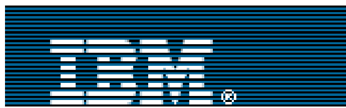


Web service invocation sans SOAP, Part 2 : The architecture of Web Service Invocation Framework

WSIF's architecture

[Nirmal K. Mukhi](#) ([nmukhi@us.ibm.com](#)), Research associate, IBM Research
[Aleksandor Slominski](#) ([aslom@indiana.edu](#)), Research assistant, IU Extreme! Lab, Indiana University

September 2001

WSIF was introduced in a previous article which explained how it provides a binding-independent API for simplified Web service invocation. This article will look at some advanced WSIF features. This will require an overview of its architecture, following which you will see how to exploit multiple service bindings, and how to update or add new binding implementations for Web services.

Contents:

[WSIF's architecture](#)
[Modifying the binding selection algorithm](#)
[Updating/adding a binding implementation at runtime](#)
[Stub architecture](#)
[Conclusion](#)
[Resources](#)
[About the authors](#)
[Rate this article](#)

Related content:

[Web Services Description Language](#)
[Web service invocation sans SOAP, Part 1](#)

Also in the Web services zone:

[Tutorials](#)
[Tools and products](#)
[Articles](#)

The Web Services Invocation Framework (WSIF) is a toolkit that provides a simple API for invoking Web services, no matter how or where the services are provided. In a previous article (see [Resources](#)), we discussed the need for WSIF, the philosophy behind its design, and described some of the main features. We compared the WSIF's WSDL-driven API to conventional APIs for using Web services and described the port type compiler as well as stubless invocation.

WSIF has even more to offer. Its architecture allows invocation ports to be discovered via port factories. We can implement invocation ports that make service invocations using customised bindings and plug them into the framework. We can also design our own port factories so that invocation ports are looked up or created using a customised algorithm. Finally, WSIF allows us to use *any* native type system for data used within messages. In this article, we will describe the architectural aspects of WSIF that enable these features, and discuss specific ways of exploiting this flexible architecture.

WSIF's architecture

WSIF invokes service operations through the following steps:

1. It loads a WSDL document.
2. It creates a port factory for this service.
3. The port factory is used to retrieve a service port.
4. It creates messages, if necessary, by using message parts typed according to some native type system.
5. It makes the invocation by supplying the port with the name of the operation to be invoked, along with an input and/or output message as is required by the operation.

The WSIFPort

The key abstraction here is the run-time representation of a WSDL port, called the `WSIFPort`. This is responsible for doing the actual invocation, based on a particular binding. So we have, for example, a `WSIFSOAPPort` that is capable of using the SOAP binding specified in the WSDL for this service in order to invoke abstract service operations. Flexibility in the architecture is centered around the `WSIFPort` interface shown in [Listing 1](#).

Listing 1: The `WSIFPort` interface

```
public interface WSIFPort {
  public boolean executeRequestResponseOperation (String op,
                                                   WSIFMessage input,
                                                   WSIFMessage output,
                                                   WSIFMessage fault)

        throws WSIFException;
  // some other methods not specified here
}
```

An implementation of this interface would have to know how to invoke the operation using the specified abstract input and output message. A SOAP implementation for the `WSIFPort`, if based on the Apache SOAP API, might create a `Call` object based on the SOAP binding infomation from the WSDL document for this service, and then create the necessary parameters for invocation using the abstract input message. The SOAP response could be used to populate the abstract output message which can then be examined by the client.

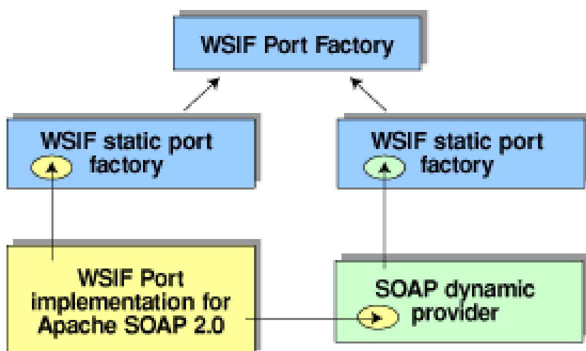
The WSIFPortFactory
The `WSIFPortFactory` is responsible for retrieving a `WSIFPort` to be used for a particular invocation. It has the interface described in Listing 2.

Listing 2: The `WSIFPortFactory` interface

```
public interface WSIFPortFactory {
  public WSIFPort getPort () throws WSIFException;
  public WSIFPort getPort (String portName) throws WSIFException;
}
```

Notice that the version of `getPort` with no parameters allows an implementation of this interface to have some customised algorithm for choosing a WSIFPort for an invocation. The WSIF distribution has two implementations of this interface: a *static port factory* and a *dynamic port factory*. The static port factory stores a list of `WSIFPort` objects and returns one of them in a pseudo random manner when `getPort()` is issued. The dynamic port factory stores a list of dynamic providers. Each of these providers is capable of creating a `WSIFPort` at runtime, based on the WSDL information for the service. When `getPort()` is invoked on the dynamic port factory, the factory picks a dynamic provider in a pseudo random manner that allows it to create a `WSIFPort` which is then returned. An overall view of the port factory architecture is illustrated in Figure 1.

Figure 1: WSIF port / port factory architecture



The WSIF part
So far we have discussed how WSIF ports are decoupled from specific implementations and how port factories are used to discover or create ports. Ports allow us to make invocations, but an essential step before carrying out the invocation itself is creation of the messages required by the operation. Messages in WSDL are composed of named parts tied to some type system. Typically, WSDL parts are typed using XML schema as the type system. This is language-independent and quite powerful. Invocation of such services from a client is done by a mapping this schema type to a more convenient type system and allowing translation between objects belonging to that type system and schema types. For example, when Java is the native type system, translation between Java and schema is achieved

through serialization and deserialization using classes designed for that explicit purpose.

WSIF's part architecture is designed to allow any native type system to be used for a message part, and allows parts within the same message to be typed using different type systems. The latter is required in cases where parts from two or more separate WSDL messages, associated with different native type systems, have to be combined into a single message. Besides allowing different native type systems, there needs to be a common way of interpreting message parts so that it is possible to view WSIF message objects uniformly. The approach taken by WSIF is that schema is the standard abstract type used for message parts, and all such schema types have a corresponding JavaBean that can be defined using a canonical mapping to convert schema to Java. This gives us a common type system for WSIF parts. So the `WSIFPart` interface looks like Listing 3.

Listing 3: The `WSIFPart` interface

```
public interface WSIFPart {
  public Class getJavaType ();
  public Object getJavaValue ();
}
```

Specific implementations of this interface can use any internal type system as long as these representations can be converted to the corresponding canonical Java types. The only implementation of the `WSIFPart` interface provided by WSIF is the `WSIFJavaPart` which uses Java as the type system. Alternative implementations can exploit common usage patterns to improve efficiency. For example, consider the case where the most common binding used involves exchange of XML documents as in the document style SOAP binding. Here, it would be very efficient to represent part values as literal XML instead of Java objects, that way there would be no parsing or serialization/deserialization required when the document style SOAP binding is used for invocation.

Modifying the binding selection algorithm
Modification of the binding selection algorithm is quite straightforward. All the developers have to do is write their own `WSIFPortFactory` implementation, or extend existing ones. Consider the `WSIFDynamicPortFactory`. This stores a list of dynamic providers that generate a `WSIFPort` for a particular WSDL binding on demand. This port factory looks up dynamic providers by the provider type, so, for example, it knows about one dynamic provider that handles SOAP ports, one that handles CORBA ports we may define, etc. We can extend this port factory with our own `getPort()` method to choose a port that is desirable. To illustrate a situation when this is helpful, consider the Web service for an address book in Listing 4.

Listing 4: The address book Web service

```
<?xml version="1.0" ?>
<definitions targetNamespace="http://www.ibm.com/namespace/wsif/samples/ab"
             xmlns:tns="http://www.ibm.com/namespace/wsif/samples/ab"
             xmlns:typens="http://www.ibm.com/namespace/wsif/samples/ab/types"
             xmlns:xsd="http://www.w3.org/1999/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:java="http://schemas.xmlsoap.org/wsdl/java/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <!-- type defs come here, but we skip them for brevity -->
  <!-- message declarations come here, but we will skip them for brevity -->
  <!-- port type declarations -->
  <portType name="AddressBook">
    <operation name="addEntry">
      <input message="tns:AddEntryRequest"/>
      <output message="tns:AddEntryResponse"/>
    </operation>
    <operation name="getAddressFromName">
      <input message="tns:GetAddressFromNameRequest"/>
      <output message="tns:GetAddressFromNameResponse"/>
    </operation>
```

```
  </portType>
  <!-- binding declarations -->
  <binding name="SOAPBinding" type="tns:AddressBook">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="addEntry">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
                   namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
    <operation name="getAddressFromName">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
                   namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>
  <binding name="JavaBinding" type="tns:AddressBook">
    <java:binding/>
  </binding>
  <!-- service declaration -->
  <service name="AddressBookService">
    <port name="SOAPPort" binding="tns:SOAPBinding">
      <soap:address location="http://localhost:8080/soap/servlet/rpcrouter"/>
    </port>
    <port name="JavaPort" binding="tns:JavaBinding">
      <java:address class="services.addressbook.AddressBook"/>
    </port>
  </service>
</definitions>
```

This service offers two bindings: SOAP over HTTP, and native Java. There is no standard way of binding a Java class in WSDL directly, but that's just a minor speed bump in the way of our race car of comprehension, so we won't dwell on it. Given the availability of multiple bindings for this service, a client would want to use the one that perhaps gave best performance such as the Java binding, since it presumably involves direct translation of an abstract operation to a Java method invocation without network, serialization/deserialization, etc. operations. Of course, the Java binding is available only in the environment in which the service was deployed. It isn't available for public use on the Internet as is likely for the SOAP binding. So what we want to do is choose the Java binding when we can, and use the SOAP binding in other cases. This could be done by writing our own `WSIFPortFactory` as shown in Listing 5.

Listing 5: A customized binding selection algorithm (see Resources for full code)

```
public class MyWSIFPortFactory extends
        WSIFDynamicPortFactory {
  public WSIFPort getPort () throws WSIFException {
    WSIFException ex = null;
    WSIFPort portInstance = null;
    // examine list of available ports
    // do we have a port with a Java binding?
    for(int i = 0; i < myPortsArr.length; ++i) {
      try {
        Port port = myPortsArr[i];
        Binding binding = port.getBinding();
        if (binding instanceof JavaBinding) {
          // attempt to create a port that can
          // make invocations using this binding
          portInstance =
             createDynamicWSIFPort(def, service, port);
          if(portInstance != null) {
            return portInstance;
          }
        }
      } catch(WSIFException wex) {
        if(ex == null) {
          ex = wex;
        }
      }
    }
    // unable to create a port that can invoke a Java
    // binding (why? perhaps there is no Java port
    // for this service, or the Java port cannot be
    // created for invocation since we are not in
    // an environment that has access to the resources
    // that the Java binding uses)
    // use WSDL ports with other
    // bindings for invoking the service
    // Full code not shown here due to length.
    // Please see the Resources section at the end of
    // this article to download the actual code.
  }
}
```

Updating/adding a binding implementation at runtime
Suppose we want to upgrade our SOAP implementation. Using this architecture we could complete the migration to the new implementation *without having to recompile user code or stub code*, since the WSIF API will remain the same. We would write a new `WSIFPort` implementation, capable of making invocations using the modified SOAP client API that is provided by our new SOAP implementation. Then, we would write a dynamic provider which translates a WSDL SOAP port to our new `WSIFPort` implementation. Once we register this dynamic provider it would replace the previously registered SOAP dynamic provider, and invocation for all WSDL SOAP ports would take place through our new `WSIFPort` implementation.

Consider a situation where we define our own WSDL binding. First of all, we have to make sure that the WSIF runtime is capable of loading the WSDL document with the new binding; this requires us to define handlers for the extensibility elements we add to our WSDL using WSDL4J's (see Resources) extension registry API. We won't go into details of how this is done. We can then write `WSIFPort` and dynamic provider implementations, and register this new dynamic provider with the port factory in the same manner as we did for our updated SOAP binding.

Stub architecture

One of the most important benefits we gain from this architecture is that it enables us to have client stubs that are customisable by managed environment. All WSIF stubs extend the base class specified in Listing 6.

Listing 6: WSIFStub base class

```
// some methods and other details omitted
public abstract class WSIFStub {
  /**
   * Locates a port factory using JNDI if available,
   * otherwise uses dynamic port factory with
   * pre-registered dynamic providers
   */
  protected void locatePortFactory(.....) {
        .....
  }
  /**
   * Initializes stub with ports defined in WSDL document
   * document must contain *exactly* one service and all
   * ports must implement one and only one port type.
   */
  protected void initializeFromLocation (...) {
        .....
  }
  /**
   * Return the port currently being used.
   */
  public WSIFPort getPort () {
    return wp;
  }
  /**
   * Return the port factory currently being used.
   */
  public WSIFPortFactory getPortFactory () {
    return wpf;
  }
  /**
   * Create a new proxy using the given WSIFPortFactory.
   */
  public void setPortFactory (WSIFPortFactory wpf) {
        .....
  }
  /**
   * Create a new proxy pre-configured to use the given port
   * for interacting with the service.
   */
  public void setPort (WSIFPort wp) {
        .....
  }
  /**
   * Select the port to use by giving the name of the port
   * that is desired. If a port of that name cannot be retrived from
   * the port factory than an exception will be thrown. The port
   * I use will only be updated if this method is successful.
   */
  public void selectPort (String portName) {
        .....
  }
```

```
}
```

This stub architecture gives us tremendous benefits. It allows applications to use the same stub while allowing the managed environment to make changes in the plumbing without having to recompile anything. Consider the case in Listing 5, where we wrote our own port factory with a customised binding selection algorithm. If we operated in an application server environment with JNDI, we could bind this port factory implementation to the appropriately named JNDI context. When the stub would lookup the port factory during initialization, it would get the new implementation. Of course, we can force the change in the port factory used by the stub by directly calling the `setPortFactory` method on the stub if that's how we wanted to do it, or we could force use of a particular port using the stub's `setPort` method.

Conclusion

As Web services continue to evolve, it will become common to use protocols other than SOAP for access to service endpoints. When we write applications that use Web services, we therefore need to use APIs that operate at the WSDL level rather than at the SOAP level.

WSIF provides an abstract way of looking at Web service invocation so we can write applications that use invocation APIs free of binding dependencies. WSIF has a port type compiler for generation of a customisable stub; the WSIF API is simple enough so that applications can make stubless invocations by using the it directly. WSIF's architecture allows a flexible way of defining custom ports and port factories and also allows messages to be created using any native type system. Generated stubs have entry points for managed environments such as application servers to make runtime modifications in the port or port factory used by the stub to make invocations.

Web services need an extensible invocation framework that is free of binding dependencies, and WSIF is an initial step in that direction.

Resources

- Participate in the discussion forum on this article by clicking **Discuss** at the top or bottom of the article.
- Read the introductory article which discusses how WSIF is an improvement over current Web service invocation models and describes some of WSIF's main features.
- You can download the full code in Listing 5 of this article and for the examples in Figure 2 and 3 from this location.
- Download the WSIF distribution on alphaWorks and try out the easier samples. This will let you see first hand the different invocation styles supported by WSIF and its advantages over protocol-specific client APIs.
- Go over the WSDL specification to see what kinds of extensions are allowed; you can also study how WSDL's extension mechanism is used to define a SOAP binding for accessing Web services.
- Go over the SOAP specification itself.
- If you haven't programmed with Web services before, the Web Services ToolKit is a good starting point.
- Take a look at WSDL4J, an extensible WSDL parsing framework over which WSIF has been built.

About the authors

Nirmal K. Mukhi is a Research Associate at IBM's T J Watson Research Lab where he has been working on various Web services technologies since November 2000. His other interests include AI, creative writing, and outdated computer games. You can reach Nirmal at nmukhi@us.ibm.com.

Aleksander Slominski is a doctoral student at Indiana University where he is working at the IU Extreme! Lab as a research assistant on implementing XML/SOAP enabled version of the Common Component Architecture. He has also designed and implemented the XML Pull Parser and is interested in performance and usability aspects of XML. During the summer of 2001 he was an intern at IBM's T J Watson Research Lab where he worked on first version of WSIF. You can reach Alek at aslom@indiana.edu.

What do you think of this article?

Killer! (5) Good stuff (4) So-so; not bad (3) Needs work (2) Lame! (1)

Send us your comments or click Discuss to share your comments with others.

About IBM | Privacy | Legal | Contact