UNIT - V

5.1. WS-BPEL language basics

WS-BPEL language is used to demonstrate how process logic can be described as part of a concrete definition that can be implemented and executed via a compliant orchestration engine.

5.1.1. A brief history of BPEL4WS and WS-BPEL

The Business Process Execution Language for Web Services (BPEL4WS) was first conceived in July, 2002, with the release of the BPEL4WS 1.0 specification, a joint effort by IBM, Microsoft, and BEA. This document proposed an orchestration language inspired by previous variations, such as IBM’s Web Services Flow Language (WSFL) and Microsoft’s XLANG specification. Joined by other contributors from SAP and Siebel Systems, version 1.1 of the BPEL4WS specification was released less than a year later, in May of 2003. This version received more attention and vendor support, leading to a number of commercially available BPEL4WS-compliant orchestration engines. Just prior to this release, the BPEL4WS specification was submitted to an OASIS technical committee so that the specification could be developed into an official, open standard.

Fig 5.1. A common WS-BPEL process definition structure.

5.1.2. The process element

The process is the root element of a WS-BPEL process definition. It is assigned a name value using the name attribute and is used to establish the process definition-related namespaces.
Example 5.1. A skeleton process definition.

```xml
<process name="TimesheetSubmissionProcess" targetNamespace="http://www.xmltc.com/tls/process/
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/
xmlns:bpl="http://www.xmltc.com/tls/process/
xmlns:emp="http://www.xmltc.com/tls/employee/
xmlns:inv="http://www.xmltc.com/tls/invoice/
xmlns:tst="http://www.xmltc.com/tls/timesheet/
xmlns:not="http://www.xmltc.com/tls/notification/">
  <partnerLinks>
    ...
  </partnerLinks>
  <variables>
    ...
  </variables>
  <sequence>
    ...
  </sequence>
</process>
```

The process construct contains a series of common child elements like partner-link, variables, sequence etc.

5.1.2.1. The partnerLinks and partnerLink elements

A partnerLink element establishes the port type of the service (partner) that will be participating during the execution of the business process. Partner services can act as a client to the process, responsible for invoking the process service. Alternatively, partner services can be invoked by the process service itself. The contents of a partnerLink element represent the communication exchange between two partners the process service being one partner and another service being the other. Depending on the nature of the communication, the role of the process service will vary. For instance, a process service that is invoked by an external service may act in the role of "TimesheetSubmissionProcess." However, when this same process service invokes a different service to have an invoice verified, it acts within a different role, perhaps "InvoiceClient." The partnerLink
element therefore contains the myRole and partnerRole attributes that establish the service provider role of the process service and the partner service respectively. Put simply, the myRole attribute is used when the process service is invoked by a partner client service, because in this situation the process service acts as the service provider. The partnerRole attribute identifies the partner service that the process service will be invoking (making the partner service the service provider).

**Example 5.2.** The partnerLinks construct containing one partnerLink element in which the process service is invoked by an external client partner and four partnerLink elements that identify partner services invoked by the process service.

```xml
<partnerLinks>
  <partnerLink name="client" partnerLinkType="tns:TimesheetSubmissionType"
myRole="TimesheetSubmissionServiceProvider"/>
  <partnerLink name="Invoice" partnerLinkType="inv:InvoiceType"
partnerRole="InvoiceServiceProvider"/>
  <partnerLink name="Timesheet" partnerLinkType="tst:TimesheetType"
partnerRole="TimesheetServiceProvider"/>
  <partnerLink name="Employee" partnerLinkType="emp:EmployeeType"
partnerRole="EmployeeServiceProvider"/>
  <partnerLink name="Notification" partnerLinkType="not:NotificationType"
partnerRole="NotificationServiceProvider"/>
</partnerLinks>
```

**5.1.2.2. The partnerLinkType element**

For each partner service involved in a process, partnerLinkType elements identify the WSDL portType elements referenced by the partnerLink elements within the process definition. Therefore, these constructs typically are embedded directly within the WSDL documents of every partner service. The partnerLinkType construct contains one role element for each role the service can play, as defined by the partnerLink myRole and partnerRole attributes. As a result, a partnerLinkType will have either one or two child role elements.

**Example 5.3.** A WSDL definitions construct containing a partnerLinkType construct.

```xml
<definitions name="Employee" targetNamespace="http://www.xmltc.com/tls/employee/wsdl/
xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:plnk= "http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
```
5.1.2.3. The **variables** element

WS-BPEL process services commonly use the **variables** construct to store state information related to the immediate workflow logic. Entire messages and data sets formatted as XSD schema types can be placed into a variable and retrieved later during the course of the process. The type of data that can be assigned to a variable element needs to be predefined using one of the following three attributes: **messageType**, **element**, or **type**. The **messageType** attribute allows for the variable to contain an entire WSDL-defined message, whereas the element attribute simply refers to an XSD element construct. The type attribute can be used to just represent an XSD **simpleType**, such as string or integer.

**Example 5.4.** The **variables** construct hosting only some of the child **variable** elements used later by the Timesheet Submission Process.

```
<variables>
  <variable name="ClientSubmission" messageType="bpl:receiveSubmitMessage"/>
  <variable name="EmployeeHoursRequest" messageType="emp:getWeeklyHoursRequestMessage"/>
  <variable name="EmployeeHoursResponse" messageType="emp:getWeeklyHoursResponseMessage"/>
  <variable name="EmployeeHistoryRequest" messageType="emp:updateHistoryRequestMessage"/>
  <variable name="EmployeeHistoryResponse" messageType="emp:updateHistoryResponseMessage"/>
...```

M.Senthil Kumar, Asst. Prof.
Department of Information Technology, SMEC
<variables>

Typically, a variable with the messageType attribute is defined for each input and output message processed by the process definition. The value of this attribute is the message name from the partner process definition.

5.1.2.4. The getVariableProperty and getVariableData functions

WS-BPEL provides built-in functions that allow information stored in or associated with variables to be processed during the execution of a business process.

getVariableProperty(variable name, property name)

This function allows global property values to be retrieved from variables. It simply accepts the variable and property names as input and returns the requested value.

getVariableData(variable name, part name, location path)

Because variables commonly are used to manage state information, this function is required to provide other parts of the process logic access to this data. The getVariableData function has a mandatory variable name parameter and two optional arguments that can be used to specify a part of the variable data. In our examples we use the getVariableData function a number of times to retrieve message data from variables.

Example 5.5. Two getVariableData functions being used to retrieve specific pieces of data from different variables.

getVariableData ('InvoiceHoursResponse', 'ResponseParameter')

gistributeData ('input', 'payload', '/tns:TimesheetType/Hours/...')

5.1.2.5. The sequence element

The sequence construct allows you to organize a series of activities so that they are executed in a predefined, sequential order. WS-BPEL provides numerous activities that can be used to express the workflow logic within the process definition. The remaining element descriptions in this section explain the fundamental set of activities used as part of our upcoming case study examples.

Example 5.6. A skeleton sequence construct containing only some of the many activity elements provided by WS-BPEL.

<sequence>

<receive>

...

</receive>
<assign>
...
</assign>
<invoke>
...
</invoke>
<reply>

5.1.2.5.1. The *invoke* element

This element identifies the operation of a partner service that the process definition intends to invoke during the course of its execution. The *invoke* element is equipped with five common attributes, which further specify the details of the invocation.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>partnerLink</td>
<td>This element names the partner service via its corresponding partnerLink.</td>
</tr>
<tr>
<td>portType</td>
<td>The element used to identify the portType element of the partner service.</td>
</tr>
<tr>
<td>operation</td>
<td>The partner service operation to which the process service will need to send its request.</td>
</tr>
<tr>
<td>inputVariable</td>
<td>The input message that will be used to communicate with the partner service operation. Note that it is referred to as a variable because it is referencing a WS-BPEL variable element with a messageType attribute.</td>
</tr>
<tr>
<td>outputVariable</td>
<td>This element is used when communication is based on the request-response MEP. The return value is stored in a separate variable element.</td>
</tr>
</tbody>
</table>

**Example 5.7. The *invoke* element identifying the target partner service details.**

```xml
<invoke name="ValidateWeeklyHours" partnerLink="Employee" portType="emp:EmployeeInterface" operation="GetWeeklyHoursLimit" inputVariable="EmployeeHoursRequest" outputVariable="EmployeeHoursResponse"/>
```

5.1.2.5.2. The *receive* element

The *receive* element allows us to establish the information a process service expects upon receiving a request from an external client partner service. In this case, the process service is viewed as a service provider waiting to be invoked. The *receive* element contains a set of attributes, each of which...
is assigned a value relating to the expected incoming communication.

Table 5.2. receive element attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>partnerLink</td>
<td>The client partner service identified in the corresponding partnerLink construct.</td>
</tr>
<tr>
<td>portType</td>
<td>The process service portType that will be waiting to receive the request message from the partner service.</td>
</tr>
<tr>
<td>operation</td>
<td>The process service operation that will be receiving the request.</td>
</tr>
<tr>
<td>variable</td>
<td>The process definition variable construct in which the incoming request message will be stored.</td>
</tr>
<tr>
<td>createInstance</td>
<td>When this attribute is set to &quot;yes,&quot; the receipt of this particular request may be responsible for creating a new instance of the process.</td>
</tr>
</tbody>
</table>

Example 5.8. The receive element used in the Timesheet Submission Process definition to indicate the client partner service responsible for launching the process with the submission of a timesheet document.

```xml
<receive name="receiveInput" partnerLink="client" portType="tns:TimesheetSubmissionInterface" operation="Submit" variable="ClientSubmission" createInstance="yes"/>
```

5.1.2.5.3. The reply element

Where there's a receive element, there's a reply element when a synchronous exchange is being mapped out. The reply element is responsible for establishing the details of returning a response message to the requesting client partner service. Because this element is associated with the same partnerLink element as its corresponding receive element, it repeats a number of the same attributes.

Table 5.3. reply element attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>partnerLink</td>
<td>The same partnerLink element established in the receive element.</td>
</tr>
<tr>
<td>portType</td>
<td>The same portType element displayed in the receive element.</td>
</tr>
<tr>
<td>operation</td>
<td>The same operation element from the receive element.</td>
</tr>
<tr>
<td>variable</td>
<td>The process service variable element that holds the message that is returned to the partner service.</td>
</tr>
<tr>
<td>messageExchange</td>
<td>It is being proposed that this optional attribute be added by the WS-BPEL 2.0 specification. It allows for the reply element to be explicitly associated with a message activity capable of receiving a message (such as the receive element).</td>
</tr>
</tbody>
</table>
Example 5.9. A potential companion reply element to the previously displayed receive element.

```xml
<reply partnerLink="client" portType="tns:TimesheetSubmissionInterface" operation="Submit"
variable="TimesheetSubmissionResponse"/>
```

5.1.2.5.4. The switch, case, and otherwise elements

These three structured activity elements allow us to add conditional logic to our process definition, similar to the familiar select case/case else constructs used in traditional programming languages. The switch element establishes the scope of the conditional logic, wherein multiple case constructs can be nested to check for various conditions using a condition attribute. When a condition attribute resolves to "true," the activities defined within the corresponding case construct are executed. The otherwise element can be added as a catch all at the end of the switch construct. Should all preceding case conditions fail, the activities within the otherwise construct are executed.

Example 5.10. A skeleton case element wherein the condition attribute uses the getVariableData function to compare the content of the EmployeeResponseMessage variable to a zero value.

```xml
<switch>
  <case condition="getVariableData('EmployeeResponseMessage','ResponseParameter')=0">
    ...
  </case>
  <otherwise>
    ...
  </otherwise>
</switch>
```

5.1.2.5.5. The assign, copy, from, and to elements

This set of elements simply gives us the ability to copy values between process variables, which allows us to pass around data throughout a process as information is received and modified during the process execution.

Example 5.11. Within this assign construct, the contents of the TimesheetSubmissionFailedMessage variable are copied to two different message variables.

```xml
<assign>
  <copy>
    <from variable="TimesheetSubmissionFailedMessage"/>
    <to variable="EmployeeNotificationMessage"/>
  </copy>
</assign>
```
5.1.2.5.6. **faultHandlers, catch, and catchAll elements**

This construct can contain multiple catch elements, each of which provides activities that perform exception handling for a specific type of error condition. Faults can be generated by the receipt of a WSDL-defined fault message, or they can be explicitly triggered through the use of the throw element. The faultHandlers construct can consist of (or end with) a catchAll element to house default error handling activities.

**Example 5.12.** The faultHandlers construct hosting catch and catchAll child constructs.

```xml
<faultHandlers>
  <catch faultName="SomethingBadHappened" faultVariable="TimesheetFault">
    ...
  </catch>
  <catchAll>
    ...
  </catchAll>
</faultHandlers>
```

5.1.2.5.7. **Other WS-BPEL elements**

The following table provides brief descriptions of other relevant parts of the WS-BPEL language.

**Table 5.4. Quick reference table providing short descriptions for additional WS-BPEL elements.**

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compensationHandler</td>
<td>A WS-BPEL process definition can define a compensation process that kicks in a series of activities when certain conditions occur to justify a compensation. These activities are kept in the compensationHandler construct.</td>
</tr>
<tr>
<td>correlationSets</td>
<td>WS-BPEL uses this element to implement correlation, primarily to associate messages with process instances. A message can belong to multiple correlationSets.</td>
</tr>
<tr>
<td><strong>correlationSets</strong>. Further, message properties can be defined within WSDL documents.</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td></td>
</tr>
<tr>
<td><strong>empty</strong></td>
<td>This simple element allows you to state that no activity should occur for a particular condition.</td>
</tr>
<tr>
<td><strong>eventHandlers</strong></td>
<td>The <strong>eventHandlers</strong> element enables a process to respond to events during the execution of process logic. This construct can contain <strong>onMessage</strong> and <strong>onAlarm</strong> child elements that trigger process activity upon the arrival of specific types of messages.</td>
</tr>
<tr>
<td><strong>exit</strong></td>
<td>See the <strong>terminate</strong> element description that follows.</td>
</tr>
<tr>
<td><strong>flow</strong></td>
<td>A <strong>flow</strong> construct allows you to define a series of activities that can occur concurrently and are required to complete after all have finished executing. Dependencies between activities within a <strong>flow</strong> construct are defined using the child <strong>link</strong> element.</td>
</tr>
<tr>
<td><strong>pick</strong></td>
<td>Similar to the <strong>eventHandlers</strong> element, this construct also can contain child <strong>onMessage</strong> and <strong>onAlarm</strong> elements but is used more to respond to external events for which process execution is suspended.</td>
</tr>
<tr>
<td><strong>scope</strong></td>
<td>Portions of logic within a process definition can be sub-divided into scopes using this construct. This allows you to define variables, <strong>faultHandlers</strong>, <strong>correlationSets</strong>, <strong>compensationHandler</strong>, and <strong>eventHandlers</strong> elements local to the scope.</td>
</tr>
<tr>
<td><strong>terminate</strong></td>
<td>This element effectively destroys the process instance. The WS-BPEL 2.0 specification proposes that this element be renamed <strong>exit</strong>.</td>
</tr>
<tr>
<td><strong>throw</strong></td>
<td>WS-BPEL supports numerous fault conditions. Using the <strong>tHRow</strong> element allows you to explicitly trigger a fault state in response to a specific condition.</td>
</tr>
<tr>
<td><strong>wait</strong></td>
<td>The <strong>wait</strong> element can be set to introduce an intentional delay within the process. Its value can be a set time or a predefined date.</td>
</tr>
<tr>
<td><strong>while</strong></td>
<td>This useful element allows you to define a loop. As with the <strong>case</strong> element, it contains a <strong>condition</strong> attribute that, as long as it continues resolving to &quot;true,&quot; will continue to execute the activities within the <strong>while</strong> construct.</td>
</tr>
</tbody>
</table>
5.2. WS-Coordination overview

WS-Coordination, which can be used to realize some of the underlying mechanics for WS-BPEL orchestrations. Specifically, we describe some of the elements from the WS-Coordination specification and look at how they are used to implement the supplementary specifications that provide coordination protocols (WS-BusinessActivity and WS-AtomicTransaction). In terms of the WS-Coordination language and its two protocol documents, what may be of interest to you is the actual CoordinationContext header that is inserted into SOAP messages. You may encounter this header if you are monitoring messages or if you need to perform custom development associated with the coordination context.

5.2.1. The CoordinationContext element

This parent construct contains a series of child elements that each house a specific part of the context information being relayed by the header.

Example 5.13. A skeleton CoordinationContext construct.

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
    <Header>
        <wsc:CoordinationContext>
            <wsu:Identifier>
                ...
            </wsu:Identifier>
            <wsu:Expires>
                ...
            </wsu:Expires>
            <wsc:CoordinationType>
                ...
            </wsc:CoordinationType>
            <wsc:RegistrationService>
                ...
            </wsc:RegistrationService>
        </wsc:CoordinationContext>
    </Header>
</Envelope>
```
The activation service returns this CoordinationContext header upon the creation of a new activity. As described later, it is within the CoordinationType child construct that the activity protocol (WS-BusinessActivity, WS-AtomicTransaction) is carried. Vendor-specific implementations of WS-Coordination can insert additional elements within the CoordinationContext construct that represent values related to the execution environment.

5.2.1.1. The Identifier and Expires elements

These two elements originate from a utility schema used to provide reusable elements. WS-Coordination uses the Identifier element to associate a unique ID value with the current activity. The Expires element sets an expiry date that establishes the extent of the activity's possible lifespan.

Example 5.14. Identifier and Expires elements containing values relating to the header.

```xml
  ...
  <wsu:Identifier>
    http://www.xmltc.com/ids/process/33342
  </wsu:Identifier>
  <wsu:Expires>
    2008-07-30T24:00:00.000
  </wsu:Expires>
  ...
</Envelope>
```

5.2.1.2. The CoordinationType element

This element is to explain about the WS-BusinessActivity and WS-AtomicTransaction coordination types section.

5.2.1.2.1. Designating the WS-BusinessActivity coordination type
The specific protocol(s) that establishes the rules and constraints of the activity are identified within the CoordinationType element. The URI values that are placed here are predefined within the WS-BusinessActivity and WS-AtomicTransaction specifications. This first example shows the CoordinationType element containing the WS-BusinessActivity coordination type identifier. This would indicate that the activity for which the header is carrying context information is a potentially long-running activity.

Example 5.15. The CoordinationType element representing the WS-BusinessActivity protocol.

```xml
<wsc:CoordinationType>
  http://schemas.xmlsoap.org/ws/2004/01/wsba
</wsc:CoordinationType>
```

5.2.1.2.2. Designating the WS-AtomicTransaction coordination type

In the next example, the CoordinationType element is assigned the WS-AtomicTransaction coordination type identifier, which communicates the fact that the header's context information is part of a short running transaction.

Example 5.16. The CoordinationType element representing the WS-AtomicTransaction protocol.

```xml
<wsc:CoordinationType>
</wsc:CoordinationType>
```

5.2.1.3. The RegistrationService element

The RegistrationService construct simply hosts the endpoint address of the registration service. It uses the Address element also provided by the utility schema.

Example 5.17. The RegistrationService element containing a URL pointing to the location of the registration service.

```xml
<wsc:RegistrationService>
  <wsu:Address>
    http://www.xmltc.com/bpel/reg
  </wsu:Address>
</wsc:RegistrationService>
```

5.3. WS-Extensions

The following five key WS-* extensions:

- WS-Addressing
• WS-ReliableMessaging
• WS-Policy Framework
• WS-MetadataExchange
• WS-Security Framework

5.3.1. **WS-Addressing language basics**

The most common implementations of WS-Addressing standardize the representation of service endpoint locations and unique correlation values that tie together request and response exchanges. However, additional features are available that allow for the design of highly self-sufficient SOAP messages. Specifically, WS-Addressing includes extensions that support endpoint references for pointing messages to specific instances of Web services and message information (MI) headers that outfit messages with various types of transportation details. WS-Addressing is a core WS-* extension providing features that can be used intrinsically or alongside features offered by other WS-* specifications.

**Fig 5.2. How WS-Addressing relates to the other WS-* specifications discussed in this chapter.**

Diagram showing the relationships between WS-Reliable Messaging, WS-Policy, WS-Metadata Exchange, WS-Security, and WS-Addressing, indicating how each relates to Web services and SOAP messages.
5.3.1.1. The **EndpointReference** element

The **EndpointReference** element is used by the **From**, **ReplyTo**, and **FaultTo** elements described in the **Message information header elements** section. This construct can be comprised of a set of elements that assist in providing service interface information (including supplementary metadata), as well as the identification of service instances.

### Table 5.5. WS-Addressing endpoint reference elements.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>The standard WS-Addressing <strong>Address</strong> element used to provide the address of the service. This is the only required child element of the <strong>EndpointReference</strong> element.</td>
</tr>
<tr>
<td>ReferenceProperties</td>
<td>This construct can contain a series of child elements that provide details of properties associated with a service instance.</td>
</tr>
<tr>
<td>ReferenceParameters</td>
<td>Also a construct that can supply further child elements containing parameter values used for processing service instance exchanges.</td>
</tr>
<tr>
<td>PortType</td>
<td>The name of the service <strong>portType</strong>.</td>
</tr>
<tr>
<td>ServiceName and PortName</td>
<td>The names of the <strong>service</strong> and <strong>port</strong> elements that are part of the destination service WSDL definition construct.</td>
</tr>
<tr>
<td>Policy</td>
<td>This element can be used to establish related WS-Policy policy assertion information.</td>
</tr>
</tbody>
</table>

### Example 5.18. A SOAP header containing the **EndpointReference** construct.

```xml
<wsa:EndpointReference>
  <wsa:Address>
    http://www.xmltc.com/railco/...
  </wsa:Address>
  <wsa:ReferenceProperties>
    <app:id>
      unn:AFJK323llws
    </app:id>
  </wsa:ReferenceProperties>
  <wsa:ReferenceParameters>
    <app:sesno>...
  </wsa:ReferenceParameters>
</wsa:EndpointReference>
```
5.3.1.2. Message information header elements

This collection of elements can be used in various ways to assemble metadata-rich SOAP header blocks.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageID</td>
<td>An element used to hold a unique message identifier, most likely for correlation purposes. This element is required if the ReplyTo or FaultTo elements are used.</td>
</tr>
<tr>
<td>RelatesTo</td>
<td>This is also a correlation header element used to explicitly associate the current message with another. This element is required if the message is a reply to a request.</td>
</tr>
<tr>
<td>ReplyTo</td>
<td>The reply endpoint (of type EndpointReference) used to indicate which endpoint the recipient service should send a response to upon receiving the message. This element requires the use of MessageID.</td>
</tr>
<tr>
<td>From</td>
<td>The source endpoint element (of type EndpointReference) that conveys the source endpoint address of the message.</td>
</tr>
<tr>
<td>FaultTo</td>
<td>The fault endpoint element (also of type EndpointReference) that provides the address to which a fault notification should be sent. FaultTo also requires the use of MessageID.</td>
</tr>
<tr>
<td>To</td>
<td>The destination element used to establish the endpoint address to which the current message is being delivered.</td>
</tr>
<tr>
<td>Action</td>
<td>This element contains a URI value that represents an action to be performed when processing the MI header.</td>
</tr>
</tbody>
</table>

Example 5.19. A SOAP header with WS-Addressing message information header elements, three of which contain Endpoint Reference elements.

```xml
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:app="http://www.xmltc.com/railco/...">
  <Header>
    ...
  </Header>
</Envelope>
```
<wsa:Action>
  http://www.xmltc.com/tls/vp/submit
</wsa:Action>

<wsa:To>
  http://www.xmltc.com/tls/vp/...
</wsa:To>

<wsa:From>
  <wsa:Address>
    http://www.xmltc.com/railco/ap1/...
  </wsa:Address>
  <wsa:ReferenceProperties>
    <app:id>
      unn:AFJK323llws
    </app:id>
  </wsa:ReferenceProperties>
  <wsa:ReferenceParameters>
    <app:sesno>
      22322447
    </app:sesno>
  </wsa:ReferenceParameters>
</wsa:From>

<wsa:MessageID>
  uuid:243234234-43gf433
</wsa:MessageID>

<wsa:ReplyTo>
  <wsa:Address>
    http://www.xmltc.com/railco/ap2/
  </wsa:Address>
  <wsa:ReferenceProperties>
    <app:id>
      unn:AFJK323llws
    </app:id>
  </wsa:ReferenceProperties>
</wsa:ReplyTo>
5.3.2. WS-ReliableMessaging language basics

WS-ReliableMessaging introduces critical quality of service features for the guaranteed delivery or failure notification of SOAP messages. It also positions itself as a fundamental WS-* extension.
When message exchanges are governed by a WS-ReliableMessaging-capable communications framework, the concepts of sequences and acknowledgements become paramount to just about every message transmission. Coming up are descriptions for the following key WS-ReliableMessaging language elements:

- **Sequence** element
- **MessageNumber** element
- **LastMessage** element
- **SequenceAcknowledgement** element
- **AcknowledgementRange** element
- **Nack** element
- **AckRequested** element

Further supplementing these descriptions is a quick reference table containing brief descriptions of the following additional elements and assertions: SequenceRef, **AcknowledgementInterval**, **WS-Policy**, **WS-Metadata Exchange**, **WS-Addressing**, **WS-Security**.
5.3.2.1. The **Sequence**, **MessageNumber**, and **LastMessage** elements

The Sequence construct resides in the SOAP message header to represent the location of the current message in relation to the overall sequence of messages within which it is being delivered. To accomplish this, the **Sequence** construct relies on a set of child elements. The **Identifier** element is used to contain an ID value associated with the sequence itself, while the **MessageNumber** element contains a number that is the position of the message within the overall sequence order. Finally, the **LastMessage** element can be added to the **Sequence** construct to communicate the fact that the current message is the final message of the sequence.

**Example 5.20. A Sequence construct with a LastMessage element, indicating that this is the final message in the sequence.**

```xml
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsrm="http://schemas.xmlsoap.org/ws/2004/03/rm">
  <Header>
    <wsrm:Sequence>
      <wsu:Identifier>
        http://www.xmltc.com/railco/seq22231
      </wsu:Identifier>
      <wsrm:MessageNumber>
        12
      </wsrm:MessageNumber>
      <wsrm:LastMessage/>
    </wsrm:Sequence>
  </Header>
  <Body>
    ...
  </Body>
</Envelope>
```

5.3.2.2. The **SequenceAcknowledgement** and **AcknowledgementRange** elements

Upon the arrival of one or more messages within a sequence, the recipient service may issue a
message containing the SequenceAcknowledgement header construct to communicate that the original delivery was successful. This construct again uses the Identifier element to identify the sequence, but it also needs an element to convey which of the messages within the sequence were received and which were not. It accomplishes this through the use of the AcknowledgementRange element, which contains the Upper and Lower attributes that indicate a range of messages that were received. This range is based on the MessageNumber values of the messages, which, when they are first generated, are incremented. So one AcknowledgementRange element communicates each consecutive set of messages received. Therefore, a message that is not received is not accounted for within the ranges specified in the AcknowledgementRange elements.

Example 5.21. A SequenceAcknowledgement construct indicating that 11 out of a sequence of 15 messages were received.

```xml
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsrm="http://schemas.xmlsoap.org/ws/2004/03/rm">
  <Header>
    <wsrm:SequenceAcknowledgement>
      <wsu:Identifier>
        http://www.xmltc.com/tls/seq22231
      </wsu:Identifier>
      <wsrm:AcknowledgementRange Upper="4" Lower="1"/>
      <wsrm:AcknowledgementRange Upper="8" Lower="6"/>
      <wsrm:AcknowledgementRange Upper="12" Lower="11"/>
      <wsrm:AcknowledgementRange Upper="15" Lower="14"/>
    </wsrm:SequenceAcknowledgement>
  </Header>
  <Body>...
  </Body>
</Envelope>
```

5.3.2.3. The Nack element

Communicating the delivery failure of a message can, alternatively, be accomplished using the
Nack (negative acknowledgement) element. Instead of identifying which messages with MessageNumber values were received, it shows which were not.

**Example 5.22.** A *SequenceAcknowledgement* construct containing a **Nack** element that indicates that the fifth message was not received.

```xml
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/
           xmlns:wsrm="http://schemas.xmlsoap.org/ws/2004/03/rm">
  <Header>
    <wsrm:SequenceAcknowledgement>
      <wsu:Identifier>
        http://www.xmltc.com/tls/seq22231
      </wsu:Identifier>
      <wsrm:Nack>
        5
      </wsrm:Nack>
    </wsrm:SequenceAcknowledgement>
  </Header>
  <Body>
    ...
  </Body>
</Envelope>
```

**5.3.2.4. The AckRequested element**

RM destinations typically issue SOAP messages with *SequenceAcknowledgement* headers at predefined times, such as upon the receipt of a message containing the LastMessage element. However, an RM source service can request that the RM destination send out a sequence acknowledgement message on demand by using the **AckRequested** header construct. This construct simply contains a standard **Identifier** element to pinpoint the sequence for which it is requesting an acknowledgement message. It also can include a **MessageNumber** element that gives an indication as to which message receipt the RM source is most interested in.

**Example 5.23.** The **AckRequested** header construct indicating that the RM source would like to receive a sequence acknowledgement message.
<Envelope
xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsrm="http://schemas.xmlsoap.org/ws/2004/03/rm">  
  <Header>
    <wsrm:AckRequested>
      <wsu:Identifier>
        http://www.xmltc.com/tls/seq22232
      </wsu:Identifier>
    </wsrm:AckRequested>
  </Header>
  <Body>
    ...
  </Body>
</Envelope>

5.3.2.5. Other WS-ReliableMessaging elements

Table 5.7 Additional WS-ReliableMessaging elements.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SequenceRef</td>
<td>This construct allows you to attach policy assertions to a sequence, which introduces the ability to add various delivery rules, such as those expressed in the delivery assurances.</td>
</tr>
<tr>
<td>AcknowledgementInterval</td>
<td>Specifies an interval period that an RM destination can use to automatically transmit acknowledgement messages.</td>
</tr>
<tr>
<td>BaseRetransmissionInterval</td>
<td>An interval period used by the RM source to retransmit messages (for example, if no acknowledgements are received).</td>
</tr>
<tr>
<td>InactivityTimeout</td>
<td>A period of time that indicates at what point a sequence will time out and subsequently expire.</td>
</tr>
<tr>
<td>Expires</td>
<td>A specific date and time at which a sequence is scheduled to expire.</td>
</tr>
<tr>
<td>SequenceCreation</td>
<td>Sequences are generally created by the RM Source, but the RM Destination may use this element to force the creation of its own sequence.</td>
</tr>
</tbody>
</table>
5.3.3. WS-Policy language basics

The WS-Policy framework establishes a means of expressing service metadata beyond the WSDL definition. Specifically, it allows services to communicate rules and preferences in relation to security, processing, or message content. Policies can be applied to a variety of Web resources, positioning this specification as another fundamental part of the WS-* extensions.

Fig 5.4. How WS-Policy relates to the other WS-* specifications discussed in this chapter.

The WS-Policy framework is comprised of the following three specifications:

- WS-Policy
- WS-PolicyAssertions
- WS-PolicyAttachments

The following elements are used to demonstrate how policies are formulated and attached to element or document-level subjects:

- Policy element
- TextEncoding, Language, SpecVersion, and MessagePredicate assertions
5.3.3.1. The Policy element and common policy assertions

The Policy element establishes the root construct used to contain the various policy assertions that comprise the policy. The WS-PolicyAssertions specification supplies the following set of common, predefined assertion elements:

- **TextEncoding** Dictates the use of a specific text encoding format.
- **Language** Expresses the requirement or preference for a particular language.
- **SpecVersion** Communicates the need for a specific version of a specification.
- **MessagePredicate** Indicates message processing rules expressed using XPath statements.

These elements represent assertions that can be used to structure basic policies around common requirements. Policy assertions also can be customized, and other WS-* specifications may provide supplemental assertions. Each assertion can indicate whether its use is required or not via the value assigned to its Usage attribute. A value of "Required" indicates that its conditions must be met. Additionally, the use of the Preference attribute allows an assertion to communicate its importance in comparison to other assertions of the same type.

5.3.3.2. The ExactlyOne element

This construct surrounds multiple policy assertions and indicates that there is a choice between them, but that one must be chosen.

**Example 5.24.** The ExactlyOne construct housing two alternative policy assertions, one of which must be used.

```xml
  <wsp:ExactlyOne>
  </wsp:ExactlyOne>
</wsp:Policy>
```
5.3.3.3. The **All** element

The **All** construct introduces a rule that states that all of the policy assertions within the construct must be met. This element can be combined with the **ExactlyOne** element, where collections of policy assertions can each be grouped into **All** constructs that are then further grouped into a parent **ExactlyOne** construct. This indicates that the policy is offering a choice of assertions groups but that the assertions in any one of the alternative **All** groups must be met.

**Example 5.25. The **All** and **ExactlyOne** constructs used together to provide two alternative policy groups.**

```xml
   <wsp:ExactlyOne ID="Invoice1">
      <wsp:All ID="Invoice2">
         <wsp:TextEncoding wsp:Usage="wsp:Required" Encoding="iso-8859-5"/>
      </wsp:All>
      <wsp:TextEncoding wsp:Usage="wsp:Required" Encoding="iso-8859-5"/>
   </wsp:ExactlyOne>
</wsp:Policy>
```

5.3.3.4. The **Usage** attribute

As you've seen in the previous examples, a number of WS-Policy assertion elements contain a **Usage** attribute to indicate whether a given policy assertion is required. This attribute is a key part of the WS-Policy framework as its values form part of the overall policy rules. The **Usage** attribute actually has a number of settings given below.
Table 5.8. Possible settings for the Usage attribute.

<table>
<thead>
<tr>
<th>Attribute Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>The assertion requirements must be met, or an error will be generated.</td>
</tr>
<tr>
<td>Optional</td>
<td>The assertion requirements may be met, but an error will not be generated if they are not met.</td>
</tr>
<tr>
<td>Rejected</td>
<td>The assertion is unsupported.</td>
</tr>
<tr>
<td>Observed</td>
<td>The assertion applies to all policy subjects.</td>
</tr>
<tr>
<td>Ignored</td>
<td>The assertion will intentionally be ignored.</td>
</tr>
</tbody>
</table>

5.3.3.5. The Preference attribute

Policy assertions can be ranked in order of preference using this attribute. This is especially relevant if a service provider is flexible enough to provide multiple policy alternatives to potential service requestors. The Preference attribute is assigned an integer value. The higher this value, the more preferred the assertion. When this attribute is not used, a default value of "0" is assigned to the policy assertion.

5.3.3.6. The PolicyReference element

So far we've only been discussing the creation of policy documents. However, we have not yet established how policies are associated with the subjects to which they apply. The PolicyReference element is one way to simply link an element with one or more policies. Each PolicyReference element contains a URI attribute that points to one policy document or a specific policy assertion within the document. (The ID attribute of the policy or grouping construct is referenced via the value displayed after the "#" symbol.) If multiple PolicyReference elements are used within the same element, the policy documents are merged at runtime.

5.3.3.7. The PolicyURIs attribute


```xml
<Employee ...>
</Employee>
```

Alternatively, the PolicyURIs attribute also can be used to link to one or more policy documents. The attribute is added to an element and can be assigned multiple policy locations. As with PolicyReference, these policies are then merged at runtime.
Example 5.27. The **PolicyURIs** attribute referencing two policy documents.

```xml
<Employee wsp:PolicyURIs="http://www.xmltc.com/tls/policy1.xml#Employee1"
     "http://www.xmltc.com/tls/policy2.xml#Employee2"/>
```

### 5.3.3.8. The **PolicyAttachment** element

Another way of associating a policy with a subject is through the use of the **PolicyAttachment** construct. The approach taken here is that the child **AppliesTo** construct is positioned as the parent of the subject elements. The familiar **PolicyReference** element then follows the **AppliesTo** construct to identify the policy assertions that will be used.

**Example 5.28. The PolicyAttachment construct using the child AppliesTo construct to associate a policy with a WS-Addressing EndpointReference construct.**

```xml
<wsp:PolicyAttachment>
  <wsp:AppliesTo>
      <wsa:Address>
        http://www.xmltc.com/tls/ep1
      </wsa:Address>
      <wsa:PortType>
        emp:EmployeeInterface
      </wsa:PortType>
      <wsa:ServiceName>
        emp:Employee
      </wsa:ServiceName>
    </wsa:EndpointReference>
  </wsp:AppliesTo>
</wsp:PolicyAttachment>
```

### 5.3.3.9. Additional types of policy assertions

It is important to note that policy assertions can be utilized and customized beyond the conventional manner in which they are displayed in the preceding examples. For example:

- Policy assertions can be incorporated into WSDL definitions through the use of a special set of
policy subjects that target specific parts of the definition structure. A separate UsingPolicy element is provided for use as a WSDL extension.

- WS-ReliableMessaging defines and relies on WS-Policy assertions to enforce some of its delivery and acknowledgement rules.
- WS-Policy assertions can be created to communicate that a Web service is capable of participating in a business activity or an atomic transaction.
- A policy assertion can be designed to express a service's processing requirements in relation to other WS-* specifications.
- WS-Policy assertions commonly are utilized within the WS-Security framework to express security requirements.

5.3.4. WS-MetadataExchange language basics

WS-MetadataExchange provides a standardized means by which service description documents can be requested and supplied. This specification establishes a set of features that supports important SOA characteristics, such as interoperability and quality of service.

Fig 5.5. How WS-MetadataExchange relates to the other WS-* specifications discussed in this chapter.
The scope of the WS-MetadataExchange language is fairly small in comparison to other WS-* specifications. The following two forms of metadata requests are standardized:

- GetMetadata
- Get

The descriptions that follow discuss the primary elements used to compose these two types of request messages.

5.3.4.1. The **GetMetadata** element

This element can be placed on its own in the Body area of a SOAP message, or it can be turned into a construct that hosts child **Dialect** and **Identifier** elements (explained next).

**Example 5.29. A SOAP request message containing the GetMetadata element in the Body construct. Note the use of the WS-Addressing message information header elements in the SOAP header.**

```xml
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope"
<Header>
  <wsa:Action>
  </wsa:Action>
  <wsa:To>
    http://www.xmltc.com/tls/ap1/
  </wsa:To>
  <wsa:MessageID>
    uuid:23492372938
  </wsa:MessageID>
  <wsa:ReplyTo>
    <wsa:Address>
      http://www.xmltc.com/railco/inv1/
    </wsa:Address>
  </wsa:ReplyTo>
</Header>

<Body>
  <wsx:GetMetadata/>
  ...
</Body>
</Envelope>

5.3.4.2. The **Dialect** element

This element specifies the type and version of the metadata specification requested. The use of the **Dialect** element guarantees that the metadata returned to the service requesting it will be understood.

**Example 5.30.** The **Dialect** element being used to indicate that the XSD schema requested should comply to version 1.0 of the XML Schema Definition Language.

<Body>
  <wsx:GetMetadata>
    <wsx:Dialect>
      http://www.w3.org/2001/XMLSchema
    </wsx:Dialect>
  </wsx:GetMetadata>
</Body>
5.3.4.3. The Identifier element

While the Dialect element specifies the type of metadata being requested, this element further narrows the criteria by asking for a specific part of the metadata.

Example 5.31. The Identifier element added to specify the XSD schema’s target namespace.

5.3.4.4. The Metadata, MetadataSection, and MetadataReference elements

These three elements are used to organize the content of the message sent in response to a GetMetadata request. The parent Metadata construct resides in the SOAP message Body area and houses one or more child MetadataSection constructs that each represent a part of the returned metadata. If the contents of the metadata document are returned, they are placed within the MetadataSection construct. However, if only a pointer to the document is returned, its location is found in the MetadataReference construct (further qualified by a regular WS-Addressing Address element).

Example 5.32. A GetMetadata response message returning the contents of an entire WSDL definition, along with a pointer to the associated XSD schema.
</wsa:Action>
<wsa:RelatesTo>
23492372938
</wsa:RelatesTo>
<wsa:To>
http://www.xmltc.com/railco/inv1
</wsa:To>
</Header>
</Body>
<wsx:Metadata>
<wsx:MetadataSection ...
<wsdl:definitions>
... the entire WSDL definition ...
</wsdl:definitions>
</wsx:MetadataSection>
<wsx:MetadataSection ...
<wsx:MetadataReference>
<wsa:Address>
http://www.xmltc.com/tls/ap1/schemas
</wsa:Address>
</wsx:MetadataReference>
</wsx:MetadataSection>
</wsx:Metadata>
</Body>
</Envelope>

5.3.4.5. The Get message

The response to a GetMetadata request message can include a MetadataReference construct that contains the location of metadata documents not returned in this initial message. To explicitly request one of these documents, a separate Get message is issued. While this message does not contain a
specific Get element, it does adhere to a standardized SOAP header format, as follows.

Example 5.33. A Get message SOAP header identified by the Action element value. The resource being requested is targeted in the To element.

<Header>
  <wsa:Action>
  </wsa:Action>
  <wsa:MessageID>23492372938</wsa:MessageID>
  <wsa:ReplyTo>
    <wsa:Address>
      http://www.xmltc.com/railco/sub1
    </wsa:Address>
  </wsa:ReplyTo>
  <wsa:To>
  </wsa:To>
</Header>

5.3.5. WS-Security language basics

The WS-Security framework provides extensions that can be used to implement message-level security measures. These protect message contents during transport and during processing by service intermediaries. Additional extensions implement authentication and authorization control, which protect service providers from malicious requestors. WS-Security is designed to work with any of the WS-* specifications.

Fig 5.6. How WS-Security relates to the other WS-* specifications discussed in this chapter.
The WS-Security framework is comprised of numerous specifications, many in different stages of acceptance and maturation. In this book we've concentrated on some of the more established ones, namely:

- WS-Security
- XML-Encryption
- XML-Signature

Note that WS-Security represents a framework but also a specification that defines language elements. Because the language element descriptions provided in this chapter originate from three separate specifications, we qualify each element name with its origin.

5.3.5.1. **The Security element (WS-Security)**

This construct represents the fundamental header block provided by WS-Security. The Security element can have a variety of child elements, ranging from XML-Encryption and XML-Signature constructs to the token elements provided by the WS-Security specification itself. Security elements can be outfitted with *actor* attributes that correspond to SOAP actor roles. This allows you to add...
multiple Security blocks to a SOAP message, each intended for a different recipient.

5.3.5.2. The UsernameToken, Username, and Password elements (WS-Security)

The UsernameToken element provides a construct that can be used to host token information for authentication and authorization purposes. Typical children of this construct are the Username and Password child elements, but custom elements also can be added.

5.3.5.3. The BinarySecurityToken element (WS-Security)

Tokens stored as binary data, such as certificates, can be represented in an encoded format within the BinarySecurityToken element.

5.3.5.4. The SecurityTokenReference element (WS-Security)

This element allows you to provide a pointer to a token that exists outside of the SOAP message document.

Example 5.34. The Security SOAP header used by RailCo to provide user name and password values.

```xml
<Envelope xmlns="http://schemas.xmlsoap.org/soap-envelope/">
  <Header>
      <wsse:UsernameToken>
        <wsse:Username>
          rco-3342
        </wsse:Username>
        <wsse:Password Type="wsse:PasswordDigest">
          93292348347
        </wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </Header>
  <Body>
    ...
  </Body>
</Envelope>
```

5.3.5.5. Composing Security element contents (WS-Security)
As previously mentioned, the WS-Security specification positions the Security element as a standardized container for header blocks originating from other security extensions. The following example illustrates this by showing how a SAML block is located within the Security construct. (As previously mentioned, single sign-on languages are beyond the scope of this book. The SAML-specific elements shown in this example therefore are not explained.)

Example 5.35. The WS-Security SOAP header hosting a SAML authorization assertion.

```xml
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Header>
      <saml:Assertion xmlns:saml="..."...>
        <saml:Conditions ...>
          <saml:AuthorizationDecisionStatement Decision="Permit"
            Resource="http://www.xmltc.com/tls/...">
            <saml:Actions>
              ...<saml:Action>
                Execute
              </saml:Action>
            </saml:Actions>
            ...<saml:AuthorizationDecisionStatement>
        </wsse:Security>
  </Header>
</Envelope>
```

5.3.5.6. The EncryptedData element (XML-Encryption)

This is the parent construct that hosts the encrypted portion of an XML document. If located at the root of an XML document, the entire document contents are encrypted. The EncryptedData element's Type attribute indicates what is included in the encrypted content. For example, a value of...
http://www.w3.org/2001/04/xmlenc#Element indicates that the element and its contents will be encrypted, whereas the value of http://www.w3.org/2001/04/xmlenc#Content states that encryption will only be applied to the content within the opening and closing tags.

5.3.5.7. The CipherData, CipherValue, and CipherReference elements (XML-Encryption)

The CipherData construct is required and must contain either a CipherValue element hosting the characters representing the encrypted text or a CipherReference element that provides a pointer to the encrypted values. Following is an example of an XML document instance of this schema.

```
<InvoiceType>
  <Number>
    2322
  </Number>
  <Total>
    $32,322.73
  </Total>
  <Date>
    07.16.05
  </Date>
</InvoiceType>
```

Example 5.36. An XML document within a SOAP message containing an encrypted element.

```
<InvoiceType>
  <Number>
    2322
  </Number>
  <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
                   Type="http://www.w3.org/2001/04/xmlenc#Element">
    <CipherData>
      <CipherValue>
        R5J7UUI78
      </CipherValue>
    </CipherData>
  </EncryptedData>
</InvoiceType>
```
5.3.5.8. XML-Signature elements

A digital signature is a complex piece of information comprised of specific parts that each represent an aspect of the document being signed. Therefore, numerous elements can be involved when defining the construct that hosts the digital signature information. Table 17.5 provides brief descriptions of some of the main elements.

Table 5.9. XML-Signature elements

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanonicalizationMethod</td>
<td>This element identifies the type of &quot;canonicalization algorithm&quot; used to detect and represent subtle variances in the document content (such as the location of white space).</td>
</tr>
<tr>
<td>DigestMethod</td>
<td>Identifies the algorithm used to create the signature.</td>
</tr>
<tr>
<td>DigestValue</td>
<td>Contains a value that represents the document being signed. This value is generated by applying the DigestMethod algorithm to the XML document.</td>
</tr>
<tr>
<td>KeyInfo</td>
<td>This optional construct contains the public key information of the message sender.</td>
</tr>
<tr>
<td>Signature</td>
<td>The root element, housing all of the information for the digital signature.</td>
</tr>
<tr>
<td>SignatureMethod</td>
<td>The algorithm used to produce the digital signature. The digest and canonicalization algorithms are taken into account when creating the signature.</td>
</tr>
<tr>
<td>SignatureValue</td>
<td>The actual value of the digital signature.</td>
</tr>
<tr>
<td>SignedInfo</td>
<td>A construct that hosts elements with information relevant to the SignatureValue element, which resides outside of this construct.</td>
</tr>
<tr>
<td>Reference</td>
<td>Each document that is signed by the same digital signature is represented by a Reference construct that hosts digest and optional transformation details.</td>
</tr>
</tbody>
</table>
Example 5.37. A SOAP message header containing a digital signature.

```xml
<Envelope ...>
  <Header>
      <Signature Id="RailCo333" xmlns="http://www.w3.org/2000/09/xmldsig#">
        <SignedInfo>
          <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
          <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
          <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126"/>
          <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
          <DigestValue>LLSFK032093548=</DigestValue>
        </SignedInfo>
        <SignatureValue>9879DFSS3=</SignatureValue>
        <KeyInfo>...
        </KeyInfo>
      </Signature>
    </wsse:Security>
  </Header>
  <Body>
    ...invoice document with total exceeding $30,000...
  </Body>
</Envelope>
```