In today's world, we see Web Services (WS) in action everywhere from online shopping and shipping trackers to mapping and geolocation tools. The methods used to create WS are just as broad and varied as the tools that use WS. However, some of the protocols defined by the WS- documents seem to exist only for Oracle to sell consulting contracts. One can easily drown in the alphabet soup of UDDI, WSDL, and SOAP. Learning SOAP (Simple Object Access Protocol) reveals there is nothing simple about it. Add to that the fact that Java and .net are incompatible in subtle ways, which creates extra complexity. REpresentational State Transfer (REST) eliminates much of the formality imposed by a WSDL-style WS.

How does REST avoid complexity? First, REST is stateless. Statelessness alone won't make WS simple. But, leveraging the Internet's most popular stateless protocol, Hypertext Transfer Protocol (HTTP), simplifies WS even more. Statelessness and HTTP let us take advantage of many existing tools. The whole spectrum of Internet infrastructure helps us implement REST-based WS. From Java libraries that provide valid HTTP sessions to caching web proxies, all of the infrastructure on the Web works for us.

By Jason Foutz
Software Programmer

What exactly is a stateless protocol? Imagine a stateless bank teller. A conversation might go something like this:

Jason: "Excuse me, ma'am, what is my balance?"
Teller: "I'm sorry, sir, for what account?"
Jason: "Of course. What is the balance of account 1234?"
Teller: "I'm sorry, who is the account holder for account 1234?"
Jason: "What is the balance of account 1234 held by Jason?"
Teller: "I'm sorry, what is the password for account 1234 held by Jason?"
Jason: "What is the balance of account 1234 held by Jason with password GetItAlready"
Teller: "The balance is 37 cents."

Stateless services require all the information about a request up front. Providing partial information will not return a valid reply so each request sends whatever the server might need in order to get a valid reply. The bank itself is stateful. They, of course, need to remember all of their customers and balances. Each request to the bank is a self-contained package, requiring no extra state. Statelessness simplifies the client and server code greatly. In fact, it's so simple we can use HTTP instead of writing our own SOAP-like system.

How, exactly, is information sent to the web server? A web browser uses HTTP to talk to a web server. Every click of a link on a web page sends an HTTP request to the server and returns an HTTP response back to the client. The protocol is a little bit too complicated to just type into a telnet session, but it is not nearly as hard as the alphabet soup of WSDL services. There are four key HTTP methods used in REST-based WS: GET, PUT, DELETE, and POST.

Every query asked of our fictional bank teller was a GET request. Requesting balance information GETs information about the bank account, a specific resource. In SQL, a SELECT statement is like the GET request. GET requests are made visible on many websites. At www.google.com, searching for something adds on many parameters about the request. It might look something like www.google.com/search?query=something. The parameters vary depending on the exact query, but the arguments display inside the address bar of the browser.

A PUT request replaces existing state on the server. Our fictional bank teller would be able to change a password with a valid PUT request. A PUT request is similar to the SQL UPDATE statement. Furthermore, PUT requests should be idempotent. Whether using PUT to change the password to "hello" once or 100 times, the password should be "hello" after every request. Intentionally, or accidentally, sending the request multiple times shall have no effect.

A DELETE request does just what it sounds like it does. DELETE removes state from the server. HTTP DELETE works like an SQL DELETE statement. Because it changes state, DELETE should also be idempotent. For example, deleting a specific account multiple times has no effect, no additional accounts are deleted.

The final HTTP method, POST, creates resources. POST is similar to the SQL INSERT statement. Every POST, creates a new resource, so it can not be idempotent. Depositing a dollar into an account POSTs that dollar to the teller. Every single dollar deposited has an effect, to create a resource on the server.

So how would we actually implement a REST Web Service? The BBj® Servlet API provides full access to HTTP. This introduction just scratches the surface of what is possible with BBjServlet. The BBj Servlet Overview in the BASIS online docs at links.basis.com/servlet covers creating and handling HTTP requests using BBj Services. In the following examples, getMethod is the workhorse for determining what kind of request the user makes. A lot more functionality is available in BBjHttpRequest and BBjHttpResponse in the online help at links.basis.com/basishelp.

When the server receives an HTTP request, the getMethod() determines which HTTP method the client used. You can implement each method GET, PUT, DELETE or POST to handle the client's requests. Since not every WS requires every method, just omit unnecessary methods. Clients must provide all the information possibly needed up front because HTTP is stateless.

Code for transaction processing might look something like this:

```
method public void transactions(BBjServletEvent p_event!)
   request! = p_event!.getHttpRequest()
   method$ = request!.getMethod()
   if "GET" = method$ then
      #showTransactions()
   endif
   if "POST" = method$ then
      #createTransaction(request!)
   endif
methodend
```

This type of transaction cannot be changed or removed so there is no need to implement PUT or DELETE.

A REST-based WS can be consumed with the http-commons library. Code for processing a request might look like this:

```
url$ = "http://server:8888/servlet/example"
client! = new org.apache.commons.httpclient.HttpClient()
request! = new org.apache.commons.httpclient.methods.GetMethod(url$)
client!.executeMethod(request)
```

REST services requires only HTTP from their clients. This makes it exceptionally easy to consume services from many different languages. BBj and Java, as we saw, may use the http-commons library. C# ships with an HttpClient class right in the .net runtime. While PHP has nothing built in, many third party http client libraries are available. HTTP is so widely used, command line libraries such as curl or wget could be used from any language's version of SCALL. C programmers might do an exec of curl to retrieve their WS results. Libraries for interacting as an HTTP client are widespread, virtually every platform and language can call a WS.

Summary

While WS provide unlimited complications, REST WS limit that complexity in two key ways. REST is stateless, requiring clients to provide all the information a server might need for every request. REST leverages HTTP, simplifying the stateless communication to the server. REST is a powerful tool for deploying WS without the risk of drowning in alphabet soup. So rest easy, Keep It Simple and Stateless, and 'KISS' WSDL goodbye! ■

- Review getMethod in the online BASIS documentation at links.basis.com/getmethod
- See also
  - Detailed examples of using the Java library at bit.ly/euhVG
  - Sample API of accessing another's Web services at bit.ly/JIywat
  - Vast list of REST Web Services at bit.ly/9XXsrF