The goal of web services is to enable communication between different software and hardware systems. These systems typically differ in both their hardware and software configurations. These differences have been overcome through the definition of standard protocols, such as those employed in building web services. Occasionally, incompatibility issues arise even when using these standard protocols, which can lead to interoperability problems. This tutorial, Part 6 of the "Understanding web services specifications" series, explains the nature and origin of web service interoperability problems. This tutorial also introduces you to the WS-I Basic Profile, which is a set of guidelines web services should adhere to in order to achieve optimum interoperability.

Before you start

In this tutorial you learn about Web Services Interoperability, or WS-Interoperability. This tutorial is for programmers who build web services and want to ensure that their services are designed to work well for the largest number of potential users. Understanding the real world interoperability problems that affect web services and the best practices you can apply to avoid them can make the difference between a web service that is easy for users to work with and one that is fraught with problems.

In order to follow along with this tutorial, you should have a basic understanding of Simple Object Access Protocol (SOAP) and related technologies, such as WSDL. The samples in this tutorial use Java™ and the Apache Axis2 SOAP toolkit. Read the first five tutorials in this series, especially Part 1 and Part 2, for the best understanding.

About this series

This tutorial series teaches the basic concepts of web services by following the exploits of the fictional newspaper, Daily Moon, as the staff uses web services to create a workflow system to increase productivity in this competitive environment.
Part 1 starts simply, explaining the basic concepts behind web services and showing you how to use SOAP, the specification that underlies most of what is to come, connecting the classifieds department with the Content Management System.

Part 2 takes things a step further, explaining how to use Web Services Description Language (WSDL) to define the messages that web services produce, enabling the team to more easily create services and the clients that connect to them.

Part 3 finds the team with a number of services in place and a desire to locate them easily. In response, Universal Description, Discovery, and Integration (UDDI) provides a searchable registry of available services as a way to publicize its own services to others.

In Part 4, Rudy, publisher of the Daily Moon, decides that the paper needs to institute better security procedures for web services that access their internal systems.

Part 5 shows the changes the teams needed to make in order to access those newly secured services.

This Part 6 of the series is about building and verifying interoperable web services. The newspaper staff members are already familiar with the importance of reaching the largest possible audience, so they decide to analyze their web services to ensure that anyone who wants to use them will have an easy time doing so.

Finally, Part 7 shows how to use Business Process Execution Language (WS-BPEL) to create complex applications from individual services.

About this tutorial

In Part 6 of the web services tutorial series, the Daily Moon staff members work to ensure that their web services are as interoperable as possible. The staff members are confident that the services satisfy the business requirements they started with, but they are not confident that the services will work for everyone who wants to use them. Interoperability problems were once commonplace with web services because of the large scope of the original SOAP and WSDL specifications, and because of the interpretation the specifications left to developers, which led to incompatible implementations. These issues have been solved through experience and by adopting a set of best practices as specified in the WS-I Basic Profile.

For the Daily Moon (and most organizations), the pressure is on to guarantee the largest possible circulation of the organization's web services, so organizations research the issue to make sure their web services are free of interoperability issues.

This tutorial teaches you about web service interoperability issues with a learn-by-doing approach. The previous parts of this series showed you how to incorporate a web service into your own software and how to build your own services. This sixth installment builds on the previous ones and focuses on what you can do to make sure that the web services you produce are as easy and as trouble-free as possible for the programmers who will use them.
Prerequisites

The code used in this tutorial is not specific to any particular programming language or environment. The examples provided are the same ones used throughout this tutorial series. To follow along with the examples, you need the following software installed:

**Java 2 Standard Edition version 1.4.2 or higher** -- All of these tools are Java-based, as are the services and clients you build in this tutorial.

**Apache Axis2 version 1.0** -- Axis2 is a full-featured SOAP toolkit that provides implementations of several web service APIs including SOAP and WSDL. A toolkit like Axis2 is invaluable when it comes to web service development. Toolkits of similar scope exist for other programming languages and environments. The Axis project at Apache has a long history and originated with an IBM effort called SOAP4J.

**Apache Geronimo or another application server** -- This tutorial series uses the Apache Geronimo J2EE server throughout (which is the basis for IBM WebSphere® Community Edition server). You can use other application servers instead, but Geronimo is simple, lightweight, and freely available, so it is a good choice for getting up-and-running quickly.

Overview

This tutorial begins with a quick overview of the web services landscape and an introduction to interoperability problems and their origins.

Understanding the need for web services

For a generation now, programmers have been building software programs that talk to each other. Today, every programming language comes with software libraries that make network programming possible, easy, and predictable.

Not only are there low-level tools for sending bits from one computer to another, but there are also high-level protocols like HTTP that provide a common set of rules that make it possible for two programs to converse, even though they were built independently by different people and are deployed on different machines and systems software. Imagine what happens, for example, when you click on a link in your Web browser and load a Web page you have never seen before. Through the adoption of a few standards (HTTP, HTML, TCP/IP), your Web browser is able to call another computer system, fetch a Web page, and render it for you. This type of unrehearsed and dynamic software networking is powerful. Web services are an attempt to extend this power to developers of complicated enterprise systems.

Seeing the complexity and the challenge

Just as there are differences in the way Microsoft® Internet Explorer and Mozilla Firefox interpret some Web pages, there are differences in how two web service programs interpret the SOAP messages they exchange with one another. As you might imagine, these differences can result in serious problems that are difficult to resolve. The best Web designers spend plenty of time and
energy understanding how their designs work on an array of different browsers, and they test like crazy to make sure their HTML, CSS, and Javascript work properly on each one. There are standards and specifications that formalize HTML, CSS, and Javascript, but these specifications are not immune to errors and ambiguity. The result is that these specifications have been interpreted in various ways, and browsers do not all behave in the same way. The same is true with the specifications that define the various web services technologies, such as SOAP, WSDL, UDDI, XML Schema, and HTTP.

Despite the simplicity that inspired web services, building web services today demands know-how and patience, as this tutorial series illustrates. To build web services today, you need to be a solid programmer familiar with at least one programming language and environment. You need to be familiar with computer networking and with a family of standard technologies used in web services today, such as XML, SOAP, WSDL, or XML Schema. You also need to understand a host of practical techniques needed to use these technologies -- and there are many.

Imagine that you put yourself through this substantial training exercise (you can call it a career) and attempt to connect your software to another piece of software using web services only to find that it does not work. Not only is this scenario incredibly frustrating, it is also quite common (usually when you are on a tight development schedule).

**Adhering to best practices and discipline**

To help catalog and remedy these interoperability issues, a group of engineers known as the Web Services Interoperability Organization (WS-I) have produced a set of recommendations for web services developers that, if followed, can help produce web services and web service consumers that work well together. This set of recommendations is known as the WS-I Basic Profile. If you are a web services developer, you should add this acronym to your resume. Understanding web service interoperability issues and proactively designing services to avoid them has become an important responsibility for web service developers, just as designing for cross-browser compatibility has become an important responsibility for Web designers.

There is no silver bullet approach to do away with all interoperability issues. Distributed computing in a heterogeneous environment is a complex activity, and there are bound to be interoperability problems. The majority of these can be avoided, however, with discipline and an adherence to the best practices described in the WS-I Basic Profile.

**Following the story so far**

This series follows the staff of the fictional *Daily Moon* newspaper as it moves many of its everyday operations to a web services-based system. In Part 1, the Classifieds Department learned about SOAP by interacting with the Content Management System, and in Part 2, they created their own service, and defined it using Web Services Description Language (WSDL). Then in part 3, they learned how to interact with a UDDI registry, and also how to find data within one, which ultimately led them to create one for the company to enable other organizations to interact with the *Daily Moon*. Because Rudy insisted that his colleagues Gene and Francis provide a way to prevent unauthorized access to their systems in Part 4, Gene and Francis had to implement WS-Security.
for their web services. In Part 5, Rudy realized he needed to define policies for web services to enforce that clients accessing the web services of the Daily Moon do so in a prescribed manner, ensuring the security that was built into it back in Part 4 of this series.

Now, continuing the story of the Daily Moon staff, this tutorial describes the staff members' experiences as they prepare to release their classified web service to the public. Phil is handling this release, and he wants to make sure that the WSDL document and the service it describes are as accessible and trouble-free as possible for the people who decide to use it to send classified ads (and revenue) to the Daily Moon. Phil has used other web services in the past and is well aware of the seriousness of interoperability problems.

**Interoperability overview**

This section describes the plan to understand interoperability problems, how to test for them, and a few common terms about web services.

**Understanding the problem**

The first challenge is to understand the problem of interoperability and its importance in the world of web services. Web services today are built using a complex family of related technologies and standards, which make for many sources of interoperability issues. This tutorial covers the most common problems and their origins.

This tutorial then introduces the WS-I Basic Profile, explaining its goals, language, and many of the specific requirements it details for interoperable web services.

**Testing**

Next, follow along as Phil inspects the Daily Moon classifieds and banking web services for possible interoperability problems. The WS-I organization produces a set of testing tools that can be used to evaluate the interoperability of a service. This tutorial shows how these tools work and how they enable a test-driven approach to building web services.

**Learning the terminology**

When talking about web services, it is best to describe things in terms of providers and consumers. Here are common definitions for both terms:

- **Consumer**: A consumer is responsible for making requests of a service that a provider implements.
- **Provider**: A provider is responsible for listening for and processing consumer service requests.

**Understanding interoperability and its importance**

In this section, this tutorial presents the issue of interoperability in detail and describes the root causes of interoperability problems. You can see the case for interoperable web services and why striving for the highest degree of interoperability is important.
Learning about messaging and interoperability

You have learned that web services are based on XML messages moving to and from a web service provider. You have also seen that these messages have a simple structure: envelope, header, and payload. In practice, there is quite a bit more to these messages and quite often one program produces a SOAP message and sends it to another program that has trouble interpreting it. This disagreement between programs is the root of most interoperability problems.

Even if you have built SOAP web services before, chances are that you have never read the SOAP and WSDL specifications. Many tools exist that make it unnecessary to know SOAP and WSDL in detail to use or build web services. You can generate Web service consumers and skeletal web service provider implementations by feeding a WSDL document to a code generator like the one provided by Axis2 -- all of the heavy lifting is done for you. Sometimes this process works well, but often these tools fail to generate usable programs, or they succeed, but they result in stubs and skeletons that do not work well with other programs.

In some dynamic programming languages, such as Python, libraries exist that interpret WSDL documents on the fly, essentially building service stubs on demand and making it even easier to use SOAP without having to dive into the SOAP spec. This is how things should work, but they don't always.

Avoiding ambiguous specifications

Technical specs are usually very clear and unambiguous. They need to be clear so that readers understand the specifics of what they are working on. However, in spite of best efforts, the SOAP and WSDL specifications contain subtly confusing examples and language that forced developers to use their best judgment and interpret for themselves where they found the texts to be unclear. This led to several interoperability issues as software implementations differed in the message formats they generated and expected. These interoperability problems undermine the spirit of SOAP and web services, which aim to help programs communicate with each other.

Solving ambiguity

SOAP is an acronym for Simple Object Access Protocol, which has been controversial since the first version of the specification, because most people agreed it was not really simple to understand or use. Aside from the ambiguity and errors, the SOAP specification is complicated and too broadly scoped to give rise to the interoperability one would hope to find in the world of web services. For example, SOAP provides two different service styles (rpc and document) and two use or encoding types (literal and encoded). The choice of service and encoding styles has a direct effect on the format of the messages understood by the web services. You can mix and match service and encoding styles, which create the following possibilities:

- rpc/literal
- rpc/encoded
- document/literal
- document/encoded

In addition, document style messages can use a pattern called wrapped, which also affects the resulting message body. It is nice for a software developer to have options, but it is not easy...
to decide which scheme to use. Plenty of discussion and confusion exists when it comes to choosing between the five available options. The WS-I Basic Profile takes the list of possibilities and effectively cuts it in half by recommending that the encoded message style not be used:

"The Profile prohibits the use of encodings, including the SOAP encoding.

R2706 A wsdl:binding in a DESCRIPTION MUST use the value of "literal" for the use attribute in all soapbind:body, soapbind:fault, soapbind:header, and soapbind:headerfault elements."

The encoding option was removed because it was a common source of interoperability problems: removing it helps narrow the variety of message formats web service software will have to deal with, and this certainly helps to simplify and encourage more interoperability.

Using interoperable SOAP

SOAP web services have received some criticism in recent years. Developers who have used SOAP and the related family of technologies that go along with might find it hard to disagree that it is a complicated technology and that interoperability challenges are especially frustrating. SOAP is definitely powerful; but simple, it is not. In fact, since the 1.2 version of the SOAP specification (06/2003), it is no longer officially called SOAP in response to the many complaints that the acronym is inaccurate.

Today, there are other types of web services competing with SOAP. The most notable are web services inspired by the concept of Representational State Transfer (REST). REST is an architectural style rather than a particular technology or formal standard. RESTful web services (as they are called) do not share a common message format. The format is entirely up to the service designer. So, RESTful services by design suffer from the same interoperability problems as SOAP services. Still, many users find that RESTful services are easier to build and use. Today, many organizations deploy web services in both styles, including amazon.com, ebay.com, and google.com. Even Axis2 includes support for RESTful web services. Both approaches have their merits, and they will be with us for some time to come.

SOAP's common message format is its most important feature. When things work as they should, it is easy to see SOAP's promise and why so much effort continually goes into making interoperable web services with SOAP. Today, with a little care and planning, you can avoid the common interoperability issues that plagued early SOAP web service implementations.

Next, take a look at the WS-I Basic Profile.

Introducing the WS-I Basic Profile

The Web Services Interoperability Organization (WS-I for short) is an industry organization with members including IBM, Microsoft, Intel®, Oracle, SAP, Hitachi, and many other leading companies. The goal of the WS-I is to work toward achieving web service interoperability "across platforms, operating systems, and programming languages." The WS-I is a group of experts who provide advice to web service developers to help them produce interoperable web service software. The WS-I is best known for the "Basic Profile", which is a set of implementation
guidelines aimed at avoiding interoperability problems. These guidelines are essentially an invaluable set of best practices.

**Addressing its audience**

The WS-I Basic Profile is another tough read, though it is more concise than the SOAP and WSDL specifications. Here is a small bit from the Basic Profile’s text that gives you a peek into how complex web services have become today:

"WSDL 1.1 is unclear as to which schema target namespaces are suitable for QName references from a WSDL element. The Profile allows QName references from WSDL elements both to the target namespace defined by the xsd:schema element, and to imported namespaces. QName references to namespaces that are only defined through a nested import are not allowed."

Most of the Basic Profile is easier to follow than the above paragraph, but it is important to keep in mind that it is a technical specification that covers over 200 common interoperability problems, some of which it blames on ambiguous language in other technical specifications.

Specifications like SOAP, WSDL, and the WS-I Basic Profile are written mostly for tools developers, such as the folks at Apache who produce Axis2. For developers building web services, much of the Basic Profile does not have a direct effect on the way they work, but it is a good idea to have a sense for what is in the Basic Profile. There are some parts of it to which you will want to pay close attention. For example, the Basic Profile contains quite a bit of information about how WSDL documents should look.

**Exploring the goals of the WS-I Basic Profile**

The Basic Profile comes with a set of *guiding principles* that help to illustrate its goals. These principles are reproduced below (with some commentary). Listing 1 gives you a sense of the scope and tone of the Basic Profile.

**Listing 1. WS-I guiding principles**

<table>
<thead>
<tr>
<th>No guarantee of interoperability. It is impossible to completely guarantee the interoperability of a particular service. However, the Profile does address the most common problems that implementation experience has revealed to date.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restriction vs. relaxation. When amplifying the requirements of referenced specifications, the Profile may restrict them, but does not relax them (for example, change a MUST to a MAY).</td>
</tr>
</tbody>
</table>

The Basic Profile *tightens up* loose items in other specifications, but it doesn’t ever loosen them up.

Listing 2 is another example of the Basic Profile’s philosophy of *tightening up* big specifications, as in the literal and encoded example mentioned previously.
Listing 2. More WS-I guiding principles

Multiple mechanisms. If a referenced specification allows multiple mechanisms to be used interchangeably, the Profile selects those that are well-understood, widely implemented, and useful. Extraneous or underspecified mechanisms and extensions introduce complexity and therefore reduce interoperability.

The additional principles in Listing 3 are worth reading because they help set the tone for the WS-I and should leave you with an impression of the scope and value of the Basic Profile.

Listing 3. Even more WS-I guiding principles

Future compatibility. When possible, the Profile aligns its requirements with in-progress revisions to the specifications it references. This aids implementers by enabling a graceful transition, and assures that WS-I does not 'fork' from these efforts. When the Profile cannot address an issue in a specification it references, this information is communicated to the appropriate body to assure its consideration.

Compatibility with deployed services. Backwards compatibility with deployed web services is not a goal for the Profile, but due consideration is given to it; the Profile does not introduce a change to the requirements of a referenced specification unless doing so addresses specific interoperability issues.

Focus on interoperability. Although there are potentially a number of inconsistencies and design flaws in the referenced specifications, the Profile only addresses those that affect interoperability.

Conformance targets. Where possible, the Profile places requirements on artifacts (e.g., WSDL descriptions, SOAP messages) rather than the producing or consuming software's behaviors or roles. Artifacts are concrete, making them easier to verify and therefore making conformance easier to understand and less error-prone.

Lower-layer interoperability. The Profile speaks to interoperability at the application layer; it assumes that interoperability of lower-layer protocols (e.g., TCP, IP, Ethernet) is adequate and well-understood. Similarly, statements about application-layer substrate protocols (e.g., SSL/TLS, HTTP) are only made when there is an issue affecting web services specifically; WS-I does not attempt to assure the interoperability of these protocols as a whole. This assures that WS-I's expertise in and focus on web services standards are used effectively."

These principles set the scope and tone for the rest of the Basic Profile and address the experience of its authors. The Basic Profile concerns itself primarily with interoperability, and it takes a pragmatic approach to recommendations so that they are easy to adopt widely.

WS-I Service taxonomy

As alluded to in the guiding principles above, the Basic Profile breaks a service into conformance targets or artifacts about which it then makes recommendations. The following is the complete list of artifacts it covers:

- **MESSAGE** -- protocol elements that transport the ENVELOPE (such as SOAP or HTTP messages).
- **ENVELOPE** -- the serialization of the soap:Envelope element and its content.
- **DESCRIPTION** -- descriptions of types, messages, interfaces, and their concrete protocol and data format bindings, and the network access points associated with web services (such as WSDL descriptions).
- **INSTANCE** -- software that implements a wsdl:port or a uddi:bindingTemplate.
• CONSUMER -- software that invokes an INSTANCE.
• SENDER -- software that generates a message according to the protocols associated with it.
• RECEIVER -- software that consumes a message according to the protocols associated with it (such as SOAP processors).
• REGDATA -- registry elements that are involved in the registration and discovery of web services (such as UDDI tModels).

All of the Basic Profile’s recommendations are made in the context of one of the above items. The Basic Profile uses a simple set of qualifiers to communicate the importance of the recommendation being made. Listing 4 shows a sample recommendation taken from the Basic Profile.

**Listing 4. A sample WS-I recommendation**

Several versions of HTTP are defined. HTTP/1.1 has performance advantages, and is more clearly specified than HTTP/1.0.

R1141 A MESSAGE MUST be sent using either HTTP/1.1 or HTTP/1.0.
R1140 A MESSAGE SHOULD be sent using HTTP/1.1.

This is easy enough to follow. Notice that each of the requirements above has a unique identifier, which can be useful for referencing it in conversations or during testing. This next point might seem a bit tedious, but because it tries to be as clear and unequivocal as possible, the Basic Profile formalizes the terms it uses in its recommendations. These terms, and their intents, are shown in Listing 5.

**Listing 5. Basic Profile terminology**

MUST. This word, or the terms "REQUIRED" or "SHALL," means that the definition is an absolute requirement of the specification.
MUST NOT. This phrase, or the phrase "SHALL NOT," means that the definition is an absolute prohibition of the specification.
SHOULD. This word, or the adjective "RECOMMENDED," means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
SHOULD NOT. This phrase, or the phrase "NOT RECOMMENDED," means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
MAY. This word, or the adjective "OPTIONAL," means that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation that does include the option, though perhaps with reduced functionality. In the same vein, an implementation that does include a particular option MUST be prepared to interoperate with another implementation that does not include the option (except, of course, for the feature the option provides.)

Any formal specification has some necessarily elaborate terms like those in Listing 5. The definitions of these terms are important to read because every recommendation in the Basic Profile is made using them. This formality is important also because so many interoperability problems arose from ambiguity in other specifications.
Combining the WS-I Basic Profile and SOAP

This section describes some of the Basic Profile's recommendations that specifically affect SOAP. These recommendations attempt to clarify interoperability issues that might exist with SOAP envelopes, faults, and the use of HTTP.

Using SOAP envelopes

SOAP specifies a protocol based on XML messages that travel in envelopes. The Basic Profile requires that conforming web services use this scheme as defined in the SOAP specification and adds several constraints on its use. The following is a summary of the constraints the Basic Profile imposes. This is not a complete list, but it gives you a sense of the breadth and scope of the Basic Profile.

- The Profile requires that envelopes conform to the structure specified in SOAP 1.1. This seems like stating the obvious, and it is in a sense; but it also makes it clear that SOAP 1.1 messaging is the only message structure the Profile supports.
- Envelopes must not contain a Document Type Declaration (DTD) or Processing Instructions. These are cited by the Profile as sources of security vulnerabilities, processing overhead, and semantic ambiguity. In addition, envelopes should not contain the XML namespace declarations `xmlns:xml="http://www.w3.org/XML/1998/namespace` and `xmlns:xml="http://www.w3.org/XML/1998/namespace`.
- In an envelope, there must not be any sibling elements to the body element. The profile says that the interpretation of such sibling elements is unclear, so the Profile disallows them. This formalizes the notion that a SOAP message is an envelope, with a single body, containing a payload.
- As mentioned earlier, the Profile does away with encoded SOAP messages by forbidding the use of the `soap:encodingStyle` attribute. This is an important recommendation because there has been plenty of discussion on this topic, and many interoperability problems can be traced to the use of encoded XML messages.

Using SOAP faults

The Basic Profile tightens up various details with SOAP fault envelopes:

- It first defines that a fault is an envelope that has a single element child of `soap:Body` that has a single element child `soap:Fault`. Nothing else is to be interpreted as a fault.
- HTTP status codes should not be used to determine the presence of a fault. Interpret the SOAP message to determine the presence of a fault.
- When an envelope is a fault, it is only allowed to have element children for `faultcode`, `faultstring`, `faultactor`, and `detail`. These elements must not be namespace-qualified.
- For extensibility, the Basic Profile says that SOAP faults can have any number of element children appearing on the detail element.

Again, these are just highlights. Most of these recommendations do not affect web service authors directly but are important to be aware of in case you encounter interoperability problems.
Understanding SOAP and HTTP

Because HTTP is the most common and popular transport for SOAP messages and because the Basic Profile selects it as the binding to use for web services as well as recommending a few additional best practices, the following apply:

- Messages must be sent with either HTTP 1.0 or HTTP 1.1, but HTTP 1.1 is preferred and should be used whenever possible.
- HTTP request messages must use the HTTP POST method and must not use the HTTP Extension Framework, which is allowed by SOAP 1.1.
- An HTTP 2XX status code should be used to indicate the successful outcome of an HTTP request. When the response message contains an envelope that is not a fault, status code 200 should be used. 200 or 202 should to indicate a successful outcome when there is no SOAP envelope resulting from the HTTP request.
- The value of the SOAPAction field in the HTTP header used with a request message must be a quoted string. This seems like a small detail, but it has an effect on WSDL, which should look like one of the following examples:
  - `<soapbind:operation soapAction="someAction" />`. This results in an HTTP header that says SOAPAction="someAction"
  - `<soapbind:operation soapAction="" />`. This results in an HTTP header that says SOAPAction=""
  - `<soapbind:operation />`. This also results in an HTTP header that says SOAPAction=""

Learning about WS-I Basic Profile and WSDL

The Basic Profile's section of recommendations for service descriptions begins by making it a requirement that the WSDL for a service provider be available to a web service consumer upon request. This highlights the importance of WSDL as the web service contract and helps foster a web service environment that is conducive to an ad hoc use of web services. You should always be able to find a service in a registry, request its WSDL, interpret the WSDL, and use the service.

Exploring the document structure

The Basic Profile requires that service descriptions be written in valid XML version 1.0. No particular version was mandated by the WSDL or XML Schema specifications. The Basic Profile formalizes this decision to help interoperability.

The Basic Profile places restrictions on the use of the WSDL import statement, restricting its use to cases where the document being imported is a WSDL document itself. This is another important requirement because it addresses an example in the WSDL 1.1 specification, which incorrectly shows the WSDL import statement being used to import XML Schema definitions. The profile places a similar restriction on uses of the XML Schema import, stating that it should be used only to import XML Schema definitions.

When you do import one WSDL document into another WSDL document, the Basic Profile requires that the two WSDL documents be in the same namespace (the `targetNamespace` attribute on the `wsdl:definitions` element of the WSDL being imported should have the same value as the
namespace attribute on the wsdl:import element in the WSDL doing the importing). This essentially restricts imports so that they bring into the parent namespace items that are already defined to be in that namespace, disallowing what has come to be known as the coercion of namespaces.

The Basic Profile recommends that items in a WSDL document appear in a specific order. WSDL import elements must precede all other elements from the WSDL namespace except for wsdl:documentation. The WSDL types elements must precede all other elements from the WSDL namespace except wsdl:documentation and wsdl:import. The WSDL documentation element can appear as the first child element of wsdl:import, wsdl:part, and wsdl:definition in addition to the elements cited in the WSDL 1.1.

The profile warns against using WSDL extensions, because they can easily lead to interoperability problems.

**Introducing types**

The Basic Profile does away with the WSDL array type, which has been interpreted in various different ways. wsdl:arrayType and soapenc:ArrayType should be avoided. The old convention of naming arrays using ArrayOfXXX is discouraged. Instead, arrays should be declared using a sequence as shown in Listing 6.

Listing 6. Basic Profile-compliant array definition

```xml
<xsd:element name="MyArray1" type="tns:MyArray1Type"/>
<xsd:complexType name="MyArray1Type">
  <xsd:sequence>
    <xsd:element name="x" type="xsd:string"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

**Understanding port types**

The Basic Profile disallows name overloading in a wsdl:portType. This means that each operation must have a distinct value for its name attribute.

**Learning about bindings**

A wsdl:binding in a DESCRIPTION MUST either be a rpc-literal binding or a document-literal binding. This fact is repeated a couple of times in this tutorial already because of its important effect on message format.

A wsdl:binding in a DESCRIPTION MUST use the value of literal for the use attribute in all soapbind:body, soapbind:fault, soapbind:header, and soapbind:headerfault elements.

As you can see, the Basic Profile makes many recommendations affecting WSDL. WSDL is such an important specification used in web services. The Basic Profile WSDL recommendations listed in this section are the ones by which you are most likely to be affected. They address the most common WSDL-related interoperability issues with web services.
Studying WS-I Basic Profile usage scenarios

In a document separate from the Basic Profile, the WS-I defines a set of three usage scenarios describing three common patterns for accessing web services. The usage scenarios document shows how these three usage scenarios work and how they can be implemented using the artifacts found in web services, such as WSDL.

One of the important features of the following usage scenarios is that they are composable, which means that you can use them as building blocks to create more complex usage scenarios.

Learning from the one-way usage scenario

The one-way usage scenario is applicable in situations where a web services consumer needs to send a message to a service provider and the consumer does not require a guarantee that the message will be delivered or processed. Every attempt will be made to deliver and process the message, of course. But in the event that something goes wrong, there is no channel available to inform the consumer about the failure because of the features of this usage scenario:

- No SOAP response message from the provider is generated or expected.
- The underlying transport is not required to guarantee the delivery of the message to the provider.

Why would you ever want to use a system that comes with these major caveats? There are some situations where a potentially undelivered message does not spell disaster, such as when a message is being sent to an application log. The one-way scenario is a good choice in such a case because it is lightweight. It can contribute to the scalability of a system because the consumer does not need to wait for the request to be processed or prepare for any kind of response from the provider. The system is free to move on to something else.

The usage scenarios document goes into detail on how the one-way usage scenario is used in WSDL. It is slightly different depending on whether you are using a document and literal binding or an rpc and literal binding (the only two binding types allowed under the Basic Profile). The usage scenarios are all built on top of the Basic Profile. They describe techniques that comply with the requirements of the Basic Profile.

In either case, the one-way scenario requires a single input message be sent to the provider, and no output message is specified.

Understanding the synchronous request/response usage scenario

Synchronous request/response is the most common type of usage pattern for web services and for distributed computing in general. This usage scenario is second nature to us because it accurately describes many situations we experience in the real world. For example, when you stop to ask directions to the nearest coffee shop you need to:

1. Find someone who looks like they might be able to help you.
2. Ask them if they know how to get to a nearby coffee shop.
3. Wait for them to think up a response.
4. Listen as they give you an answer.
It does not help much if you ask someone for directions and then walk away without waiting for him to respond, as you would if you employed the one-way usage scenario. In this example, it is important that you wait for the receiver to process the information you have provided them and formulate a response that you can use.

In the synchronous request/response scenario, the service receives a SOAP request message and produces a SOAP response message, for which the client anxiously waits.

Understanding the basic callback usage scenario
The basic callback scenario is for situations where the consumer is interested in receiving a result from the provider, but the consumer does not want to wait around for the result. This could be the case, for example, if the consumer asked the Provider to assemble a complicated report that would require substantial processing time. Instead of waiting for the report, the consumer gives the provider enough information so that it knows how to call the consumer back once the result is ready.

The basic callback is accomplished by composing two asynchronous callback usage scenarios, an initial request from the consumer to the provider, which ultimately results in a final response from the provider back to the consumer. Listing 7 shows how the WS-I describes the basic callback scenario.

Listing 7. Basic callback usage scenario
1. Consumer initiates the service by sending a SOAP message bound to an HTTP request to the Provider (the "initial request")
2. Provider acknowledges receipt using a SOAP message bound to an HTTP response to the Consumer (the "initial response")
3. Provider completes the exchange by sending a SOAP message bound to an HTTP request to the Consumer with the results of the initial request (the "final request" or "callback")
4. Consumer acknowledges receipt of the callback message with a SOAP message bound to an HTTP response (the "final response")

These usage scenarios are not specified by any of the web service standards; they're just common communication patterns used by web services and the applications that consume them. Cataloging these scenarios, as the WS-I has done, is important because the scenarios give developers advice on how to implement them to be interoperable. For more detail, read the usage scenarios document (see Related topics for a link). There you will find examples of the WSDL definitions used to implement these three usage patterns defined by the WS-I.

Using the WS-I test tools
The WS-I Basic Profile's authors wanted to make compliance with their requirements easy. They understand that most developers would rather focus on building software than reading yet another dry technical document. With this understanding, they created a set of test tools that developers can use to check various web service artifacts for compliance with the Basic Profile. This is a great strategy since most web service developers would love to know that their services are compliant (and therefore as devoid of interoperability issues as possible), and then move on. Most of us
are not interested in becoming interoperability experts, though we would love to benefit from their expertise.

You can download a suite of test tools directly from WS-I. This suite includes a tool called Monitor for intercepting messages on their way to and back from a web service and a tool called Analyzer for inspecting various web service artifacts (including the messages intercepted using Monitor).

Going back to the Daily Moon and Phil, whose task it is to worry about the interoperability of the newspaper’s new web services. Phil has spent substantial time researching interoperability, has found the WS-I, scanned over the Basic Profile, and downloaded the WS-I testing tools.

Using the WS-I monitor

The WS-I test tools come with a monitoring application that has the capability to snoop on the conversations that occur between a web service provider and consumer. It does this by posing as a man in the middle such that the consumer talks to the monitor application, which forwards requests to the provider without modifying them. The monitor forwards any responses from the provider back to the consumer, again without touching them in any way. The monitor keeps a log of everything it sees. It can see SOAP messages and HTTP traffic: two important components of web services that can be analyzed later for interoperability problems.

The monitor application uses an XML configuration file. The WS-I tests come with a sample monitor configuration file which, for the Java version of the suite, can be found at $WSI_HOME/java/samples/monitorConfig.xml. Phil begins by copying this sample file to his local workspace and editing it to specify a good location for the log file it will keep. He decides to create two separate configurations: one for the classified advertisements service, and another for the banking service. The resulting log files are important because they will be analyzed later by another testing tool.

The monitor configuration file also specifies the ports that it should expect consumer requests on and the provider ports it should forward to. The provider ports are fairly standard. The default listening ports should be safe for most systems. After checking these port numbers, Phil decides to stick with the defaults.

Phil makes sure that his test services are running, and he then starts the monitor using the provided batch file (Monitor.bat) and the configuration file he created for the classified advertisement service. Listing 8 shows the output that results from starting the Monitor tool.

### Listing 8. Starting the monitor tool

```
$ $WSI_HOME/java/bin/Monitor.bat -config ClassifiedSvcMonitorConfig.xml
Conformance Monitor Tool, Version: 1.0.0, Release Date: 2004-11-19
Copyright (C) 2002-2003 by The Web Services-Interoperability Organization and Certain of its Members. All Rights Reserved. Use of this Material is governed by WS-I licenses included within the documentation.

comment ..................... This configuration file is used to test the WS-I sample applications running on a single system.
logURI ..................... log.xml
replaceLog ................ true
logDuration ................. 600
timeout ..................... 3
addStyleSheet ............... <?xml-stylesheet
```
Next, Phil needs to change his test cases to use alternative service URLs so that the Monitor has a chance to intercept messages from each consumer and forward them to the running service instances. This is easily done by altering each test so that each constructs the test stub instance with an explicit service URL to override the URL included in the WSDL used to create the test case.

For the classified service test, this means creating the stub as shown in Listing 9 so that port 4040 is used instead of directly communicating with the service provider, which is on port 8080.

### Listing 9. Constructing the ClassifiedServiceStub for testing

```java
ClassifiedServiceStub stub = new ClassifiedServiceStub(null,
    "http://localhost:4040/axis2/services/ClassifiedService");
```

Next, Phil runs each test case for the classified advertisement service. It succeeds as shown in Listing 10.
Listing 10. Running the ClassifiedServiceTest

```
$ ant run.test
Buildfile: build.xml

init:

pre.compile.test:
  [echo] Stax Availability= true
  [echo] Axis2 Availability= true

compile.src:

compile.test:

run.test:
  [junit] Running com.bm.classifiedclient.ClassifiedServiceTest
  [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 2.516 sec

BUILD SUCCESSFUL
Total time: 5 seconds
```

As these tests run, Phil notices that the monitor application's log jumps as it intercepts the requests and responses from the consumer and the provider. The generated log entries are shown in Listing 11.

Listing 11. Monitor log entries

```
Log message entry [ID, Type, Sender]: 1, request, 127.0.0.1:4582
Log message entry [ID, Type, Sender]: 2, response, localhost:8080
```

Now that the monitoring process is done, Phil is ready to run the analyzer to look for interoperability problems.

Using the analyzer test tool

The analyzer test tool also uses an XML configuration file. Again, Phil begins by copying a sample file from the WS-I test suite. For the Java suite, the sample file can be found at `$WSI_HOME/java/samples/analyzerConfig.xml`. Phil makes two copies: one copy for the classified advertisements service and another copy for the banking service.

The analyzer program is capable of processing log files created by the monitor to look for messages to analyze. It is also capable of analyzing WSDL documents and other artifacts.

Phil edits the sample configuration file for his service, adding a reference to his service WSDL. The resulting configuration file is shown in Listing 12.

Listing 12. Updated WSDL reference in analyzer configuration file

```
<wsi-analyzerConfig:wsdlReference>
  <wsi-analyzerConfig:wsdlElement type="port"
    parentElementName="BankService"
    namespace="http://userguide.axis2.apache.org/BankService"> BankPort
  </wsi-analyzerConfig:wsdlElement>
  <wsi-analyzerConfig:wsdlURI>../../BankService.wsdl</wsi-analyzerConfig:wsdlURI>
</wsi-analyzerConfig:wsdlReference>
```
Phil then makes sure the analyzer's configuration knows the right location for the log file generated by our monitoring session. He chooses to log to log.xml in the current directory. This change is shown in Listing 13.

**Listing 13. Updated log file location in analyzer configuration file**

```xml
<wsi-analyzerConfig:logFile correlationType="endpoint">
   ./log.xml
</wsi-analyzerConfig:logFile>
```

At this point, all Phil has to do is run the analyzer, as shown in Listing 14.

**Listing 14. Running the analyzer tool**

```
$ Analyzer.bat -config BankSvcAnalyzerConfig.xml
Conformance Analyzer Tool, Version: 1.0.0, Release Date: 2004-11-19
Copyright (C) 2002-2003 by The Web Services-Interoperability Organization and Certain of its Members. All Rights Reserved. Use of this Material is governed by WS-I licenses included within the documentation.

verbose .................... false
Assertion Results:
   type ..................... all
   messageEntry ............ true
   assertionDescription ..... false
   failureMessage .......... true
   failureDetail .......... true
Report File:
   replace .................. true
   location ................. report.xml
Style Sheet:
   href ................... ../wsi-test-tools/common/xsl/report.xsl
type ................... text/xsl
testAssertionsFile ..........
   ../wsi-test-tools/common/profiles/SSBP10_BP11_TAD.xml
Message Log File:
   location ................. ./log.xml
correlationType .......... endpoint
WSDL Reference:
   WSDL Element:
      type ................... port
      namespace ............. http://userguide.axis2.apache.org/BankService
      name .................. BankPort
      parentElementName ..... BankService
      wsdlURI ............... bankservice/BankService.wsdl
description ............ This file contains a sample of the configuration file for the Basic Profile Analyzer, which can be used with the other sample files.

Please wait while the specified artifacts are analyzed...
Conformance report has been created.
```

Phil repeats this process for the classified advertisement service. The most interesting output from the analyzer is the report it produces. The report is an XML file that can be viewed in any browser that understands XSLT, such as Internet Explorer or Firefox.

**Reading the report**

The analyzer's report gives an overall pass-or-fail assessment for each web service it analyzes. It also provides an itemized list of the Basic Profile requirements with their own pass-or-fail
assessments. Phil loads the analyzer's report for the banking service first and notices that it has received a failing grade, as shown in Figure 1.

**Figure 1. Analyzer tool report showing a failing grade**

<table>
<thead>
<tr>
<th>Result</th>
<th>Failed</th>
</tr>
</thead>
</table>

Unfortunately, it looks like this service violates one or more of the Basic Profile's requirements. The analyzer gave it a failing assessment. In looking through the details, Phil finds one requirement that has not been met by the WSDL description for the Banking service. The report gives quite a bit of detail, as shown in Figure 2.

**Figure 2. Detailed analyzer report**

After reading this a few times and staring at the WSDL for the banking service, Phil realizes that the problem must be the namespace he has included in a soap:body element as shown in Listing 15.

**Listing 15. Input definition with namespace specified in soap:body**

```xml
<input name="BookReqMsg">
  <soap:body namespace="http://userguide.axis2.apache.org/ClassifiedService" use="literal"/>
</input>
```

He removes the namespace definition in each input in this WSDL file so they look like Listing 16, hoping that this will allow the analyzer tool to pass his service.

**Listing 16. The soap:body without a namespace**

```xml
<input name="BookReqMsg">
  <soap:body use="literal"/>
</input>
```

He then runs the analyzer tool again, reloads the report in the browser, and notices that the service is now fully compliant with the WS-I Basic Profile. Phil was lucky; this was a fairly simple change.
Phil realizes that he has changed the WSDL description for the banking service. Now he has to ensure that the WSDL still works properly with the Axis2 toolkit. To do this, he regenerates the service skeleton and the client stub classes using the WSDL2Java command, compiles everything, and tests again. Fortunately, everything works fine, and he can be sure that the banking web service is free of the interoperability problems described in the Basic Profile.

Phil repeats this entire exercise for the classified advertising web service and has a similar experience. The service fails because of BP2019. He makes the same adjustment to this service, and it passes all tests afterwards.

Using alternative tools
The testing tools from the WS-I are incredibly useful and make it easy to comply with the best practices of the Basic Profile. If you happen to use Eclipse, you also have the option of using an alternative set of tools for testing Basic Profile conformance. The Eclipse Web Tools Project comes with a variety of useful enhancements for Eclipse, including special editors for editing WSDL and XML Schema documents, syntax highlighting for many common Web programming file formats, and a set of testing tools that can analyze a WSDL document to see if it conforms to the Basic Profile. Though the testing tools from WS-I are impressive in their quality and scope, many find that the tools from the Eclipse Web Tools Project fit into their workflow much better.

Having a set of test tools enables you to validate your services in a test-driven fashion. Having these tests integrated into your development environment can make this process even more efficient. Figure 3 shows a sample of the Eclipse Web Tools plug-in. It highlights the same Basic Profile recommendation that was highlighted by the analyzer test run that Phil encountered above, but this time it is being done in Eclipse and shown exactly at the site of the problem.

Figure 3. The Eclipse Web Tools plug-in highlighting errors

Including conformance claims
Now that Phil has verified that his services pass the Basic Profile conformance tests, he decides to advertise the fact to give potential users the confidence that every effort has been made to avoid interoperability issues. The Basic Profile describes how you can make such claims. Phil follows examples to add a `wsi:Claim` element to the port definitions in his WSDL documents as shown in Listing 17.

Listing 17. A sample Basic Profile conformance claim on a WDSL port

```xml
<wSDL:port name="MyPort" binding="tns:MyBinding">
  <wSDL:documentation>
    <wsi:Claim
      conformsTo="http://ws-i.org/profiles/basic/1.1" />
  </wSDL:documentation>
</wSDL:port>
```
Summary

This tutorial introduced you to the concept of web services interoperability and provided you with a simple set of practices you can adopt to take advantage of substantial expertise in avoiding interoperability problems. Interoperability is an important issue, and it is an issue that each web services developer should concern himself with. You learned about the WS-I Basic Profile and the automated tests you can run on your services to ensure that they are free of common interoperability problems.

Web services continue to grow in popularity and, thanks to the work of organizations like the WS-I, they get easier to use as time passes and more interoperability problems are discovered, documented, and avoided. Distributed programming on the scale that web services enable will always come with a fair degree of challenge and complexity. Despite the early interoperability challenges that web service developers encountered, SOAP-based web services continue to mature as developers acquire experience and awareness about interoperability issues and how developers should design and test their software to avoid them.
### Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part 6 source code</td>
<td>ws-interoperabilitycode.zip</td>
<td>172B</td>
</tr>
</tbody>
</table>
Related topics

• RSS feed for this series: Request notification for the upcoming articles in this series. (Find out more about RSS feeds of developerWorks content.)
• Visit the WS-I organization home page.
• Get the WS-I Basic Profile 1.1 - Second Edition.
• See Interoperability at the SOAP message level: A WSDL design case study (developerWorks, July 2006) for an article that shows you how to define a WSDL document to achieve interoperability at the Simple Object Access Protocol (SOAP) message level.
• Check out Which style of WSDL should I use? (developerWorks, May 2005) to help you determine which combination of style and use to use in WSDL binding.
• See First look at the WS-I Usage Scenarios (developerWorks, January 2003) for an article that summarizes the details of the WS-I Usage Scenarios document.
• See Keep up with web service styles (and uses) for an article about controlling SOAP formats to avoid interoperability issues.
• Read the Introduction to XML tutorial (developerWorks, August 2002) to get a good grounding in XML.
• Download Java 2 Standard Edition version 1.4.2 or higher.
• The IBM SOA Web site offers an overview of SOA and how IBM can help you to get there.

Trademarks (www.ibm.com/developerworks/ibm/trademarks/)