

Services-based enterprise integration patterns made easy, Part 3: Web services and registry

Skill Level: Intermediate

[Dr. Waseem A. Roshen, PhD \(waroshen@us.ibm.com\)](mailto:waroshen@us.ibm.com)
IT Architect
IBM

10 Apr 2008

[Part 1](#) and [Part 2](#) of this [series](#) covered the basic concepts necessary to develop services-based integration patterns. This article, the third in the series, and the upcoming Part 4 further develop these ideas so the services-based integration patterns become full-blown services-based patterns. This article in particular deals with the components that are together commonly referred to as *Web services*, which were originally designed for services that can be accessed over the Internet. You'll also see that many of the Web services components can be used with services that don't use the Internet and that only require a network connection.

Introduction

In the first two articles in this series you mastered the basic concepts. Now you jump into Web services, which define standards to deal with the *heterogeneity problem*. This problem refers to the fact that in the IT infrastructure of a typical large enterprise, there's usually more than one technology used to integrate applications, and it's impossible to impose enterprise-wide standards in this type of environment.

There are usually several different kinds of technological heterogeneity in a large enterprise, including:

- **Middleware heterogeneity:** In a large enterprise, there's usually more than one type of middleware being used. The two most common types are application servers and message-oriented middleware (MOM). There's also *brand heterogeneity*, which requires support for different brands of application servers and MOMs.

- **Protocol heterogeneity:** This heterogeneity refers to the different transport protocols being used to access the services offered by various applications. Examples of such protocols include Internet Inter-ORB Protocol (IIOP), Java™ Remote Method Protocol (JRMP), HTTP, and HTTPS.
- **Synchrony heterogeneity:** There's almost always a need to support both synchronous and asynchronous interactions between applications. Also, there's often a need for callback methods as well as publish and subscribe methods.
- **Protocol mismatch:** Related to the heterogeneity of communication protocols is the problem that different applications want to communicate with each other using incompatible protocols. For example, application A might want to communicate with application B using HTTP. However, for application B the suitable protocol might be IIOP. In such cases, there's a need for protocol transformation so that application A can communicate with application B.
- **Diversity of data formats:** There's a diversity of the data format being exchanged. Most of the time the data is dependent upon the middleware being used.
- **Diversity of interface declarations:** There were also large differences in the way the services interfaces were being declared and used to invoke the service. Thus, the way interfaces were declared in, for example, Common Object Request Broker Architecture (CORBA) and Java Remote Method Invocation (RMI) were different.
- **No common place for service lookup:** There was a lack of a common place to look up services to deal with the diversity of the services in a large enterprise.

Another common problem is that as soon as a new version of the provider software becomes available, the consumer applications must be modified to account for the change in the provider application. The solution for this problem requires methods to be found that allow the services to be extended, for example by adding more parameters without breaking the previous versions of the consumer application

This diversity and the extendibility have been partly dealt with by developing standards and partly by further evolution of technology. This article mostly deals with standards. (Part 4 will look at evolution and development in technology.) The standards are a collection of specifications, rules, and guidelines formulated and accepted by the leading market participants and are independent of implementation details. Standards establish a base for commonality and enable wide acceptance through interoperability. Examples of standards include:

- A common communication language (XML).
- A common format for exchanging messages (SOAP).
- A common service specification format (Web Services Description Language, or WSDL).
- Common means for service lookup (Universal Description, Discovery, and Integration, or UDDI).

Examples of technology development include:

- Further development of enterprise service bus (ESB) ideas so as to be able to handle different protocols for the service provider and service consumer.
- Further development of registry ideas for easy registration and discovery of services.

XML

XML has been adopted as a popular middleware-independent standard format for the exchange of data and documents. XML is basically the smallest common denominator upon which the IT industry can agree. Unlike CORBA IDL or Java interfaces, XML isn't bound to any particular technology or middleware standard and is often used today as an ad hoc format for processing data across different, largely incompatible middleware platforms. XML is free and comes with a large number of tools on many different platforms, including different open source parsing APIs, such as Simple API for XML (SAX) and Document Object Model (DOM). These tools enable processing and management of XML documents. Another advantage of XML is that it retains the data's structure in transit. XML is also very flexible, which positions it as the most suitable standard for solving middleware and application heterogeneity problems.

SOAP

Although adopting XML is an important step forward to deal with the heterogeneity and extensibility requirements, XML alone isn't sufficient for the two parties (the service provider and service consumer applications) to properly communicate. For effective communications, the parties must be able to exchange messages according to an agreed-upon format. SOAP is such a protocol; it provides a common message format for services.

SOAP is a text-based messaging format that uses an XML-based data encoding format. SOAP is independent of both a programming language and operational

platform. It doesn't require any specific technology at the endpoints, making it completely agnostic to vendors, platforms, and technologies. Its text format also makes SOAP a firewall-friendly protocol. Although SOAP was originally designed to work only with HTTP, any transport protocol or messaging middleware can be used to carry a SOAP message.

A SOAP message is a complete XML document, with the top element being the envelope element. The envelope element contains a body element and an optional header element. The body element usually carries the actual message that's consumed by the recipient. The header element is usually used for intermediate processors for advanced features. A simple but complete example of a SOAP request for obtaining a stock quote is shown in Listing 1. The listing shows how a SOAP message is encoded using XML and illustrates some SOAP elements and attributes.

Listing 1 shows that the top element in SOAP must be the `envelope` element, which must contain two namespaces: The namespace `SOAP:encodingStyle` indicates the SOAP encoding, and the other namespace connotes the SOAP envelope. The header element is optional, but when it's present it should be the first immediate child of the `envelope` element. The `body` element must be present in all SOAP messages and must follow the `header` element if it's present. The body usually contains the specification of the actual message. In Listing 1, the message contains the name (`GetLastTradePrice`) of the method and an input parameter value (`IBM`).

Listing 1. SOAP message example

```
<soap:envelope
xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
  <soap:header>
  </soap:header>
  <soap:body>
    <m:GetLastTradePrice
xmlns:m='http://example.org/Tradeprice'  >
      <tickerSymbol> IBM </tickerSymbol>
    </m:GetLastTradePrice>
  </soap:body>
</soap:envelope>
```

Web Services Description Language

The second application of XML to solve the heterogeneity problems mentioned above in the services-based integration pattern is in the declaration of the service interface through the use of WSDL. This is mostly selected for the benefits of XML outlined in a previous paragraph. In addition, new features were added to address the problem of heterogeneity of the enterprise. These include a provision to specify the transport protocol, which is needed to access the service, and a clearer method

of declaring both synchronous and asynchronous services. Last, but equally important, a new method allows extending the services without making the previous versions of the client software obsolete.

It's instructive to look at an example of a WSDL document. Part of a WSDL document is shown in Listing 2, which declares a service for getting weather information.

Listing 2. An example of WSDL

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions name = 'WeatherWebService'
    targetNamespace= 'urn:WeatherWebService'
    xmlns:tns= 'urn:WeatherWebService'
    xmlns= 'http://schemas.xmlsoap.org/wsdl/'
    xmlns:xsd= 'http://www.w3.org/2001/XMLSchema'
    xmlns:soap= 'http://schemas.xmlsoap.org/wsdl/soap/'
  <types/>
  <message name= 'WeatherService_getWeather' >
    <part name= 'City' type= 'xsd:string' />
  </message>
  <message name= 'WeatherService_getWeatherResponse' >
    <part name= 'result' type= 'xsd:string' />
  </message>
  <portType name= 'WeatherService' >
    <operation name= 'getWeather' parameterOrder= 'City' >
      <input message= 'tns:WeatherService_getWeather' />
      <output message= 'WeatherService_getWeatherResponse' />
    </operation>
  </portType>
  <binding name= 'WeatherServiceBinding'
    type= 'tns:WeatherService' >
    <operation name= 'getWeather' >
      <input>
        <soap:body use= 'literal'
          namespace= 'urn:WeatherWebService' />
      </input>
      <output>
        <soap:body use= 'literal'
          namespace= 'urn:WeatherWebService' />
      </output>
      <soap:operation soapAction= '' />
    </operation>
  </binding>
  <transport= 'http://schemas.xmlsoap.org/soap/http'
    style= 'rpc' />
  </binding>
  <service name= 'WeatherWebService' >
    <port name= 'WeatherServicePort'
      binding= 'tns:WeatherServiceBinding' >
      <soap:address
        location= 'http://mycompany.com/weatherservice' />
      </port>
    </service>
```

As Listing 2 shows, a complete WSDL consists of a set of definitions starting with a root definitions element followed by six individual element definitions— types, message, portType, binding, port, and service—which describe a service. Let's break these elements down into more detail:

- **types:** Defines the data types contained in messages exchanged as part of the service. Data types can be simple, complex, derived, or array types. Types, either schema definitions or references, that are referred to in a WSDL document's message element are defined in the WSDL document's type element.
- **message:** Defines the messages that the service exchanges. A WSDL document has a message element for each message that's exchanged, and the message element contains the data types associated with the message. For example, in [Listing 1](#) the first message contains a single part, which is of the type string.
- **portType:** Specifies, in an abstract manner, the operations and messages that are part of the service. A WSDL document has one or more portType definitions for each service it defines. In [Listing 1](#), only one service type, `WeatherService`, is defined.
- **binding:** Binds the abstract port type, along with its messages and operations, to a transport protocol and message format. In [Listing 1](#), one operation, `getWeather`, is defined, which has both an input and output message. Both of these messages are exchanged in SOAP body formats. The binding transport protocol is HTTP.
- **service and port:** Define, together, the name of an actual service and, by providing a single address for binding, assign an individual endpoint for the service. A port can have only one address. The service element groups related ports together and, through its name attribute, provides a logical name for the service. In [Listing 1](#), one service with the name `WeatherWebService` is defined, which has a single port (or endpoint) with the address `http://mycompany.com/weatherservice`.

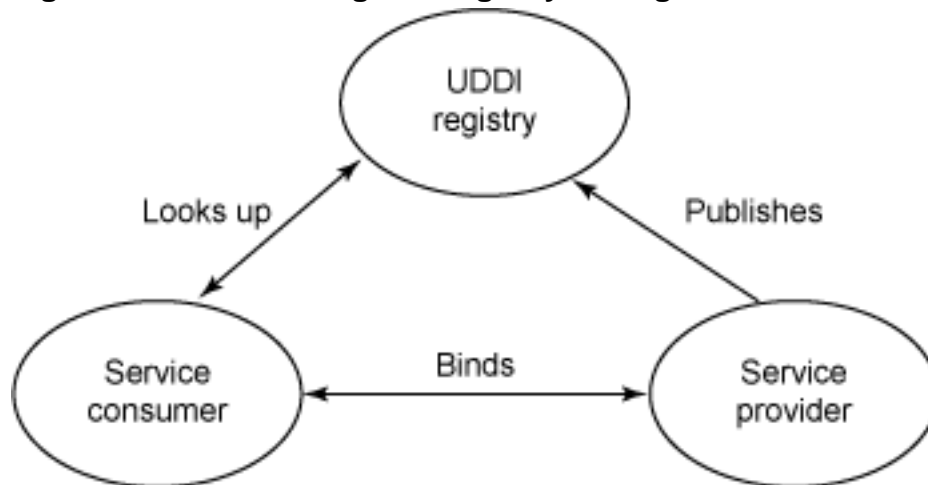
Registry and UDDI

In addition to a service interface declaration (which, in this case, is WSDL) and SOAP messaging standard, a large enterprise also needs a central place where the service provider can publish its services using WSDL and where the service consumers can discover existing services. This is mainly due to the fact that in a large enterprise, developer resources may be dispersed geographically. Such a central place is called a *registry*. A registry is like a library card catalog, used for recording the arrival of new books and other media, and looking up existing items. Another common analogy is the telephone system's yellow pages, which is used to publish services by the service providers and to find services by the service consumers. When a consumer finds a service, the registry has no role to play between the service provider and service consumer.

The UDDI specification defines a standard way of registering, deregistering, and

looking up services. Figure 1 shows how UDDI enables dynamic description, discovery, and integration of services. A service provider first registers a service with a UDDI registry. A service consumer then looks up the required service in the UDDI registry, and, when it finds the required service, the consumer directly binds with the provider to use the service.

Figure 1. Basic working of a registry through the use of UDDI



There are three categories of binding to a specific service after the service has been discovered through the use of the registry:

- **Development time binding:** In this case, in addition to the signatures of the service operations and the service (network) protocol, the actual physical location of the service is known at development time. The client logic is developed accordingly. Thus, the binding is hard-coded to use a specific service and is permanent.
- **Partly runtime binding:** As in the previous case, the signatures of the service operations are known at the development time as well as the network protocol. However, the address of the specific service isn't known during code development. In this case the consumer application is enabled to dynamically bind to a different service instance by looking up services with a specific name or property in the repository. For example, a consumer application looks up printing services with different names depending on the printer name selected by the user. Another example is the case when a printer service is selected based on the properties, such as the floor number and document type.
- **Runtime binding:** In this case, even the service specification (the operations signatures) and the protocol aren't known at development time. The client can still discover a service by using the properties, such as a floor number and document type, but with an unknown service interface. Here, some kind of reflection mechanism must be implemented

at the client side, which enables the client to dynamically discover the semantics of the service and format of valid requests. This type of service discovery is the most complex and unused, often because it requires complex client logic to dynamically interpret the semantics of an unknown service interface.

Conclusion

In this installment of the series you've learned about various standards that mostly constitute Web services. These standards include XML, WSDL, SOAP, and UDDI. The use of these standards solves many of the heterogeneity problems in a large enterprise. However, there are still other heterogeneity problems left unresolved. For example, there may be a mismatch between the transport protocols employed by a service consumer and a service provider. The solution of these types requires further technology development, which will be addressed in Part 4 of the series.

Resources

Learn

- Learn more about SOA architecture in the book [Enterprise SOA: Service-oriented Architecture Best Practices](#).
- Find out more about [WSDL](#).
- Take an [XML tutorial](#).
- Learn more about [SOAP](#).
- Read [Enterprise Integration Patterns](#), an excellent resource for integration patterns.
- The [SOA and Web services zone](#) on IBM developerWorks hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials on how to develop Web services applications.
- Play in the [IBM SOA Sandbox!](#) Increase your SOA skills through practical, hands-on experience with the IBM SOA entry points.
- The [IBM SOA Web site](#) offers an overview of SOA and how IBM can help you get there.
- Stay current with [developerWorks technical events and webcasts](#).
- Browse for books on these and other technical topics at the [Safari bookstore](#).
- Check out a quick [Web services on demand demo](#).
- Get an [RSS feed for this series](#). (Find out more about [RSS](#).)

Get products and technologies

- Innovate your next development project with [IBM trial software](#), available for download or on DVD.

Discuss

- [Participate in the discussion forum for this content](#).
- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

About the author

Dr. Waseem A. Roshen, PhD

Dr. Waseem Roshen is an IT architect in the Enterprise Architecture and Technology

Center of Excellence of IBM Global Business Services in Columbus, Ohio. He works on enterprise architecture and integration. He's also a Sun Certified J2EE Architect, has published 60 articles, and has worked on 24 patents.

Trademarks

IBM, the IBM logo, and WebSphere are registered trademarks of IBM in the United States, other countries or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.