of this is in the default rules:

rule: any present group: currentTag

When processing an XSchema file with this rule the program will determine what the current XML tag being defined is and use the name of the XML tag as the name of the SIENA attribute to store its value in. The only real use for this keyword is in rules that specify "any" as the tag specifier. Once the action has been specified then the rule definition is finished and another rule can be started.

Submitting Rules to the Program

There are two ways that the program is able to get rules. The first method is through the set of default rules. These rules are meant to be very generic so that they can adapt to any situation and provide a basic framework to translate the XML notification into a SIENA notification. It is recommended that these rules not be altered. The second method of giving the program a set of rules is by using the user rules interface. This interface allows an application to give the program a set of rules, defined in a file, to be used when translating the XML notification. If there is a conflict between a default rule and a "user" rule then a warning message will be logged stating the conflict. The program will then use the "user" rule instead of the default

rule. More information about the user rules interface is contained in a later portion of this document.

Rule Conflicts

It is permissible for rules to have the same tag specifier, same actions and even the same conditions. When there are multiple rules that have the same tag specifier there is a possibility that an XML tag will satisfy the conditions for all the rules that have the same tag specifier. The type of rule that is being processed determines the outcome of this situation. If the rule is one of the default rules then the *first* rule to match is the one used for the XML tag. If the rule is one that was specified by a user then a warning message would be logged listing the rules that could match the XML tag and the *last* rule to match is applied to the XML tag. The warning message is only logged if the matching rules produce different actions or results when applied to the XML tag.

Chapter 2

SIENA'S XML INTERFACE

Introduction

There are two ways to use the XML client for SIENA. One way is to use the XMLClient object as the main interface to SIENA. The second way is to use the XML client package classes directly, such as the XMLProcessor and the XMLSubscriptionHandler. The second way of using the client requires more knowledge of SIENA than the first and should be used only by applications that may need to monitor the data flowing between the SIENA client and server closely. The first method is described first.

Using the XMLClient Class

The XMLClient object provides a XML interface to the SIENA servers. The application client only needs to know XML and very little about the underlying SIENA data structures. When an XMLClient object is created the name of a SIENA server is passed to it in the form of "senp: host.name: port". The host name is the name of the server where the SIENA server resides. The port is the port the SIENA server is listening on. The constructor will create a connection to the SIENA server if it can. If a server cannot be connected to then the constructor will throw a siena.InvalidSenderException.

The next step of setting up the XMLClient is to tell it where the file containing the XSchema for the XML notification is located. Before sending or receiving XML notifications this must be done otherwise the XML Client will not be able to correctly translate the notifications and subscriptions into their SIENA equivalents. The setStyleSheet method is used for this purpose. The method takes a single string argument which should contain the name, and if necessary the path, to the XSchema file. The method will then process the contents of the file. If there is

an error during processing a siena.SienaException is thrown containing information about the error that occurred.

To publish a XML notification to SIENA the publish method in the XMLClient object is invoked. This method takes a single parameter, a string containing the XML notification. The method will then translate the XML notification and send it to the SIENA server for publishing. If an error occurs during this process a siena.SienaException is thrown containing the error message. This is all that needs to be done to publish a XML notification.

To subscribe to XML notifications the method subscribe is called in the XMLClient object. The subscribe method has two parameters: a string containing the XPath expression that is the subscription language for the XML notification and a siena.Notifiable object. The interface siena.Notifiable has two methods that are called when a XML notification matches a subscription. The application client needs to fill in these methods with the code to process the XML notifications. The notify methods both receive a siean.Notification object. This object will have a single attribute in it called "xml". This attribute will be a string containing the complete XML Notification. See figure 5 at the end of this appendix for an example of how to extract the XML notification from the SIENA notification. This is all that needs to be done to subscribe to XML notifications. Multiple subscriptions can be issued using the same method. The XMLClient keeps track of the different subscriptions and objects that are to be notified when a particular subscription is fulfilled.

If there are user rules that the program should know about then the method setUserRules is invoked, passing it the name of the file that contains the rule definitions. If the rules cannot be processed then a siena.SienaException is thrown with the message containing information about the error that occurred. Otherwise the user rules will be used on the subsequent translation method calls. Note that this method can be called multiple times and the user rules will be combined together into one set. If two rules have the same tag specifier then the rule

that was added last is the one that is used. To clear the user rules the method clearUserRules can be called. This will clear all user rules that have been given to the program.

Using the XMLProcessor and XMLSubscriptionHandler Classes

The second way of using the XML client is to directly use the XMLProcessor and XMLSubscriptionHandler objects. The application client must handle the registering of a SIENA client with the SIENA server. It must also handle the actual publishing and subscribing of the SIENA notifications and filters. In order for the XMLProcessor and XMLSubscriptionHandler objects to correctly handle the XML notifications they need to have the setStyleSheet method called. This method, like the one for the XMLClient object, accepts a single parameter that is the name of the file containing the XSchema. The objects will process the file and build the internal map they use for translations. Only after this method has been invoked will the translation methods work correctly. If an error happens during the processing of the file the method will throw a siena.SienaException with a message containing information about the error.

If any user rules are to be used in the processing of the XML notification the setUserRules method should also be called at this time, though it is not necessary to do so. This method takes in the name of the file that contains the user rules. A siena.SienaException is thrown if there are any troubles processing the rules. If this method is called more than once then the rules in the newer file will be appended to the current set of rules. For rules that have the same tag specifier then the new rule will replace the current rule.

To publish a subscription the translate method on the XMLProcessor is called, passing it a string containing the XML notification. The method will return a siena.Notification object that contains the original XML notification translated into a SIENA notification. It is up to the application client to send the notification to the SIENA server. There are no methods on the XMLProcessor that do this for the application client.

Creating a subscription for a XML notification is more complex than sending a XML notification. The XMLSubscriptionHandler constructor requires a siena.Notifiable object to be passed in to it. The object passed in will be the one that will be notified when an XML notification has met the requirements of the XPath statement. After the XMLSubscriptionHandler is built then the setStyleSheet method is invoked as describe in a previous paragraph. The XMLSubscriptionHandler will then be ready to build filters for a subscription.

To build a filter for the subscription the buildFilter method is invoked with a single parameter. The parameter is the string containing the XPath expression to be used as the subscription. This method will return an array of siena.Filter objects that are based on the XPath passed in. The application client is responsible for submitting the Filter objects to a SIENA server as a subscription. When the application client does this the XMLSubscriptionHandler should be the object given to the server as the object to be notified. The reason for this is that the XMLSubscriptionHandler does some post processing on the SIENA notification in order to validate the XPath against the XML notification received. The application client must also call the attach method, passing in an Observer object. If this is not done then the application client will never receive a XML notification, as the XMLSubscriptionHandler has no way to inform the application client that a XML notification has been received. When a XML notification is validated, i.e. the XPath statement returns at least one node, the XMLSubscriptionHandler will notify the application client buy invoking the update method on the Observer object, passing itself into the method. The application client is responsible for getting the siena.Notifiable object from the XMLSubscriptionHandler, given to it during its construction, and call the notify method on it, passing it the siena.Notification object stored in the XMLSubscriptionHandler. The siena.Notification object can be retrieved from the XMLSubscriptionHandler by calling the getNotification method. The XML notification must be extracted by the Notifiable object from the siena.Notification object using the getAttribute method with the string parameter equal to “xml”. Calling toString on the returned object will

return a String object containing the complete XML notification sent by another client. This process is shown in figure 5.

```
Public void notify(siena.Notification notification) {
    String xmlNotification =
        notification.getAttribute("xml").toString();
}
```

Figure 5: Extracting a XML Notification

It is possible for the XMLSubscriptionHandler not to be the object notified by the SIENA servers when a SIENA notification matches a subscription but the application client would have to reconstruct the XML notification and then validate it against the XPath expression. This is duplicate work since the XMLSubscriptionHandler already does all of this.

Using either method to use the XML client interface to SIENA works. In fact, the XMLClient uses the second method itself to implement the first method. In the end though the same results occur, the use of XML as a notification language in the SIENA environment.