How to write a provider

1 What is a WSIF Provider?

WSIF, as its name suggests is an open-ended framework for invoking WSDL described services. As long as a WSDL description of the endpoint exists, it should be possible to use this framework to invoke that piece of software, whatever it may be.

As has been described in this introductory article, the WSIF API is driven by the abstract WSDL description of the service, and thus operates at a protocol-independent level. When an invocation takes place at runtime, the invocation of the abstract operation offered by the service needs to be converted into a protocol-specific mechanism for contacting the service endpoint, conveying the input message or other required information to the service, and receiving the response if any that results from the invocation. This result is eventually returned to the application that uses the high-level, protocol-independent WSIF API.

The protocol-specific piece of software that enables the invocation to take place is the WSIF provider. There is one WSIF provider for each kind of WSDL binding supported by a particular installation of WSIF. For example, a particular installation may include the WSIF core along with the SOAP and EJB providers only. This would allow clients using WSIF at that site to invoke WSDL-described service that have SOAP or EJB bindings. Providers are pluggable and can be added to the installation at runtime.

So far we understand that when the code using the WSIF API (let's call it the application, even though this code might actually be WSIF's dynamic proxy which provides an additional layer of abstraction for the real application) to invoke a service with a SOAP binding, the invocation takes place through a WSIF provider that supports this binding. How is a provider discovered? What happens if multiple providers support the same binding? (This is indeed the case for SOAP, since WSIF includes a SOAP provider based on Axis and one based on Apache SOAP). We will not address these issues in this document, here we will concentrate on the provider architecture itself, without regard to how WSIF knows about their existence.

2 Writing your own WSIF provider

A pre-condition for a working provider is that there should be a well-defined WSDL binding that it supports, along with the associated WSDL4J code that is capable of parsing the binding. Note that WSDL4J, by default, supports only the standard WSDL 1.1 bindings:

SOAP, HTTP and MIME. To add support for other providers in WSIF, one would first need to define the WSDL binding extensions, define the corresponding WSDL4J extensibility elements, and then register the serializers and deserializers for these extensibility elements with the WSDL4J extension registry. Details on this are available here. The WSIF provider will use the WSDL4J extensibility elements defined since these are the in memory representation of the data in the binding.

3 The WSIFProvider interface

Let's discuss the specifics of a WSIF provider. A provider must implement the following interface:

```
    /**
     * For the given WSDL definition, service and port
     * try to provide dynamic port,
     * or return null if this provider can not do it.
     * It is required to pass definition and service in addition to port
     * as in current WSDL4J it is not possible to retrieve service to
     * which port belongs and definition in which it was defined.
     */
    public WSIFPort createDynamicWSIFPort(
        Definition def,
        Service service,
        Port port,
        WSIFDynamicTypeMap typeMap)
        throws WSIFException;

    /**
     * Returns the WSDL namespace URIs of any bindings this provider
supports.
     * The assumption is made that the provider supports all combinations
of
     * binding and address namespaces returned by this and the
     * getAddressNamespaceURIs method.
     * @return an array of all binding namespaces supported by this
provider
     */
    public String[] getBindingNamespaceURIs();

    /**
     * Returns the WSDL namespace URIs of any port addresses this provider
supports.
     * The assumption is made that the provider supports all combinations
of
     * binding and address namespaces returned by this and the
     * getBindingNamespaceURIs method.
     * @return an array of all address namespaces supported by this
provider
     */
    public String[] getAddressNamespaceURIs();
```

OK, that's all you need to know. What, you mean the above wasn't self-explanatory?

So let's get into a little more detail. Let's start with the simpler methods: `getBindingNamespaceURIs()` and `getAddressNamespaceURIs()`. Each binding extension in WSDL is defined in a particular XML namespace. For example, the SOAP binding is defined in the namespace `http://schemas.xmlsoap.org/wsdl/soap/`, and similarly, so are the extensions under the `<port>` section of a WSDL document (often, as in the case with the SOAP extensions, the namespace is the same as the one in which binding extensions are defined). So, as the javadoc comments suggest, the WSIF runtime assumes that the provider supports invocation of any service with a namespace URI for binding extensions that is one of those returned by `getBindingNamespaceURIs()` and a namespace URI for address extensibility elements that is one of those returned by `getAddressNamespaceURIs()`.

The core of the provider is the method `createDynamicWSIFPort(..)`. This returns a `WSIFPort` object that is constructed dynamically using:

- the definition of the service we are invoking (the `javax.wsdl.Definition` parameter)
- the service we are invoking within that particular WSDL (the `javax.wsdl.Service` parameter)
- the port offered by the service that lets us know the specific service endpoint, the binding and the port type that is being invoked (the `javax.wsdl.Port` parameter)
- a type map that maps abstract schema types in the WSDL messages for the operations in the port type being invoked to java types that correspond to them. WSIF will expect messages provided at the time of invocation to have parts that are java objects of the required type, as specified by the type map (it of course allows the object used for the invocation to be of a less specific type as well).

Let's consider how a specific provider implements this and other interfaces. We will concentrate on identifying patterns that most provider implementations follow. Consider, for example, the Apache SOAP provider. Here is the source code for this implementation of the `WSIFProvider` interface. It is fairly straightforward. The constructor usually does very little, in most cases nothing. In this particular case it has a mechanism to set up the binding and address namespace URIs supported by this provider. It also adds to the WSDL4J extension registry used by WSIF the capability to parse JMS extensions. The capability to parse SOAP, EJB and java extensions are pre-registered, all other binding extensions may be registered in this fashion. The `createDynamicWSIFPort` method in the Apache Axis provider parses the binding associated with the port being invoked and verifies that it is indeed a SOAP binding. If this is the case, it creates a `WSIFPort_ApacheAxis` object, which implements the `WSIFPort` interface.

4 The WSIFPort interface

The main function of the WSIF port that is to create a `WSIFOperation` object at runtime when a WSDL operation needs to be invoked. At the minimum, the `WSIFPort` needs to know the name of the operation. If there is a possibility of overloaded operations, the application may invoke the form that takes the names of the input and output messages in addition to the name of the operation. `WSIFOperation` objects are the brains of the outfit; they actually perform the invocation based on a specific protocol. But more on that later, first let's see how the Axis provider implements the `WSIFPort` interface. The implementation in the constructor itself parses the binding being invoked, doing some processing based on the transport being used (since WSIF's Axis provider supports SOAP messages over HTTP or JMS). In addition, it also verifies that the binding is indeed supported (for example WSIF does not support SOAP bindings that use document style instead of RPC style invocations *using the Axis provider* (document style is supported when using the Apache SOAP provider to handle SOAP bindings). Finally, the Axis implementation of the port actually iterates through the binding, creating a `WSIFOperation_ApacheAxis` object (the protocol-specific implementation of the `WSIFOperation` interface) for each operation in the binding. These `WSIFOperation` objects are cached so that they don't have to be created each time. Of course that is optional, a bare-bones version of a provider could do very little in the constructor. The most useful method implemented here is `createOperation`, which creates the appropriate `WSIFOperation_ApacheAxis` object (based on the operation name and input and output names, if provided, to resolve the exact operation the application wants to invoke). The method first looks up the cache containing previously created instances and may reuse them or may create a new one.

5 The WSIFOperation interface

The real brains of the outfit is the `WSIFOperation` object. This interface has a number of useful methods, so let's do it in some detail:

```
/**
 * A WSIFOperation is a handle on a particular operation of a portType
 * that can be used to invoke web service methods. This interface is
 * implemented by each provider. A WSIFOperation can be created using
 * {@link WSIFPort#createOperation(String)}.
 *
 * @author Owen Burroughs <owenb@apache.org>
 * @author Ant Elder <antelder@apache.org>
 * @author Jeremy Hughes <hughesj@apache.org>
 * @author Mark Whitlock <whitlock@apache.org>
 */
public interface WSIFOperation extends Serializable {
```

```
    /**
     * Execute a request-response operation. The signature allows for
     * input, output and fault messages. WSDL in fact allows one to
     * describe the set of possible faults an operation may result
     * in, however, only one fault can occur at any one time.
     *
     * @param op name of operation to execute
     * @param input input message to send to the operation
     * @param output an empty message which will be filled in if
     *        the operation invocation succeeds. If it does not
     *        succeed, the contents of this message are undefined.
     *        (This is a return value of this method.)
     * @param fault an empty message which will be filled in if
     *        the operation invocation fails. If it succeeds, the
     *        contents of this message are undefined. (This is a
     *        return value of this method.)
     *
     * @return true or false indicating whether a fault message was
     *         generated or not. The truth value indicates whether
     *         the output or fault message has useful information.
     *
     * @exception WSIFException if something goes wrong.
     */
    public boolean executeRequestResponseOperation(
        WSIFMessage input,
        WSIFMessage output,
        WSIFMessage fault)
        throws WSIFException;

    /**
     * Execute an asynchronous request
     * @param input   input message to send to the operation
     *
     * @return the correlation ID or the request. The correlation ID
     *         is used to associate the request with the WSIFOperation.
     *
     * @exception WSIFException if something goes wrong.
     */
    public WSIFCorrelationId executeRequestResponseAsync(WSIFMessage input)
        throws WSIFException;

    /**
     * Execute an asynchronous request
     * @param input   input message to send to the operation
     * @param handler   the response handler that will be notified
     *        when the asynchronous response becomes available.
     *
     * @return the correlation ID or the request. The correlation ID
     *         is used to associate the request with the WSIFOperation.
     *
     * @exception WSIFException if something goes wrong.
     */
    public WSIFCorrelationId executeRequestResponseAsync(
        WSIFMessage input,
```

```
        WSIFResponseHandler handler)
        throws WSIFException;

    /**
     * fireAsyncResponse is called when a response has been received
     * for a previous executeRequestResponseAsync call.
     * @param response   an Object representing the response
     * @exception WSIFException if something goes wrong
     */
    public void fireAsyncResponse(Object response) throws WSIFException;

    /**
     * Processes the response to an asynchronous request.
     * This is called for when the asynchronous operation was
     * initiated without a WSIFResponseHandler, that is, by calling
     * the executeRequestResponseAsync(WSIFMessage input) method.
     *
     * @param response   an Object representing the response.
     * @param output an empty message which will be filled in if
     *        the operation invocation succeeds. If it does not
     *        succeed, the contents of this message are undefined.
     *        (This is a return value of this method.)
     * @param fault an empty message which will be filled in if
     *        the operation invocation fails. If it succeeds, the
     *        contents of this message are undefined. (This is a
     *        return value of this method.)
     *
     * @return true or false indicating whether a fault message was
     *         generated or not. The truth value indicates whether
     *         the output or fault message has useful information.
     *
     * @exception WSIFException if something goes wrong
     *
     */
    public boolean processAsyncResponse(
        Object response,
        WSIFMessage output,
        WSIFMessage fault)
        throws WSIFException;

    /**
     * Execute an input-only operation.
     *
     * @param input input message to send to the operation
     * @exception WSIFException if something goes wrong.
     */
    public void executeInputOnlyOperation(WSIFMessage input) throws
WSIFException;

    /**
     * Allows the application programmer or stub to pass context
     * information to the binding. The Port implementation may use
     * this context - for example to update a SOAP header. There is
     * no definition of how a Port may utilize the context.
```

```
     * @param context context information
     */
    public void setContext(WSIFMessage context);

    /**
     * Gets the context information for this binding.
     * @return context
     */
    public WSIFMessage getContext();

    /**
     * Create an input message that will be sent via this port.
     * It is responsibility of caller to set message name.
     * @return a new message
     */
    public WSIFMessage createInputMessage();

    /**
     * Create an input message that will be sent via this port.
     * @param name for the new message
     * @return a new message
     */
    public WSIFMessage createInputMessage(String name);

    /**
     * Create an output message that will be received into via this port.
     * It is responsibility of caller to set message name.
     * @return a new message
     */
    public WSIFMessage createOutputMessage();

    /**
     * Create an output message that will be received into via this port.
     *
     * @param name for the new message
     * @return a new message
     */
    public WSIFMessage createOutputMessage(String name);

    /**
     * Create a fault message that may be received into via this port.
     * It is responsibility of caller to set message name.
     * @return a new message
     */
    public WSIFMessage createFaultMessage();

    /**
     * Create a fault message that may be received into via this port.
     *
     * @param name for the new message
     * @return a new message
     */
    public WSIFMessage createFaultMessage(String name);
```

```
}
```

Most of the above is self-explanatory. The important things to note is that the invocation is achieved through this object. It also is capable of creating the messages (input, output, fault) that are associated with any invocation; such messages can be populated with data and then provided to methods such as `executeRequestResponseOperation`. Note also that as operations are designed right now, instances may not be reused since they often carry state that interferes with subsequent invocations using the same object. To prevent such misuse, the default implementation of the interface (which is extended by other implementations including `WSIFOperation_ApacheAxis`) has a `close` method which must be invoked at the end of an invocation and sets a flag preventing further use. Applications may create a new operation instance using the `WSIFPort`. At some point we expect to modify the way operation instances are handled to allow reuse except in specific cases, hopefully simplifying the provider contract and improving performance.

The Axis provider implementation of this interface is here. Everything the class does boils down to the use of the `invokeRequestResponseOperation` method. We won't get into detail, but this uses a protocol-specific library (in this case JAX-RPC, which is the client programming model supported by Axis) to invoke the service. Note how the provider code handles the type map that was provided at the time of creating the `WSIFPort` for invoking this service. For Axis, we need to register serializers and deserializers to marshall and unmarshall java objects into SOAP messages. This must be done prior to an invocation for it to work. Other providers may have to take similar steps to make sure they are capable of handling the java-typed data that populate the input and output messages used for invocation.

So far we have not touched on `WSIFMessage` objects which are ubiquitous in the provider code. It's enough to think of a `WSIFMessage` as a map of WSDL message part names to java objects representing the value of that particular part. The type of the java object must match the expected type according to the type mapping supplied to the WSIF provider. WSIF also allows for creation of messages using primitive java types. The WSIF message interface is available here.

That's it!

The protocol-specific implementation of the `WSIFProvider` interface, the `WSIFPort` interface and the `WSIFOperation`interface are all that are generally required to implement a WSIF provider. Supporting classes may also be included - for example the SOAP providers (both, Apache SOAP and Apache Axis providers) have utility classes to handle message exchanges using the JMS transport.

6 Open Issues

Performance and stateful operations (for example in JCA connectors):

from Paul Fremantle email

How about explicitly identifying which providers require stateful operations and which dont. Those that dont could reuse operations from a cache, ignore closes etc. I'm not sure this will help us either. At the moment our HTTP support is not efficient because we don't reuse HTTP connections. If we updated it to be much more closely bound to the real HTTP interactions then maybe close() might be useful?? I havent really thought this through, but I know that close() was designed to allow long running connections.