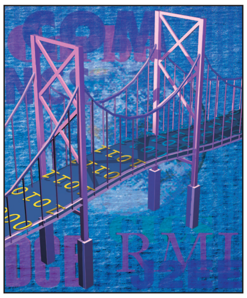
Scripting JAX-WS



Steve Vinoski • *IONA Technologies*

In my last column, I explored possible reasons for the existence of "the language divide," which arises in the traditional standards-based middleware community because few developers use dynamic languages to implement their applications (March/April 2006, pp. 82–84). Instead, they seem to strongly favor Java and C++.

As a long-time fan of both dynamic languages and middleware, I'd like to help shrink or even eliminate the divide, but describing the issues and highlighting the situation isn't the same as directly contributing a usable solution. This time, I describe an integration of the ECMAScript programming language[1] — more commonly known as JavaScript — with an implementation of the Java API for XML Web Services (JAX-WS) 2.0.[2] This integration lets developers implement JAX-WS services using either plain JavaScript or its XML-oriented counterpart, the ECMAScript for XML (E4X) language.[3]

JavaScript and E4X

Despite its name, JavaScript and the Java language are unrelated, except perhaps by some similarity in syntax. Brendan Eich of Mozilla invented JavaScript in 1995 as a scripting language for use within the Netscape Web browser to allow Web pages to be more dynamic. The result is an interpreted language with a simplicity that belies its power.

JavaScript supports only a few types: numbers, strings, booleans, functions, and objects (it also supports arrays, but they're technically objects as well). It supports object-oriented programming, but in a different way than most developers are used to. JavaScript objects are essentially data structures containing name–value pairs called properties. Because JavaScript supports first-class functions, object methods are simply properties whose values are functions. JavaScript inheritance differs from class-based languages as well; because JavaScript has no notion of classes, it supports inheritance through prototype objects.

JavaScript is best known for its use in Web pages. One currently popular approach for enhancing sites' interactive nature and responsiveness is via Asynchronous JavaScript and XML (AJAX; see http://en.wikipedia.org/wiki/AJAX). With AJAX, Web pages contain JavaScript code that asynchronously invokes requests on a Web server by creating an `XmlHttpRequest` request object, attaching to it a JavaScript callback function to handle the response, and then invoking the request. AJAX enhances responsiveness and interactivity by allowing specific page updates without requiring full page reloads. All the hype currently surrounding AJAX could lead you to believe that it's a new approach, but it's really been in use for about 10 years. Its current popularity is due to the fact that different Web browsers have finally gained reasonable consistency in their support of JavaScript. Despite its popularity for use within Web pages, however, JavaScript isn't limited to browser applications.

E4X is a JavaScript extension geared toward handling XML. Specifically, it treats XML as a first-class type that can be created, accessed, and modified using intuitive syntax — as with any other JavaScript type. Later I'll show examples of both JavaScript and E4X.

JAX-WS 2.0

The JAX-WS 2.0 specification — the new version of the Java API for XML-Based RPC (JAX-RPC) — defines standard APIs and approaches for building Java-based Web services. As its name implies, JAX-RPC was concerned mostly with how to implement RPC-oriented Web services in Java. JAX-WS expands and improves on the specification in several ways, including

- tying together and updating its support for several base Web services specifications, such as SOAP 1.2;
- providing a coherent design for using Java annotations to specify Web services metadata;
- providing support for document-oriented Web services, asynchronous services, and services that use transports other than HTTP;
- addressing implementation issues surrounding handlers, which are interceptors that provide hooks into message flows between senders and receivers; and
- describing practices for dealing with versioning in Web services.

Systems based on RPC-oriented specifications such as JAX-RPC generally attempt to let applications work with data in the form of native programming language types. Such systems hide the mechanisms by which application data are marshaled to and from network form, in addition to hiding the network data formats. Making remote requests look like local method calls might seem like a good idea at first glance, but the approach has several significant and well-documented problems. These include the fact that remote methods can suffer partial failures within the overall distributed system that can't occur with local calls,[4] as well as that mapping between marshaled data types and language-specific types is often imperfect.[5] As I described in my September/October 2005 column,[6] Steve Loughran and Edmund Smith of Hewlett-Packard Laboratories authored a detailed critique of JAX-RPC, which included the fact that the specification "relies on a perfect two-way mapping between XML data and native language objects."[7] They refer to such mappings as *object/XML mappings*, often abbreviated as "O/X mappings" or "X/O mappings."

Attempting to devise an X/O mapping for JavaScript service implementations would create much the same problems as those associated with the Java X/O mappings. Fortunately, although JAX-WS also relies on an X/O mapping, and thus presumably suffers from the same problems that Loughran and Smith attribute to JAX-RPC, it offers other approaches for implementing services as well. One improvement that JAX-WS makes over JAX-RPC is that it provides *dispatch* and *provider* interfaces for dealing directly with messages — typically, SOAP messages or SOAP message payloads — to avoid RPC-oriented marshaling layers and the problems associated with X/O mappings.

Because JavaScript doesn't support Java annotations, it can't fulfill JAX-WS annotation requirements by itself.

The dispatch and provider interfaces fit well with JavaScript because an X/O JavaScript mapping would likely be unnatural to JavaScript programmers. In a Web browser environment, JavaScript programs access HTML pages and XML data through Document Object Model (DOM) objects. Therefore, within a JAX-WS setting, providing the JavaScript programmer with a DOM-based approach for accessing raw JAX-WS messages and message payloads through a provider interface introduces a natural path for developing JavaScript-based service endpoints. Given that E4X treats XML as a first-class type, direct access to XML messages and message payloads is an even better fit in E4X than DOM in plain JavaScript.

Implementation

Implementing the underpinnings for JavaScript Web service implementations in JAX-WS obviously requires supporting infrastructure for both JAX-WS and JavaScript. In developing a JAX-WS implementation, I chose the Celtix open-source enterprise service bus (http://celtix.objectweb.org) — in part because, as of this writing, it's less than a year old, and its design and implementation are both very clean. This helps simplify the job of figuring out how to integrate JavaScript into it. Another reason I chose Celtix is that some of the primary developers sit a couple rows away from me at my office, so answers are readily available if I have any questions about the code.

Given that Celtix is implemented in Java, it's best to use a Java-based JavaScript engine such as the very popular Mozilla Rhino open-source engine (see www.mozilla.org/rhino/). Nearly a decade old, it supports JavaScript and E4X and is readily embeddable into Java applications. With Rhino, it's relatively easy to invoke Java methods from JavaScript and vice versa.

Support for JavaScript service implementations in Celtix is relatively straightforward. First, Celtix provides an application class that accepts the names of JavaScript files, E4X files, or the directories containing them. For each JavaScript file (with a .js suffix) or E4X file (with a .jsx suffix) specified on the command line, the application calls Rhino to compile the file; assuming that this succeeds, the application class then creates JAX-WS Provider instances and publishes them as service endpoints.

One tricky problem with publishing a JavaScript service implementation is determining what form the implementation expects for incoming requests. This problem's two aspects are determining

- whether the implementation expects a full message or just a payload, and

- whether the implementation is plain JavaScript or E4X.

The latter is easy, given that the input file suffixes should be different for the two languages. The former isn't quite as straightforward.

JAX-WS relies on Java annotations for specifying metadata associated with Web service implementations. These annotations generally help tie Java implementations together with the details of the WSDL they implement. For example, JAX-WS requires every Provider implementation to carry a `WebServiceProvider` annotation. This particular annotation specifies the service's name, the name of the service's port, the service's target XML namespace, and a URL for the WSDL description. The `ServiceMode` annotation specifies whether the Provider implementation wants full messages or just message payloads. If no `ServiceMode` annotation is present, the Provider implementation receives a message payload. Still another service annotation is `BindingType`, which specifies the binding the JAX-WS runtime should use when publishing the endpoint. If not present, the binding type defaults to the SOAP 1.1/HTTP binding.

Because JavaScript doesn't support Java annotations, it can't fulfill JAX-WS annotation requirements by itself. To solve this, the Celtix JavaScript support code includes general Java Provider implementations that delegate to JavaScript and E4X functions. These delegators specify defaulted annotations as required, but JavaScript service implementations must still somehow specify the required metadata. We achieve this by declaring global JavaScript variables that supply the necessary metadata. After Celtix invokes Rhino to compile the JavaScript or E4X implementations, it then iterates through the compiled code looking for global variables whose names start with the string "`WebServiceProvider`," which should be JavaScript objects that specify annotation metadata. Upon finding the metadata, Celtix uses it to register the specified JavaScript or E4X service endpoint implementation in its runtime.

For example, such a variable might look like this:

```
var WebServiceProvider1 = {
    'wsdlLocation':
'file:./hello_world.wsdl',
    'serviceName':
        'SOAPService1',
    'portName': 'SoapPort1',
    'targetNamespace':
'http://objectweb.org/hw',
    'ServiceMode': 'MESSAGE',
};
WebServiceProvider1.invoke =
   function(request) { ... };
```

Rhino makes it easy to invoke Java methods from JavaScript and vice versa.

This variable supplies information that exactly matches what would appear in a Java `WebServiceProvider` annotation, with two additions. First, it also includes the equivalent of the Java `ServiceMode` annotation, rather than making that a separate variable. It treats the Java `BindingType` annotation (not used in this example) the same way. Because this example specifies "`MESSAGE`" as the service mode and doesn't override the default binding type, the implementation will receive requests in the form of full SOAP 1.1 messages. The variable also includes a property named `invoke`, which should refer to a JavaScript function — effectively serving as an implementation of the base Java Provider interface's `invoke` method.

When the Celtix runtime receives a request for a JavaScript or E4X service, it passes it up to the delegator Provider implementation that the Celtix JavaScript support code previously published as the service endpoint implementation. The delegator then converts the request as required by the implementation. If the implementation is plain JavaScript, the current delegator implementation wraps the incoming DOM document as a JavaScript object and passes it to the JavaScript `invoke` function. Given that the Rhino JavaScript engine allows direct access to Java code from JavaScript, the latter can invoke methods on the Java DOM object just as normal Java code would. In the E4X case, the delegator converts the incoming DOM object into an E4X XML object and passes it to the `invoke` function. Future modifications to the Celtix JavaScript support code will include more effective message handling that avoids the DOM and E4X conversions that currently take place in the delegator.

For example, consider a request with the following XML payload:

```
<ns:greetMe xmlns:ns=
"http://objectweb.org/hw/ns">
  <ns:requestType>Jake
     </ns:requestType>
</ns:greetMe>
```

Let's assume the service implementation just wants to return a response consisting of a similar document that precedes the name provided in the request with the string "Hi." For this example, the response document would be

```
<ns:greetMeResponse xmlns:ns=
 "http://objectweb.org/hw/ns">
  <ns:responseType>Hi Jake
  </ns:responseType>
</ns:greetMeResponse>
```

A plain JavaScript implementation

can do that using normal Java DOM methods to create and populate the response document. An E4X implementation is even simpler:

```
var ns = new Namespace('ns',
  'http://objectweb.org/hw/ns');

 WebServiceProvider1.invoke =
  function(req) {
   default xml namespace = ns;
   var name =
    (req..requestType)[0];
   var resp =
    <ns:greetMeResponse
     xmlns:ns={ns}>
    <ns:responseType>
     {'Hi ' + name}
    </ns:responseType>
    </ns:greetMeResponse>;
   return resp;
 }
```

Note how E4X lets us make XML statements directly inline. This approach is superior to the DOM approach because it's easier to read, less verbose, and more easily reflects the structure of the XML documents being manipulated. E4X also allows XML values and attributes to be set and accessed using the normal JavaScript property access operator

```
resp.ns::responseType =
‘Hi ‘ + name;
```

which is an equivalent way to set the return string in the `responseType` XML element.

The JavaScript and E4X approaches to implementing JAX-WS services provide several benefits over traditional Java or C++ approaches. First, manipulating XML documents completely avoids X/O impedance-mismatch problems. Next, service implementation modifications require no recompilation; just modify your JavaScript or E4X code and rerun the application. Service implementations need to be flexible so they can change as quickly as business requirements change, and I vastly prefer this approach to developing service implementations over the traditional approach of slogging through heavyweight Java or C++ code. If you're interested in experimenting with Celtix JavaScript/E4X service implementations, the code will be available from http://celtix.objectweb.org by the time you read this. By then, I'll also likely have added support for JavaScript/E4X on the client side. In the near future, I also hope to add similar support for Jython (www.jython.org) clients and services to Celtix.

References

1. *ECMA-262, ECMAScript Language Specification*, 3rd ed., ECMA Int'l, Dec. 1999; www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.
2. R. Chinnici, et al. "The Java API for XML Web Services (JAX-WS) 2.0," proposed final draft standard, Sun Microsystems, 7 Oct., 2005; www.jcp.org/en/jsr/detail?id=224.
3. *ECMA-357, ECMAScript for XML Specification*, 2nd ed., ECMA Int'l, Dec. 2005; www.ecma-international.org/publications/files/ECMA-ST/Ecma-357.pdf.
4. J. Waldo et al., *A Note on Distributed Computing*, tech. report SMLI TR-94-29, Sun Microsystems Labs, 1994; www.sunlabs.com/technical-reports/1994/abstract-29.html.
5. S. Vinoski, "It's Just a Mapping Problem," *IEEE Internet Computing*, vol. 7, no. 3, 2003, pp. 88–90.
6. S. Vinoski, "RPC Under Fire," *IEEE Internet Computing*, vol. 9, no. 5, 2005, 93–95.
7. S. Loughran and E. Smith, *Rethinking the Java SOAP Stack*, tech. report HPL-2005-83, Hewlett-Packard Bristol Labs, May 2005; www.hpl.hp.com/techreports/2005/HPL-2005-83.html.

Steve Vinoski is chief engineer for IONA Technologies. He's been involved in middleware for more than 17 years. Vinoski has helped develop middleware standards for the Object Management Group (OMG) and the World Wide Web Consortium (W3C). Contact him at vinoski@ieee.org.