UNIT V

Service-Oriented Analysis

The process of determining how business automation requirements can be represented through service-orientation is the domain of the service-oriented analysis.

• ***Objectives of service-oriented analysis***

– The primary questions addressed during this phase are:

• What services need to be built?

• What logic should be encapsulated by each service?

• The overall goals of performing a service-oriented analysis are as follows:

– Define a preliminary set of service operation candidates.

– Group service operation candidates into logical contexts. These contexts represent service candidates.

– Define preliminary service boundaries so that they do not overlap with any existing or planned services.

– Identify encapsulated logic with reuse potential.

– Ensure that the context of encapsulated logic is appropriate for its intended use

– Define any known preliminary composition models.

• The service-oriented analysis process

– Service-oriented analysis can be applied at different levels, depending on which of the SOA delivery strategies are used to produce services.

– The previous chapter, the chosen strategy will determine the layers of abstraction that comprise the service layers of a solution environment.

– From an analysis perspective, each layer has different modeling requirements.

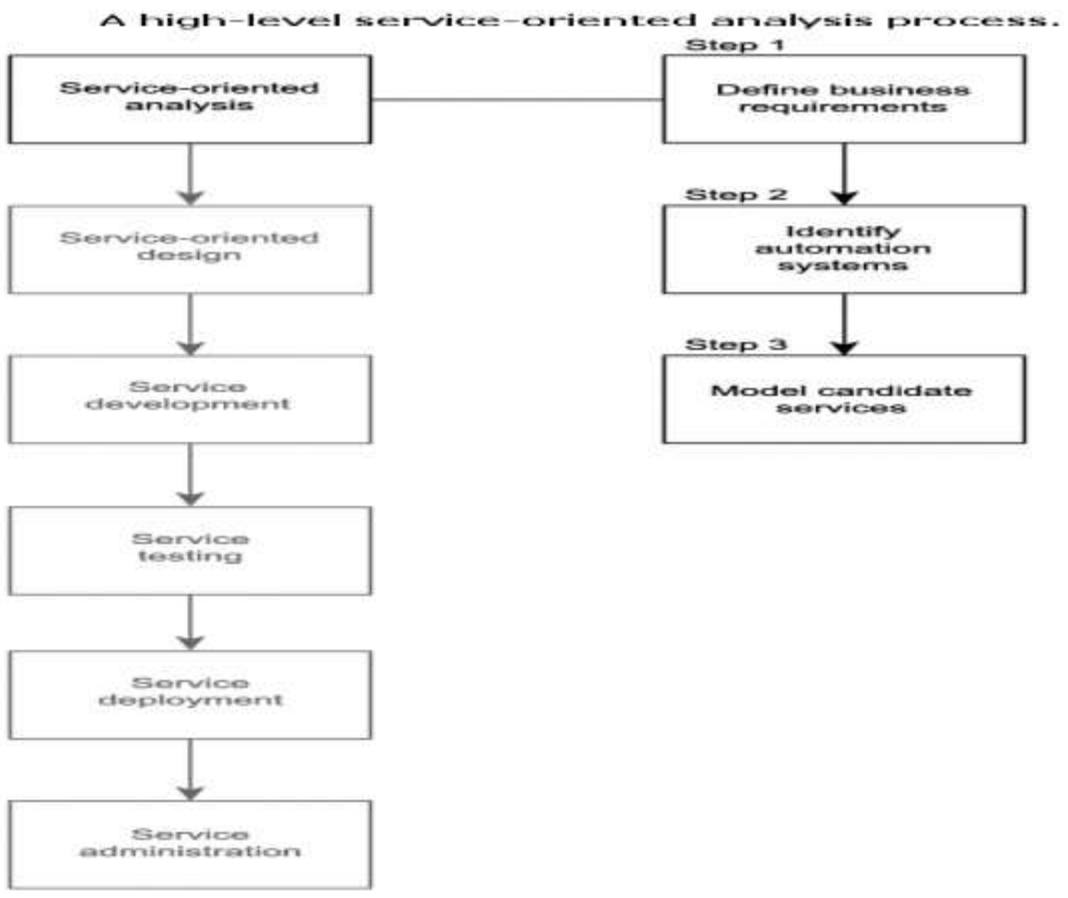
– Other questions that should be answered prior to proceeding with the serviceoriented analysis include:

• What outstanding work is needed to establish the required business model (s) and ontology?

• What modeling tools will be used to carry out the analysis?

• Will the analysis be part of an SOA transition plan?

– The service-oriented analysis process is a sub-process of the overall SOA delivery lifecycle.



Step 1: Define business automation requirements

- Through whatever means business requirements are normally collected, their documentation is required for this analysis process to begin. Given that the scope of our analysis centers around the creation of services in support of a service-oriented solution, only requirements related to the scope of that solution should be considered.
- Business requirements should be sufficiently mature so that a high-level automation process can be defined. This business process documentation will be used as the starting point of the service modeling process described in Step 3.

Step 2: Identify existing automation systems

- Existing application logic that is already, to whatever extent, automating any of the requirements identified in Step 1 needs to be identified. While a service-oriented analysis will not determine how exactly Web services will encapsulate or replace legacy application logic, it does assist us in scoping the potential systems affected.

- The details of how Web services relate to existing systems are ironed out in the serviceoriented design phase. For now, this information will be used to help identify application service candidates during the service modeling process described in Step 3.

Step 3: Model candidate services

• A service-oriented analysis introduces the concept of service modeling a process by which service operation candidates are identified and then grouped into a logical context. These groups eventually take shape as service candidates that are then further assembled into a tentative composite model representing the combined logic of the planned serviceoriented application. Three major levels of abstraction within SOA:

• Operations: Transactions that represent single logical units of work (LUWs). Execution of an operation will typically cause one or more persistent data records to be read, written, or modified. SOA operations are directly comparable to object-oriented (OO) methods. They have a specific, structured interface, and return structured responses. Just as for methods, the execution of a specific operation might involve invocation of additional operations.

• Services: Represent logical groupings of operations. For example, if we view CustomerProfiling as a service, then,Lookup customer by telephone number, List customers by name and postal code, and Save data for new customerrepresent the associated operations.

• Business Processes: A long running set of actions or activities performed with specific business goals in mind. Business processes typically encompass multiple service invocations. Examples of business processes are: Initiate New Employee, Sell Products or Services, and Fulfill Order.

Service-Oriented Design

• Service-oriented design is the process by which concrete physical service designs are derived from logical service candidates and then assembled into abstract compositions that implement a business process.

• Objectives of service-oriented design

– The primary questions answered by this phase are:

• How can physical service interface definitions be derived from the service candidates modeled during the service-oriented analysis phase?

• What SOA characteristics do we want to realize and support?

• What industry standards and extensions will be required by our SOA to implement the planned service designs and SOA characteristics?

– To address these questions, the design process actually involves further analysis. This time our focus is on environmental factors and design standards that will shape our services.

– The overall goals of performing a service-oriented design are as follows:

- Determine the core set of architectural extensions.
- Set the boundaries of the architecture.
- Identify required design standards.
- Define abstract service interface designs.
- Identify potential service compositions.
- Assess support for service-orientation principles.
- Explore support for characteristics of contemporary SOA.
- Design standards" versus "Industry standards"

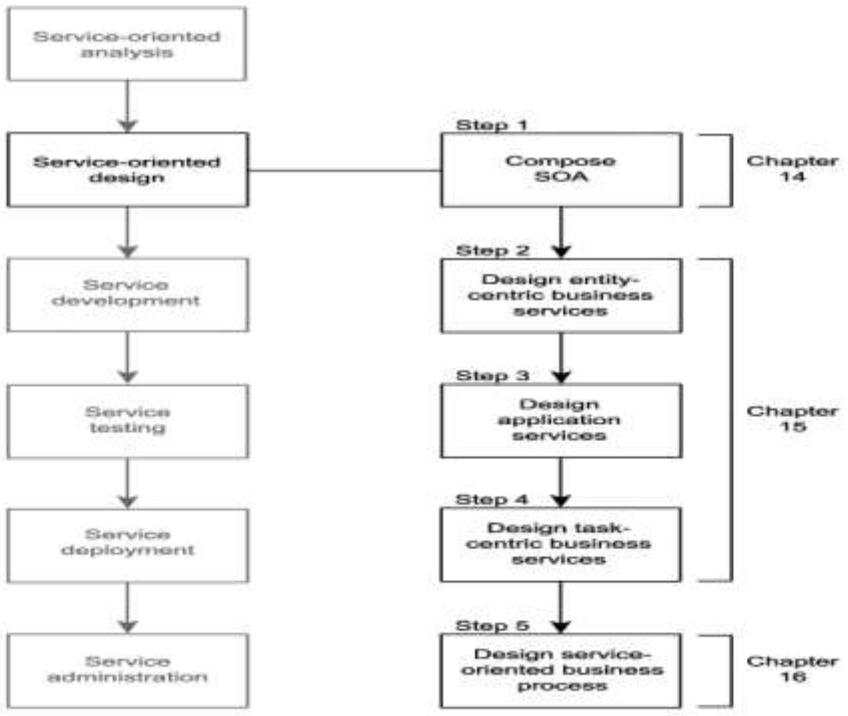
– Design standards represent custom standards created by an organization to ensure that services and SOAs are built according to a set of consistent conventions.

– Industry standards are provided by standards organizations and are published in Web services and XML specifications

The service-oriented design process

– We first establish a parent process that begins with some preparatory work. This leads to a series of iterative processes that govern the creation of different types of service designs and, ultimately, the design of the overall solution workflow

A high-level service-oriented design process.



Service-oriented analysis
Service-oriented design
Service development
Service testing
Service deployment
Service administration
Step 1
Compose SOA
Chapter 14
Step 2
Design entity-centric business services
Step 3
Design application services
Step 4
Design task-centric business services
Chapter 15
Step 5
Design service-oriented business process
Chapter 16

Step 1: Compose SOA

• A fundamental quality of SOA is that each instance of a service-oriented architecture is uniquely composable. Although most SOAs will implement a common set of shared technologies based on key XML and first-generation Web services specifications, the modular nature of the WS-* specification landscape allows for extensions to this core architecture to be added as required.

• This step consists of the following three further steps

1.Choose service layers

2.Position core SOA standards.

3.Choose SOA extensions.

Steps 2 to 4: Design services

• These steps are represented by the following three separate processes

– Entity-centric business service design process.

– Application service design process.

– Task-centric business service design process.

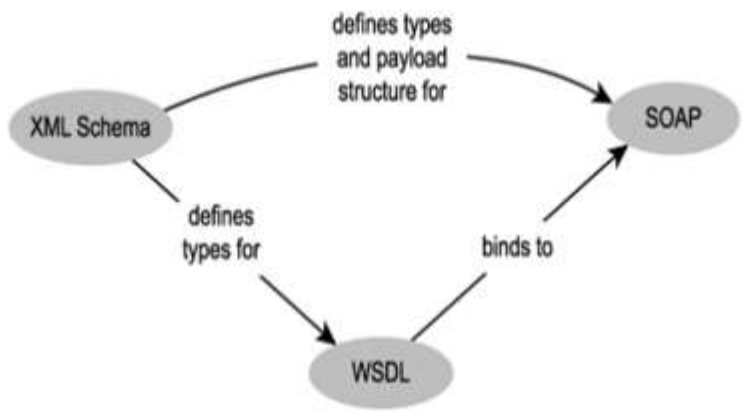
- Our primary input for each of these service design processes is the corresponding service candidates we produced in the service modeling process during the service-oriented analysis.

Step 5: Design service-oriented Business process

– Upon establishing an inventory of service designs, we proceed to create our orchestration layer the glue that binds our services with business process logic. This step results in the formal, executable definition of workflow logic, which translates into the creation of a WS-BPEL process definition.

- Prerequisites
  - Before we get into the details of the service-oriented design process, we should make sure that we have a sufficient understanding of key parts of the languages required to design services.
  - WSDL and SOAP
  - XML Schema Definition Language

Three core specifications associated with service design.



Service Modeling

- A service modeling process is essentially an exercise in organizing the information we gathered in Steps 1 and 2 of the parent service-oriented analysis process.
- "Services" versus "Service Candidates"

– Candidate

- The primary goal of the service-oriented analysis stage is to figure out what it is we need to later design and build in subsequent project phases. We are producing abstract candidates that may or may not be realized as part of the eventual concrete design.

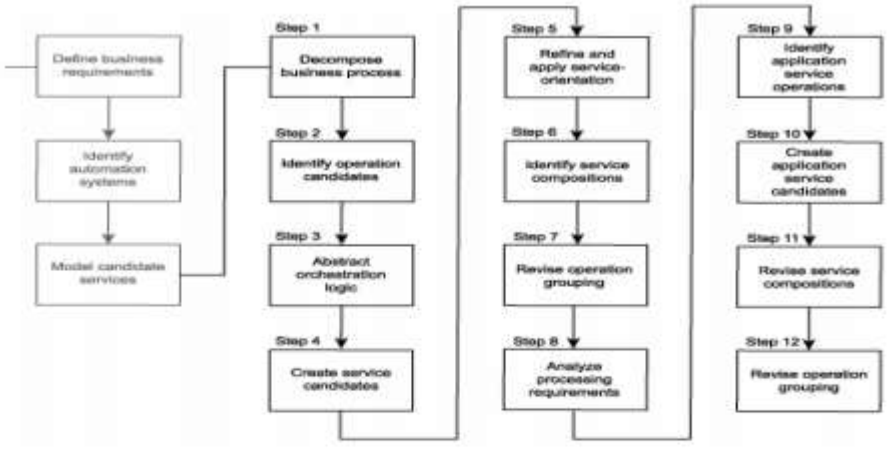
– Service Candidates

- Their behavior has significant departure from the corresponding original candidate having subjected to the realities of the technical architecture. Then we propose service operation candidates. Finally, service candidates and service operation candidates are the end-result of a process called service modeling.

• Process description

- A series of 12 steps that comprise a proposed service modeling process

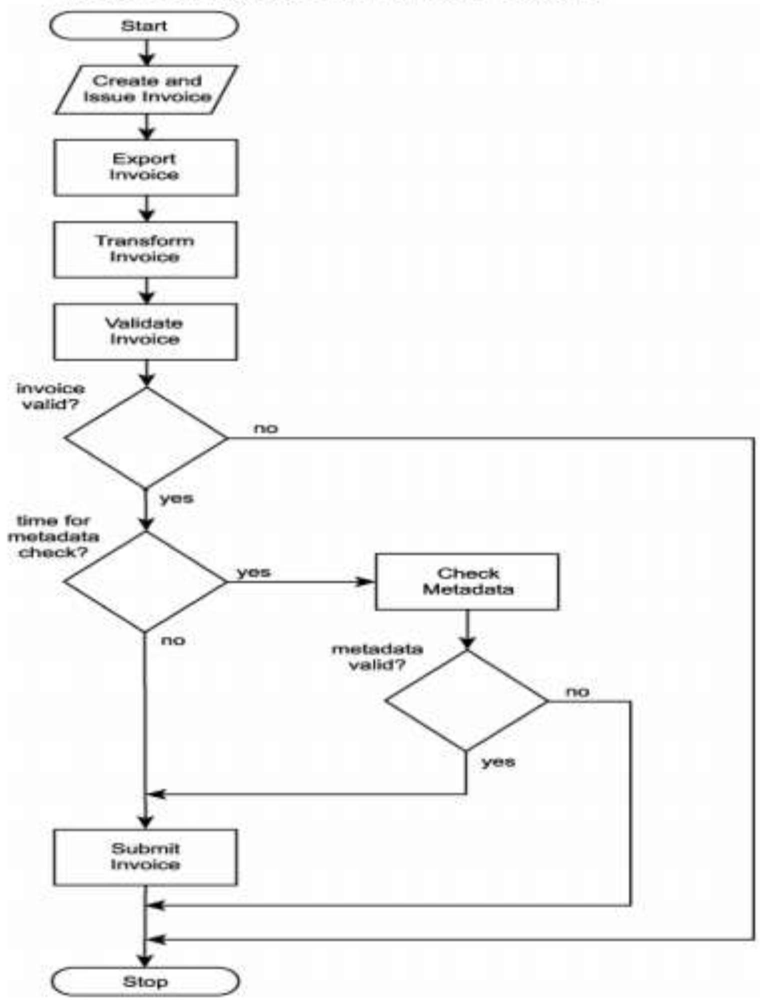
A sample service modeling process.


Define business requirements
Identify automation systems
Model candidate services
Step 1
Decompose business process
Step 2
Identify operation candidates
Step 3
Abstract orchestration logic
Step 4
Create service candidates
Step 5
Refine and apply service-orientation
Step 6
Identify service compositions
Step 7
Revise operation grouping
Step 8
Analyze processing requirements
Step 9
Identify application service operations
Step 10
Create application service candidates
Step 11
Revise service compositions
Step 12
Revise operation grouping

Step 1: Decompose the business process

– Take the documented business process and break it down into a series of granular process steps. It is important that a process's workflow logic be decomposed into the most granular representation of processing steps, which may differ from the level of granularity at which the process steps were originally documented.

The RailCo Invoice Submission Process.



Step 2: Identify business service operation candidates
– Some steps within a business process can be easily identified as not belonging to the potential logic that should be encapsulated by a service candidate

– Examples include:

– Manual process steps that cannot or should not be automated.

– Process steps performed by existing legacy logic for which service candidate encapsulation is not an option.

– By filtering out these parts we are left with the processing steps most relevant to our service modeling process.

Step 3: Abstract orchestration logic

– We should identify the parts of the processing logic that this layer would potentially abstract. (If you are not incorporating an orchestration service layer, then skip this step.)

– Potential types of logic suitable for this layer include:

- business rules
- conditional logic
- exception logic
- sequence logic

– These forms of orchestration logic may or may not be represented accurately by a step description. In this case, only remove the condition and leave the action.

– Also some of the identified workflow logic likely will be dropped eventually. This is because not all processing steps necessarily become service operations.

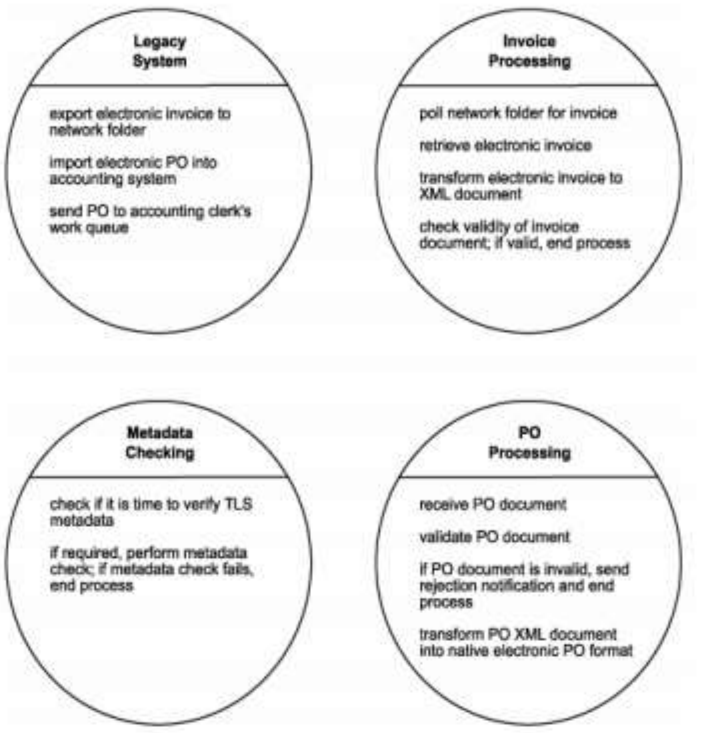
Step 4: Create business service candidates

– Review the processing steps that remain and determine one or more logical contexts with which these steps can be grouped. Each context represents a service candidate. The contexts you end up with will depend on the types of business services you have chosen to build

– For example, task-centric business services will require a context specific to the process, while entity-centric business services will introduce the need to group processing steps according to their relation to previously defined entities.

– Also it is encouraged that entity-centric business service candidates be equipped with additional operation candidates that facilitate future reuse

Our first round of service candidates.


Legacy System
export electronic invoice to network folder
import electronic PO into accounting system
send PO to accounting clerk's work queue
Invoice Processing
poll network folder for invoice
retrieve electronic invoice
transform electronic invoice to XML document
check validity of invoice document; if valid, end process
Metadata Checking
check if it is time to verify TLS metadata
if required, perform metadata check; if metadata check fails, end process
PO Processing
receive PO document
validate PO document
if PO document is invalid, send rejection notification and end process
transform PO XML document into native electronic PO format

Step 5: Refine and apply principles of service-orientation

– To make our service candidates truly worthy of an SOA, we must take a closer look at the underlying logic of each proposed service operation candidate.

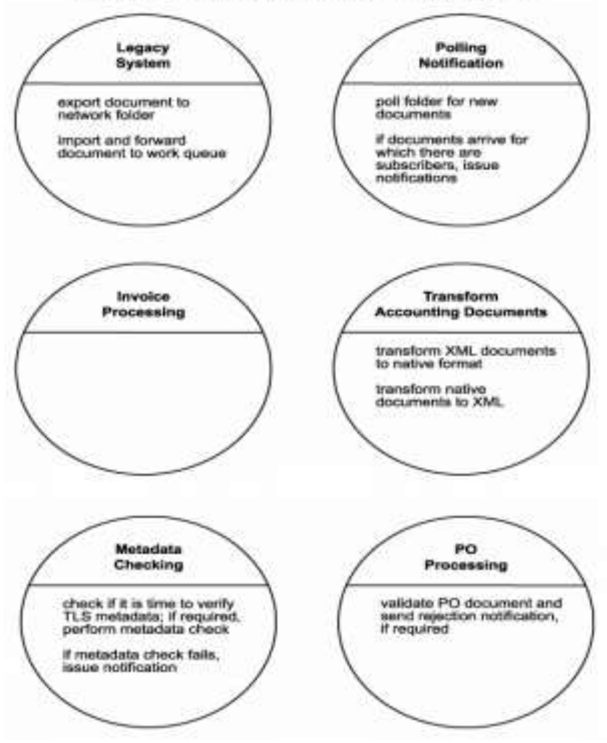
– This step gives us a chance to make adjustments and apply key service-orientation principles.
– This is where the study we performed in the Native Web service support for service-orientation principles section becomes useful.

– We identified the following four key principles as those not intrinsically provided through the use of Web services:

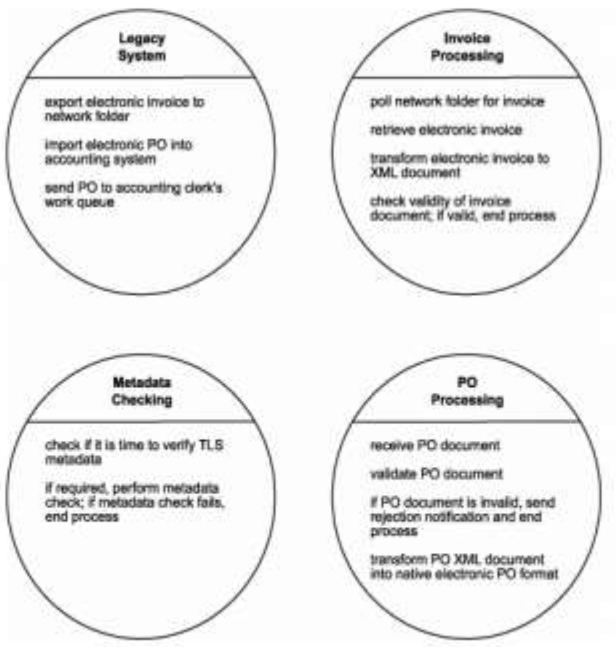
- reusability
- autonomy
- statelessness
- Discoverability

– Of these four, only the first two are important to us at the service modeling stage.

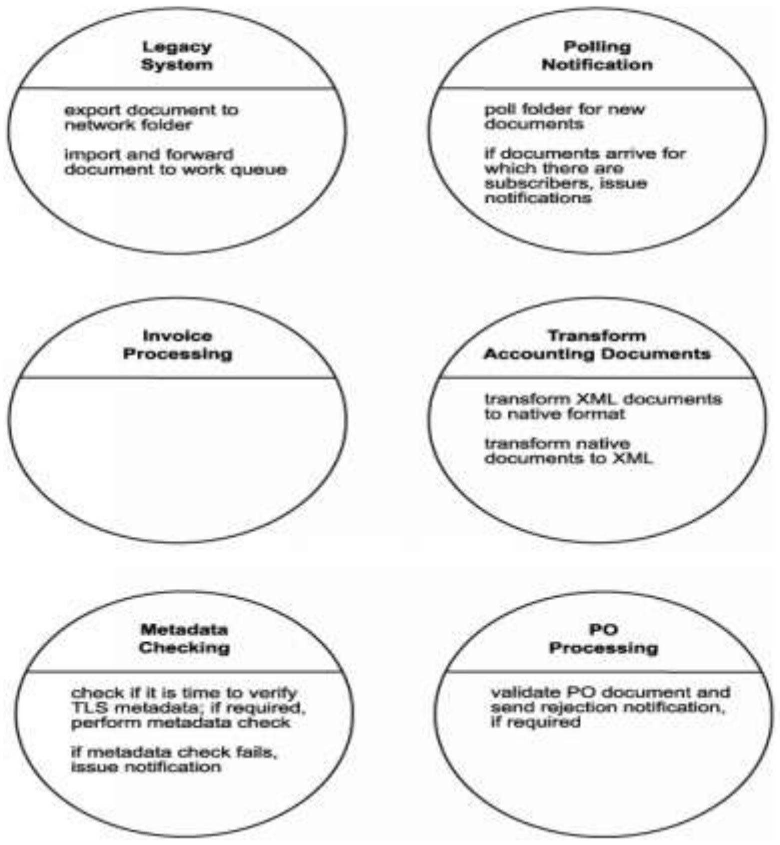
The revised set of RailCo service candidates.


Legacy System
export document to network folder
import and forward document to work queue
Polling Notification
poll folder for new documents
if documents arrive for which there are subscribers, issue notifications
Invoice Processing
Transform Accounting Documents
transform XML documents to native format
transform native documents to XML
Metadata Checking
check if it is time to verify TLS metadata; if required, perform metadata check
if metadata check fails, issue notification
PO Processing
validate PO document and send rejection notification, if required

Our first round of service candidates.


Legacy System
export electronic invoice to network folder
import electronic PO into accounting system
send PO to accounting clerk's work queue
Invoice Processing
poll network folder for invoice
retrieve electronic invoice
transform electronic invoice to XML document
check validity of invoice document; if valid, end process
Metadata Checking
check if it is time to verify TLS metadata
if required, perform metadata check; if metadata check fails, end process
PO Processing
receive PO document
validate PO document
if PO document is invalid, send rejection notification and end process
transform PO XML document into native electronic PO format

The revised set of RailCo service candidates.


Legacy System
export document to network folder
import and forward document to work queue
Polling Notification
poll folder for new documents
if documents arrive for which there are subscribers, issue notifications
Invoice Processing
Transform Accounting Documents
transform XML documents to native format
transform native documents to XML
Metadata Checking
check if it is time to verify TLS metadata; if required, perform metadata check
if metadata check fails, issue notification
PO Processing
validate PO document and send rejection notification, if required

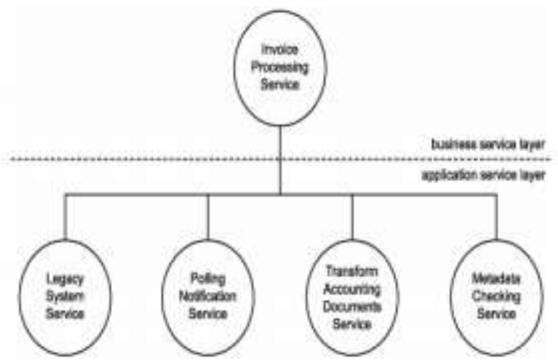
Step 6: Identify candidate service compositions

– Identify a set of the most common scenarios that can take place within the boundaries of the business process. For each scenario, follow the required processing steps as they exist now.

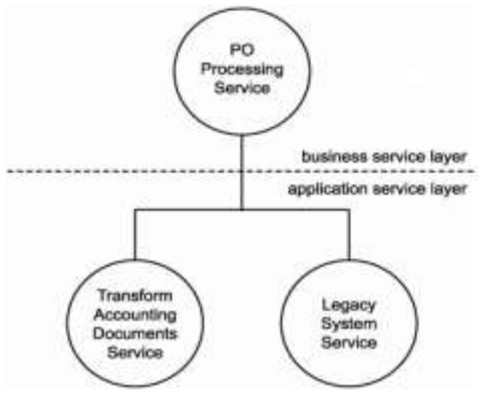
– This exercise accomplishes the following:

- It gives you a good idea as to how appropriate the grouping of your process steps is.
- It demonstrates the potential relationship between orchestration and business service layers.
- It identifies potential service compositions.
- It highlights any missing workflow logic or processing steps.

A sample composition representing the Invoice Submission Process.


Invoice Processing Service
business service layer
application service layer
Legacy System Service
Polling Notification Service
Transform Accounting Documents Service
Metadata Checking Service

A sample composition representing the Order Fulfillment Process.



Step 7: Revise business service operation grouping

– Based on the results of the composition exercise in Step 6, revisit the grouping of your business process steps and revise the organization of service operation candidates as necessary. It is not unusual to consolidate or create new groups (service candidates) at this point.

• Step 8: Analyze application processing requirements

– By the end of Step 6, you will have created a business-centric view of your services layer. This view could very well consist of both application and business service candidates, but the focus so far has been on representing business process logic.

– This next series of steps is optional and more suited for complex business processes and larger service-oriented environments. It requires that you more closely study the underlying processing requirements of all service candidates to abstract any further technology-centric service candidates from this view that will complete a preliminary application services layer.

– Specifically, what you need to determine is:

• What underlying application logic needs to be executed to process the action described by the operation candidate.

• Whether the required application logic already exists or whether it needs to be newly developed.

• Whether the required application logic spans application boundaries. In other words, is more than one system required to complete this action?

• Step 9: Identify application service operation candidates

– Break down each application logic processing requirement into a series of steps. Be explicit about how you label these steps so that they reference the function they are performing. Ideally, you would not reference the business process step for which this function is being identified.

• Step 10: Create application service candidates

– Group these processing steps according to a predefined context. With application service candidates, the primary context is a logical relationship between operation candidates.

– This relationship can be based on any number of factors, including:

• association with a specific legacy system

• association with one or more solution components

• logical grouping according to type of function

• Step 11: Revise candidate service compositions

– Revisit the original scenarios you identified in Step 5 and run through them again. Only, this time, incorporate the new application service candidates as well. This will result in the mapping of elaborate activities that bring to life expanded service compositions. Be sure to keep track of how business service candidates map to underlying application service candidates during this exercise.

• Step 12: Revise application service operation grouping

– Going through the motions of mapping the activity scenarios from Step 11 usually will result in changes to the grouping and definition of application service operation candidates. It will also likely point out any omissions in applicationlevel processing steps, resulting in the addition of new service operation candidates and perhaps even new service candidates.

• Optional Step: Keep an inventory of service candidates

– So far, this process has assumed that this is the first time you are modeling service candidates. Ideally, when going through subsequent iterations of this process, you should take existing service candidates into account before creating new ones. It can, however, be tricky to look for reuse opportunities when modeling documents. This is because so much of the verbiage used to describe service and service operation candidates gets lost when this information is later translated into concrete service designs. As a result, this step is considered optional. The service-oriented design processes also provide steps dedicated to checking for reuse opportunities

WS-BPEL

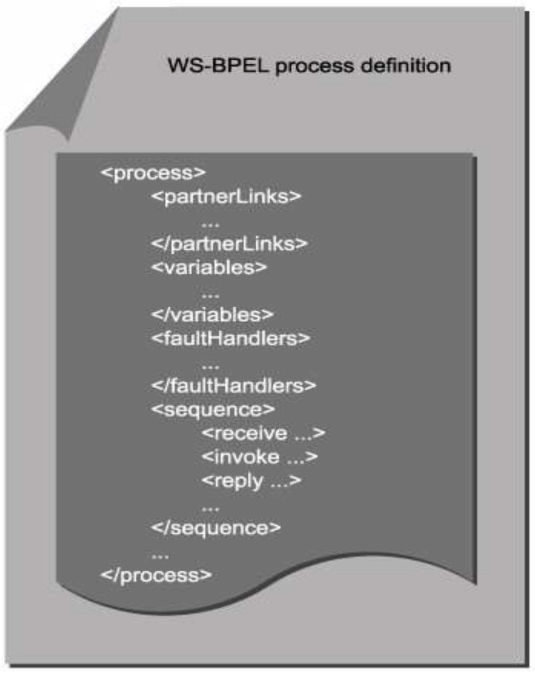
WS-BPEL language basics

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.

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.



```
<process>
     <partnerLinks>
          ...
     </partnerLinks>
     <variables>
          ...
     </variables>
     <faultHandlers>
          ...
     </faultHandlers>
     <sequence>
          <receive ...>
          <invoke ...>
          <reply ...>
          ...
     </sequence>
     ...
</process>
```

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.

A skeleton process definition.

```
<processname="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.

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).

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.

```
<partnerLinks>
<partnerLink name="client" partnerLinkType="tns:TimesheetSubmissionType"
myRole="TimesheetSubmissionServiceProvider"/>
<partnerLinkname="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**>

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.

A WSDL definitions construct containing a partnerLinkType construct. <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/"
  ...
>
  ...
  <plnk:partnerLinkType name="EmployeeServiceType" xmlns=
    "http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
    <plnk:role name="EmployeeServiceProvider">
      <portType name="emp:EmployeeInterface"/>
    </plnk:role>
  </plnk:partnerLinkType>
  ...
</definitions>
```

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.

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"/>
  ...
</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.

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.

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

```
getVariableData ('InvoiceHoursResponse', 'ResponseParameter')
getVariableData ('input','payload','/tns:TimesheetType/Hours/...')
```

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.

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

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

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.

invoke element attributes

Attribute	Description
partnerLink	This element names the partner service via its corresponding partnerLink.

portType	The element used to identify the portType element of the partner service.
operation	The partner service operation to which the process service will need to send its request.
inputVariable	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 WSBPEL variable element with a messageType attribute.
outputVariable	This element is used when communication is based on the request-response MEP. The return value is stored in a separate variable element.

The invoke element identifying the target partner service details.

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

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.

receive element attributes

Attribute	Description
partnerLink	The client partner service identified in the corresponding partnerLink construct.
portType	The process service portType that will be waiting to receive the request message from the partner service.
operation	The process service operation that will be receiving the request.
variable	The process definition variable construct in which the incoming request message will be stored.
createInstance	When this attribute is set to "yes," the receipt of this particular request may be responsible for creating a new instance of the process.

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.

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

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.

reply element attributes

Attribute	Description
partnerLink	The same partnerLink element established in the receive element.
portType	The same portType element displayed in the receive element.
operation	The same operation element from the receive element.
variable	The process service variable element that holds the message that is returned to the partner service.
messageExchange	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).

A potential companion reply element to the previously displayed receive element.

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

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.

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

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

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.

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

```
<assign>
  <copy>
    <from variable="TimesheetSubmissionFailedMessage"/>
    <to variable="EmployeeNotificationMessage"/>
  </copy>
  <copy>
    <from variable="TimesheetSubmissionFailedMessage"/>
    <to variable="ManagerNotificationMessage"/>
  </copy>
</assign>
```

.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.

The faultHandlers construct hosting catch and catchAll child constructs.

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

Other WS-BPEL elements

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

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

Element	Description
compensationHandler	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.
correlationSets	WS-BPEL uses this element to implement correlation, primarily to associate messages with process instances. A message can belong to multiple correlationSets. Further, message properties can be defined within WSDL documents.
empty	This simple element allows you to state that no activity should occur for a particular condition.
eventHandlers	The eventHandlers element enables a process to respond to events during the execution of process logic. This construct can contain onMessage and onAlarm child elements that trigger process activity upon the arrival of specific types of messages .

exit	See the terminate element description that follows.
flow	A flow 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 flow construct are defined using the child link element.
pick	Similar to the eventHandlers element, this construct also can contain child onMessage and onAlarm elements but is used more to respond to external events for which process execution is suspended.
scope	Portions of logic within a process definition can be sub-divided into scopes using this construct. This allows you to define variables, faultHandlers, correlationSets, compensationHandler, and eventHandlers elements local to the scope.
terminate	This element effectively destroys the process instance. The WS-BPEL 2.0 specification proposes that this element be renamed exit.
throw	WS-BPEL supports numerous fault conditions. Using the tHRow element allows you to explicitly trigger a fault state in response to a specific condition.
wait	The wait element can be set to introduce an intentional delay within the process. Its value can be a set time or a predefined date.
while	This useful element allows you to define a loop. As with the case element, it contains a condition attribute that, as long as it continues resolving to "true," will continue to execute the activities within the while construct.

WS-COORDINATION

WS-Coordination, which can be used to realize some of the underlying mechanics for WSBPEL 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 WSCoordination 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.

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.

A skeleton CoordinationContext construct.

```
<Envelope                                    xmlns="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:wsc="http://schemas.xmlsoap.org/ws/2002/08/wscoor"
          xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">
  <Header>
    <wsc:CoordinationContext>
      <wsu:Identifier>
        ...
      </wsu:Identifier>
      <wsu:Expires>
        ...
      </wsu:Expires>
      <wsc:CoordinationType>
        ...
      </wsc:CoordinationType>
<wsc:RegistrationService>
        ...
      </wsc:RegistrationService>
    </wsc:CoordinationContext>
  </Header>
  <Body>
    ...
  </Body>
</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 WSCoordination can insert additional elements within the CoordinationContext construct that represent values related to the execution environment. **The Identifier and Expires elements**

These two elements originate from a utility schema used to provide reusable elements. WSCoordination 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.

Identifier and Expires elements containing values relating to the header. <Envelope

```
  ...
  xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">
  ...
  <wsu:Identifier>
http://www.xmltc.com/ids/process/33342
  </wsu:Identifier>
  <wsu:Expires>
    2008-07-30T24:00:00.000
  </wsu:Expires>
  ...
</Envelope>
```

The CoordinationType element

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

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.

The CoordinationType element representing the WS-BusinessActivity protocol.

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

</wsc:CoordinationType>

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.

The CoordinationType element representing the WS-AtomicTransaction protocol.

```
<wsc:CoordinationType>
http://schemas.xmlsoap.org/ws/2003/09/wsat
</wsc:CoordinationType>
```

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.

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

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

WS-EXTENSIONS

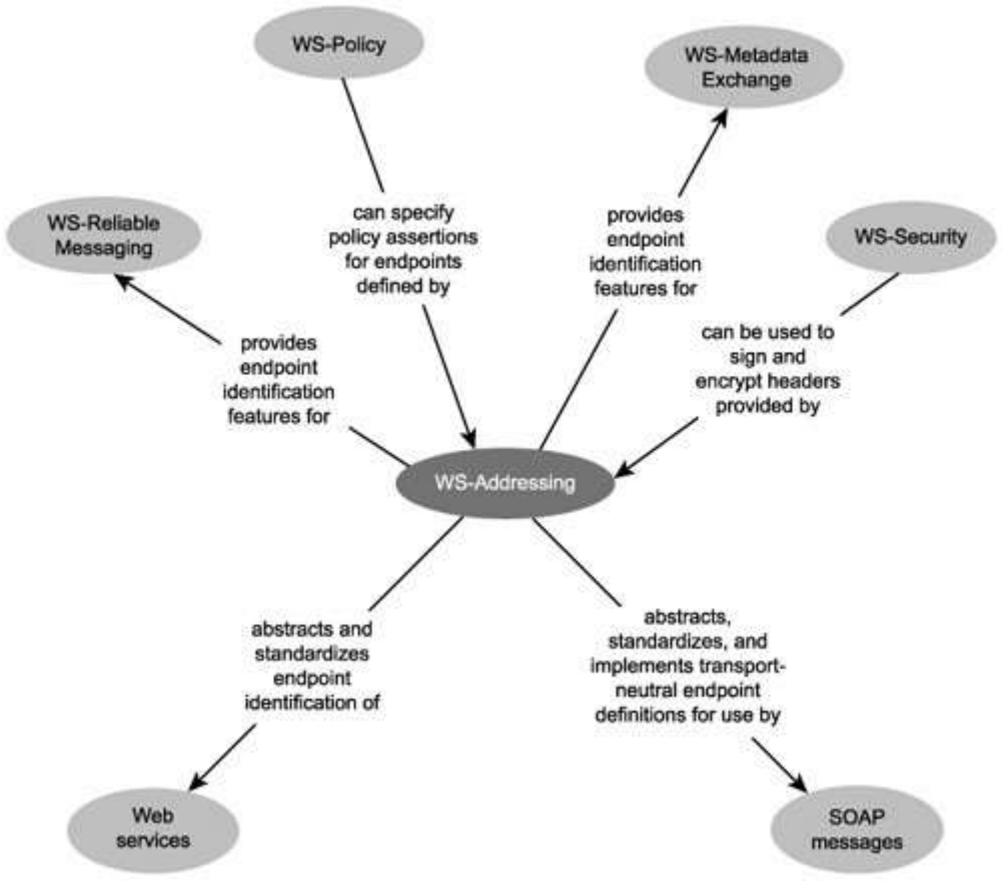
The following five key WS-* extensions:

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

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.

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


WS-Policy
WS-Metadata Exchange
WS-Reliable Messaging
can specify policy assertions for endpoints defined by
provides endpoint identification features for
WS-Security
provides endpoint identification features for
can be used to sign and encrypt headers provided by
WS-Addressing
abstracts and standardizes endpoint identification of
abstracts, standardizes, and implements transport-neutral endpoint definitions for use by
Web services
SOAP messages

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.

Element	**Description**
Address	The standard WS-Addressing Address element used to provide the address of the service. This is the only required child element of the EndpointReference element.

ReferenceProperties	This construct can contain a series of child elements that provide details of properties associated with a service instance.
ReferenceParameters	Also a construct that can supply further child elements containing parameter values used for processing service instance exchanges.
PortType	The name of the service portType.
ServiceName and PortName	The names of the service and port elements that are part of the destination service WSDL definition construct.
Policy	This element can be used to establish related WS-Policy policy assertion information.

A SOAP header containing the EndpointReference construct.

```
<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>
     22322447
    </app:sesno>
  </wsa:ReferenceParameters>
</wsa:EndpointReference>
```

Message information header elements

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

Table 5.6. WS-Addressing message information header elements

Element	Description

MessageID	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.
RelatesTo	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.
ReplyTo	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.
From	The source endpoint element (of type EndpointReference) that conveys the source endpoint address of the message.
FaultTo	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.
To	The destination element used to establish the endpoint address to which the current message is being delivered.
Action	This element contains a URI value that represents an action to be performed when processing the MI header.

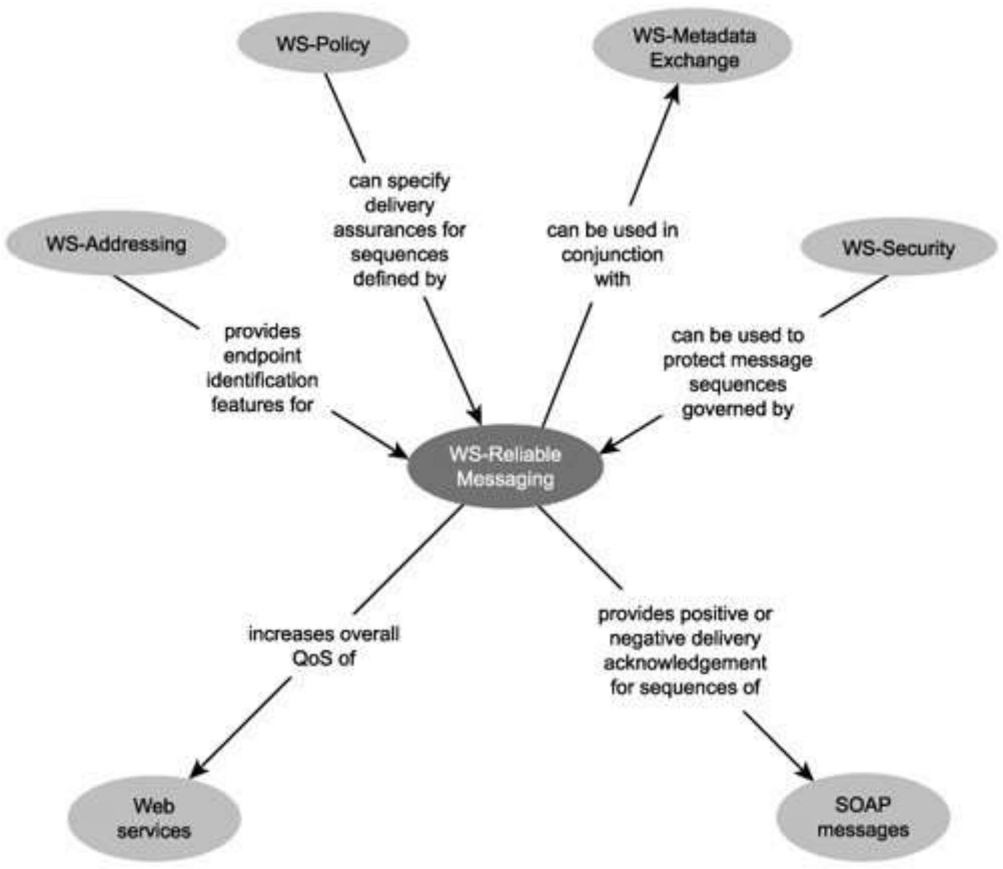
A SOAP header with WS-Addressing message information header elements, three of which contain Endpoint Reference elements.

```
<Envelope                                  xmlns="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
          xmlns:app="http://www.xmltc.com/railco/...">
  <Header>
    <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:ReferenceParameters>
        <app:sesno>
          22322447
        </app:sesno>
      </wsa:ReferenceParameters>
    </wsa:ReplyTo>
    <wsa:FaultTo>
<wsa:Address>
http://www.xmltc.com/railco/ap-err/
      </wsa:Address>
      <wsa:ReferenceProperties>
        <app:id>
unn:AFJK323llws
        </app:id>
      </wsa:ReferenceProperties>
      <wsa:ReferenceParameters>
        <app:sesno>
          22322447
        </app:sesno>
      </wsa:ReferenceParameters>
    </wsa:FaultTo>
  </Header>
  <Body>
    ...
  </Body>
</Envelope>
```

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.

How WS-ReliableMessaging relates to the other WS-* specifications discussed in this chapter.


WS-Policy
WS-Metadata Exchange
can specify delivery assurances for sequences defined by
can be used in conjunction with
WS-Addressing
WS-Security
provides endpoint identification features for
can be used to protect message sequences governed by
WS-Reliable Messaging
increases overall QoS of
provides positive or negative delivery acknowledgement for sequences of
Web services
SOAP messages

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, BaseRetransmissionInterval, InactivityTimeout, Expires, and SequenceCreation.

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.

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

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
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>
```

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.

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

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
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>
```

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.

```
<Envelope
```

```
xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
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>
```

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.

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:wsu="http://schemas.xmlsoap.org/ws/2002/07/util
ity"
```

```
xmlns:wsrm="http://schemas.xmlsoap.org/ws/2004/03/r
m">
  <Header>
    <wsrm:AckRequested>
      <wsu:Identifier>
http://www.xmltc.com/tls/seq22232
      </wsu:Identifier>
    </wsrm:AckRequested>
  </Header>
  <Body>
    ...
  </Body>
</Envelope>
```

Other WS-ReliableMessaging elements

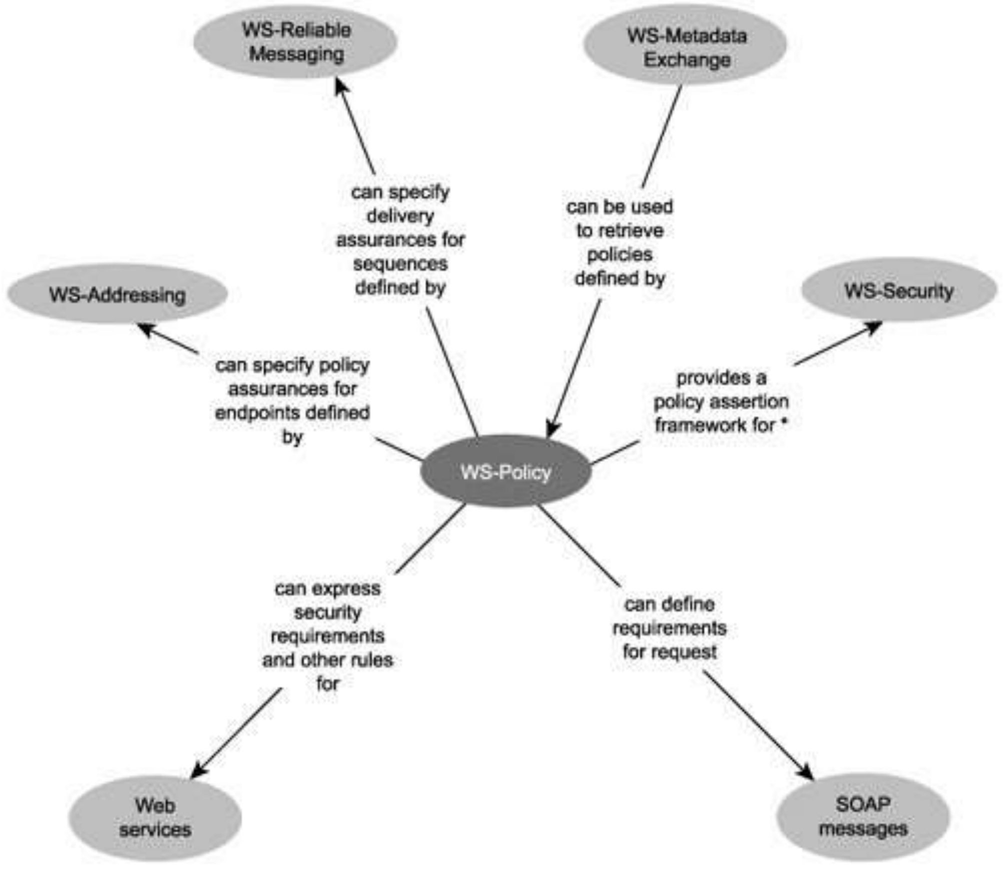
Additional WS-ReliableMessaging elements.

Element	Description
SequenceRef	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.
AcknowledgementInterval	Specifies an interval period that an RM destination can use to automatically transmit acknowledgement messages.
BaseRetransmissionInterval	An interval period used by the RM source to retransmit messages (for example, if no acknowledgements are received).
InactivityTimeout	A period of time that indicates at what point a sequence will time out and subsequently expire.
Expires	A specific date and time at which a sequence is scheduled to expire.
SequenceCreation	Sequences are generally created by the RM Source, but the RM Destination may use this element to force the creation of its own sequence.

WS-POLICY LANGUAGE

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.

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


WS-Reliable Messaging
WS-Metadata Exchange
can specify delivery assurances for sequences defined by
can be used to retrieve policies defined by
WS-Addressing
WS-Security
can specify policy assurances for endpoints defined by
provides a policy assertion framework for *
WS-Policy
can express security requirements and other rules for
can define requirements for request
Web services
SOAP messages

* A separate WS-SecurityPolicy specification provides a set of predefined policy assertions for WS-Security.

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
- ExactlyOne element

- All element
- Usage and Preference attributes
- PolicyReference element
- PolicyURIs attribute
- PolicyAttachment element

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.

The ExactlyOne element

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

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

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2002/12/policy">
  <wsp:ExactlyOne>
    <wsp:SpecVersion wsp:Usage="wsp:Required" wsp:Preference="10" wsp:URI=
                                                "http://schemas.xmlsoap.org/ws/2004/03/rm"/>
    <wsp:SpecVersion wsp:Usage="wsp:Required" wsp:Preference="1" wsp:URI=
                                                 "http://schemas.xmlsoap.org/ws/2003/02/rm"/>
</wsp:ExactlyOne>
</wsp:Policy>
```

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.

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

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2002/12/policy">
  <wsp:ExactlyOne ID="Invoice1">
    <wsp:All>
      <wsp:SpecVersion wsp:Usage="wsp:Required" wsp:Preference="10" wsp:URI=
                                                   "http://schemas.xmlsoap.org/ws/2004/03/rm"/>
      <wsp:TextEncoding    wsp:Usage="wsp:Required"    Encoding="iso-8859-5"/>
</wsp:All>
    <wsp:All ID="Invoice2">
      <wsp: SpecVersion wsp:Usage="wsp:Required" wsp:Preference="1" wsp:URI=
                                                   "http://schemas.xmlsoap.org/ws/2003/02/rm"/>
      <wsp:TextEncoding    wsp:Usage="wsp:Required"    Encoding="iso-8859-5"/>
</wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

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.

Possible settings for the Usage attribute.

Attribute Value	**Description**
Required	The assertion requirements must be met, or an error will be generated.
Optional	The assertion requirements may be met, but an error will not be generated if they are not met.

Rejected	The assertion is unsupported.
Observed	The assertion applies to all policy subjects.
Ignored	The assertion will intentionally be ignored.

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.

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.

The PolicyURIs attribute

Example 5.26. Separate PolicyReference elements referencing two policy documents.

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

The PolicyURIs attribute referencing two policy documents.

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

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.

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

```
<wsp:PolicyAttachment>
  <wsp:AppliesTo>
    <wsa:EndpointReference xmlns:emp="http://www.xmltc.com/tls/employee">
      <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:PolicyReference URI= "http://www.xmltc.com/EmployeePolicy.xml"/>
</wsp:PolicyAttachment>
```

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:

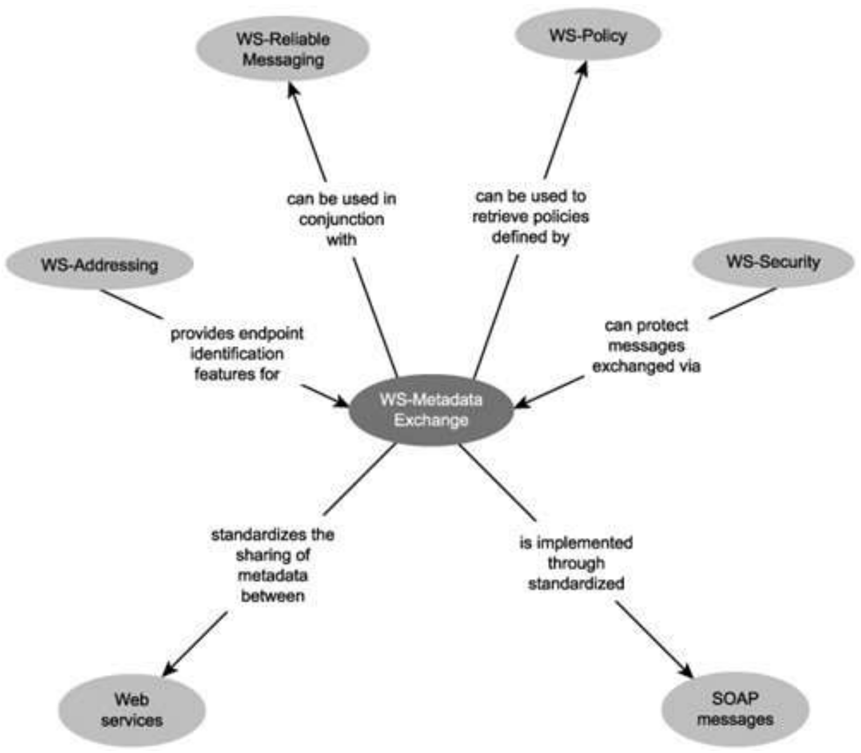
- Policy assertions can be incorporated into WSDL definitions through the use of a special set of policy subjects that target specific parts of the definition structure. A separate UsingPolicy element is provided for use as a WSDL extension.
- WS-ReliableMessaging defines and relies on WS-Policy assertions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

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.


WS-Reliable Messaging
WS-Policy
can be used in conjunction with
can be used to retrieve policies defined by
WS-Addressing
WS-Security
provides endpoint identification features for
can protect messages exchanged via
WS-Metadata Exchange
standardizes the sharing of metadata between
is implemented through standardized
Web services
SOAP messages

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.

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).

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.

```
<Envelope            xmlns="http://www.w3.org/2003/05/soap-envelope"
             xmlns:wsa=
             "http://schemas.xmlsoap.org/ws/2004/08/addressing"
             xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex">
  <Header>                                               <wsa:Action>
http://schemas.xmlsoap.org/ws/2004/09/mex/GetMetadata/Reques
t
     </wsa:Action>
<wsa:To>
http://www.xmltc.com/tls/ap
1/
     </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>
```

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.

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/XMLSchem
a      </wsx:Dialect>
  </wsx:GetMetadata>
</Body>
```

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.

The Identifier element added to specify the XSD schema's target namespace.

```
<Body>
  <wsx:GetMetadata>
<wsx:Dialect>
http://www.w3.org/2001/XMLSchem
a
    </wsx:Dialect>                    <wsx:Identifier>
http://www.www.xmltc.com/tls/schemas/ap1/schema
s
    </wsx:Identifier>
  </wsx:GetMetadata>
</Body>
```

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).

A GetMetadata response message returning the contents of an entire WSDL definition, along with a pointer to the associated XSD schema.

```
<Envelope                                   xmlns="http://www.w3.org/2003/05/soap-envelope"
          xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
          xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex">
  <Header>
    <wsa:Action>
      http://schemas.xmlsoap.org/ws/2004/09/
mex/GetMetadata/Response
    </wsa:Action>
    <wsa:RelatesTo>
      23492372938
    </wsa:RelatesTo>
<wsa:To>
http://www.xmltc.com/railco/inv
1
    </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.www.xmltc.com/tls/ap1/schemas
          </wsa:Address>
        </wsx:MetadataReference>
      </wsx:MetadataSection>
```

```
   </wsx:Metadata>
  </Body>
</Envelope>
```

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.

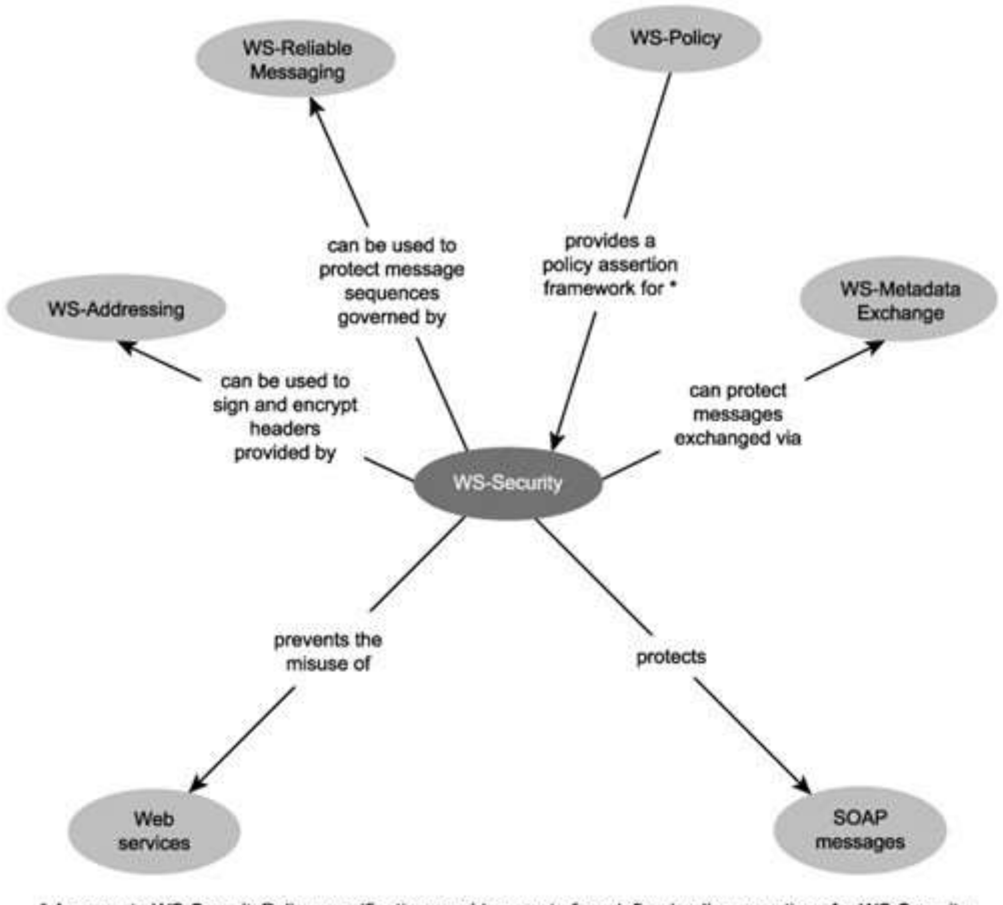
A Get message SOAP header identified by the Action element value. The resource being requested is targeted in the To element.

```
<Header>
  <wsa:Action>
http://schemas.xmlsoap.org/ws/2004/09/mex/Get/Request
  </wsa:Action>
  <wsa:MessageID>
    23492372938
  </wsa:MessageID>
  <wsa:ReplyTo>
<wsa:Address>
http://www.xmltc.com/railco/sub
1
    </wsa:Address>
  </wsa:ReplyTo>                    <wsa:To>
http://www.www.xmltc.com/tls/schemas/n
ot1
  </wsa:To>
</Header>
```

WS-SECURITY

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.

How WS-Security relates to the other WS-* specifications discussed in this chapter.


WS-Reliable Messaging
WS-Policy
can be used to protect message sequences governed by
provides a policy assertion framework for *
WS-Addressing
WS-Metadata Exchange
can be used to sign and encrypt headers provided by
can protect messages exchanged via
WS-Security
prevents the misuse of
protects
Web services
SOAP messages

* A separate WS-SecurityPolicy specification provides a set of predefined policy assertions for WS-Security.

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.

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.

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.

The BinarySecurityToken element (WS-Security)

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

The SecurityTokenReference element (WS-Security)

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

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

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Header>
    <wsse:Security xmlns:wsse= "http://schemas.xmlsoap.org/ws/2002/12/secext">

<wsse:UsernameToken>
<wsse:Username>
rco-3342
        </wsse:Username>
        <wsse:Password Type="wsse:PasswordDigest">
          93292348347
        </wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </Header>
  <Body>
    ...
  </Body>
```

```
</Envelope>
```

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.)

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

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Header>
    <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext">
      <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>
  <Body>
    ...
  </Body>
</Envelope>
```

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.

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>
```

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>
  <Date>
    07.16.05
  </Date>
</InvoiceType>
```

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.

XML-Signature elements

Element	**Description**
CanonicalizationMethod	This element identifies the type of "canonicalization algorithm" used to detect and represent subtle variances in the document content (such as the location of white space).
DigestMethod	Identifies the algorithm used to create the signature.
DigestValue	Contains a value that represents the document being signed. This value is generated by applying the DigestMethod algorithm to the XML document.
KeyInfo	This optional construct contains the public key information of the message sender.
Signature	The root element, housing all of the information for the digital signature.
SignatureMethod	The algorithm used to produce the digital signature. The digest and canonicalization algorithms are taken into account when creating the signature.
SignatureValue	The actual value of the digital signature.
SignedInfo	A construct that hosts elements with information relevant to the SignatureValue element, which resides outside of this construct.

Reference	Each document that is signed by the same digital signature is represented by a Reference construct that hosts digest and optional transformation details.

A SOAP message header containing a digital signature.

```
<Envelope ...>
  <Header>
    <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext">
      <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>
          </Reference>
        </SignedInfo>
        <SignatureValue>
          9879DFSS3=
        </SignatureValue>
        <KeyInfo>
          ...
        </KeyInfo>
      </Signature>
    </wsse:Security>
  </Header>
  <Body>
    ...invoice document with total exceeding $30,000...
  </Body>
</Envelope>
```

SOA support in J2EE

• The Java 2 Platform Enterprise Edition (J2EE) is one of the two primary platforms currently being used to develop enterprise solutions using Web services.

• Platform overview

– The Java 2 Platform is a development and runtime environment based on the Java programming language.

– It is a standardized platform that is supported by many vendors that provide development tools, server runtimes, and middleware products for the creation and deployment of Java solutions.

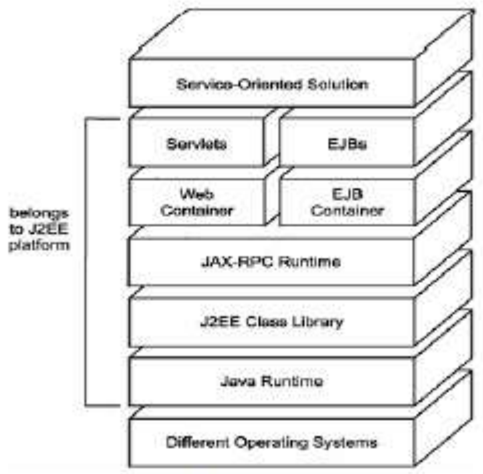
– The Java 2 Platform is divided into three major development and runtime platforms,.

• J2SE -The Java 2 Platform Standard Edition is designed to support the creation of desktop applications,

• J2ME - the Micro Edition (J2ME) is geared toward applications that run on mobile devices.

• J2EE -The Java 2 Platform Enterprise Edition is built to support large-scale, distributed solutions.

– The Servlets + EJBs and Web + EJB Container layers (as well as the JAX-RPC Runtime) relate to the Web and Component Technology layers established



Relevant layers of the J2EE platform as they relate to SOA.

Three of the more significant specifications that pertain to SOA are listed here:

– Java 2 Platform Enterprise Edition Specification

• This important specification establishes the distributed J2EE component architecture and provides foundation standards .

\

– **Java API for XML-based RPC (JAX-RPC)**

• This document defines the JAX-RPC environment and associated core APIs. It also establishes the Service Endpoint Model used to realize the JAX-RPC Service Endpoint.

– Web Services for J2EE

  • The specification that defines the J2EE service architecture and clearly lays out what parts of the service environment can be built by the developer, implemented in a vendor-specific manner, and which parts must be delivered according to J2EE standards.

• Architecture components

  – J2EE solutions inherently are distributed and therefore componentized.

  – The following types of components can be used to build J2EE Web applications:

    • Java Server Pages (JSPs)

      – Dynamically generated Web pages hosted by the Web server.

      – JSPs exist as text files comprised of code interspersed with HTML.

    • Java Servlets

      – These components also reside on the Web server and are used to process HTTP request and response exchanges.

• **Enterprise JavaBeans (EJBs)**

    – The business components that perform the bulk of the processing within enterprise solution environments.

    – They are deployed on dedicated application servers and can therefore leverage middleware features, such as transaction support.

Struts

  – An extension to J2EE that allows for the development of Web applications with sophisticated user-interfaces and navigation.

  – Struts provides its own Controller component and integrates with other technologies to provide the Model and the View. For the Model,

  • Struts can interact with standard data access technologies, like JDBC and EJB, as well as most any third-party packages, like Hibernate, iBATIS, or Object Relational Bridge.

• For the View, Struts works well with JavaServer Pages, including JSTL and JSF, as well as XSLT, and other presentation systems.

– Runtime environments

– J2EE supply two types of component containers that provide hosting environments geared toward Web services-centric applications that are generally EJB or servlet-based.

EJB container

– This container is designed specifically to host EJB components, and it provides a series of enterprise-level services that can be used collectively by EJBs participating in the distributed execution of a business task.

– Examples of these services include

– transaction management,

– concurrency management,

– operation-level security,

– object pooling.

Web container

– A Web container can be considered an extension to a Web server and is used to host Java Web applications consisting of JSP or Java servlet components.

– Web containers provide runtime services geared toward the processing of JSP requests and servlet instances.