Writing a Web Service in BBj

By David Wallwork

This article focuses on developing a Web service provider for BBj® programs. See the Q1 2003 issue of the Advantage magazine for details about writing a BBj Web services consumer. A Web service must follow certain standards so that the deploying Web server knows how to serve it to the Web service consumer. There are two steps required to offer a Web service. First, write the service and second, deploy the service. This article explains how to write a Web service that allows the service consumer to execute code written in BBj, offered by the Web service. Because BBj runs in a Java runtime environment, BASIS implemented the Java Web services paradigm.

A Simple BBj Web Service

The BBj 3.0 release includes the following two classes:

```
com.basis.bbj.webservice.BBjWS
com.basis.bbj.webservice.BBjWSImpl
```

These classes allow BBj to offer a Web service. Offering a Web service enables other developers to write Web service consumers in other languages (VB, C++, BBj, .NET, etc.) that can run any BBj program offered by the Web service. Although this approach is certainly easy, the developer may want to do some additional work to

- Control which BBj programs the consumer runs
- Allow the consumer to pass information into the BBj programs
- Allow the consumer to receive information back from the BBj programs

Controlling Which BBj Programs the Consumer May Invoke

Although the classes `BBjWS.java` and `BBjWSImpl.java` enable BBj to run as a Web service, this is not the standard method for deploying a Web service. The Web service developer should provide method signatures that are meaningful to the consumer of their service, rather than an interface that offers to run a BBj program. Additionally, developers should extend the classes `BBjWS.java` and `BBjWSImpl.java`, providing descriptive methods for the developers of the consuming applications. These methods can then invoke BBj code on behalf of the consumer. This way, the developer controls which BBj programs the consumer can run.

For example, if the consumer of BBjAuction invokes the `BBjAuction.listAuctionItems()` method, then the BBj program calls `auction_getItems.bbj`. However, BBjAuction controls which BBj programs the consumer application invokes. Now, the consumer of BBjAuction can only run a BBj program through the interface offered by BBjAuction.

Data Exchange From the BBj Program's Perspective

The BBj program receives a limited amount of data from the Web service by inspecting ARGV(). If the Web service requires more data than possible or reasonable via the command line parameters, or if the Web service needs to obtain response data from the program, then the BBj program should:

- OPEN a bridge channel
- READ from that channel as though it were a single keyed MKEYED file to obtain data from the Web service

- WRITE to that channel as though it were a single keyed MKEYED file, or as though it were a STRING file in order to return data to the Web service

After the BBj program ends, the Web service may access any data written to the bridge channel.

Data Exchange From the Web Service's Perspective

The BBjWS interface exposes two methods:

```
public byte[] invokeBBjReturnBytes(String p_programName,
                                   String p_programParams,
                                   HashMap p_keyValues,
                                   int p_timeoutInSeconds) throws
RemoteException;
public HashMap invokeBBjReturnMap (String p_programName,
                                   String p_programParams,
                                   HashMap p_keyValues,
                                   int p_timeoutInSeconds)throws
RemoteException;
```

The syntax for **p_programParams** is the same as the syntax for the command line parameters when invoking a BBj program from the command line. If **p_programParam** ends with "..- argv1 argv2 argv3", then the BBj program has access to argv1, argv2, and argv3 using the ARGV() function.

If **p_keyValues** is not null, then the BBj program accesses each key-value pair in **p_keyValues** by reading the **bridgeChannel**. During its execution, if the BBj program writes information to the bridge channel, then the Java code of the Web service can access that information. If the BBj program writes to the bridge channel as though it were a String File, then the Java code uses the method **invokeBBjReturnBytes()** to retrieve the output of the BBj program. Alternatively, if the BBj program writes to the bridge channel as though it were an MKEYED file, then the Java code uses the method **invokeBBjReturnMap()** to retrieve the output.

Hello World Web Service

The following **HelloWebService** program:

- Allows the consumer to run only one program
- Accepts the user's name as input
- Returns a greeting to the user

The developer needs to write two files: **Hello.java** and **HelloImpl.java**. Once deployed, these files enable the developer to offer the Web service. In addition, the developer needs to write **Hello.bbj**, which is the BBj program that our Web service invokes.

Hello.java

```
import java.rmi.*;
public interface Hello extends java.rmi.Remote
{
    String sayHello(String name) throws RemoteException;
}
```

HelloImpl.java

```
import java.rmi*;
public class HelloImpl extends com.basis.bbj.webservice.BBjWSImpl implements
Hello
{
   String sayHello(String name)throws RemoteException
   {
      String params = name;
      byte answer[] = invokeBBjReturnBytes("Hello.bbj", params, null,
2000);
      return new String(answer);
   }
}
```

Hello.bbj

```
Bridge = unt
REM  config file must contain a bridge alias for J0 see JavaBBjBridge docs
OPEN (bridge) "J0"

IF argc < 2
        Answer$ = "hello world"
ELSE
        Answer$ = "hello " + argv(1)
ENDIF

WRITE(bridge)answer$

RELEASE
```

With BASIS's Web service, the consumer no longer runs an arbitrary BBj program. The only option for the consumer is to invoke the method **sayHello()**. The Web service method **HelloServiceImpl.sayHello()** obtains its return value by invoking **Hello.bbj**. Additionally, this Web service requires only a small amount of Java code for implementation. Of course, it does not do much.

The next example is BBjAuction, which uses complex data types and multiple BBj programs to offer more meaningful methods to the consumer of the Web service. The following section returns to the BBjAuction, demonstrating how to pass more data than is practical on the command line.

Passing More Data

The BBjAuction Web service allows the consumer to logon to the system to obtain a listing of biddable items, bid on an item, and discover which item(s) the user won. Below, we examine the method **BBjAuctionImpl.listActiveItems()**. Listed below are the relevant portions of the code. See the complete code on the enclosed CD.

Complete code of the method **BBjAuctionImpl.java**

```
public AuctionItem[] listActiveItems(String p_loginID)throws
java.rmi.RemoteException
{
        HashMap map = invokeBBjReturnMap("Auction_getItems.bbj",
p_loginID, null, 30);
        return convertToAuctionItems(map);
}
```

Relevant portions of **Auction_getItems.bbj**

```
0050    clientChannel = unt
0090    open (clientChannel)"J0"

0110    call "auction_util.bbj::initialize"

0130    call "auction_util.bbj::makeItemTemplate", item$

0150 index = 0
0160 findRecord(dataChannel,knum=2,key="0",err=finished)item$

0170 while 1
0210    call "auction_util.bbj::validateRecord", dataChannel,
bankChannel, item$
0220    closed = item.biddingClosed

0230    if !closed
0240      call "auction_util.bbj::templateToItem", item!, item$
0250      closed = item!.getBiddingClosed()
0260      value$ = techcon03.AuctionItem.toByteString(item!)
0270      writeRecord(clientChannel, key=str(index))value$
0280      index = index + 1
0290    endif
0300    readRecord(dataChannel, end=finished)item$
0310 wend
```

auction_getItems.bbj iterates through the data file, reading each record into the templated string **Item$**. At line 240, the program uses **Item$** to create a Java Object of type **techcon03.AuctionItem**. At line 260, the code converts **AuctionItem** to bytes and places it into the string value$. At line 270, the code writes a key-value pair to the bridge channel.

auction_getItems.bbj continues until it writes all "open for bidding" items to the bridge channel. Furthermore, **BBjAuction.listActiveItems()** receives a HashMap as the return value of its call to **invokeBBjReturnMap("auction_getItems.bbj:", p_loginID, null, 30)**.

Recall that the key-value pairs written by **auction_getItems.bbj** originated from strings. **BBjAuction.listActiveItems()** calls to **BBjAuctionImpl.convertToAuctionItems**, which converted the map of strings into an array of AuctionItems. The program then returns that list to the Web service consumer.

Passing Complex Data Types Between Web Service and BBj Program

When a BBj program reads input from its bridge channel or writes to its bridge channel, the program is only able to read or write bytes. Web service provides support for much more complex data types. If the developer can convert (serialize) those data types to bytes, the developer can pass them between the Web service and the BBj program.

The following lines from **auction_getItems.bbj** illustrate this example:

```
0240        call "auction_util.bbj::templateToItem", item!, item$
0260        value$ = techcon03.AuctionItem.toByteString(item!)
0270        writeRecord(clientChannel, key=str(index))value$
```

Line 240 converts the templated string **Item$** to an instance of the Java class **AuctionItem**. Line 260 converts the **AuctionItem** to a string and line 270 writes the string to the bridge channel. Consequently, **BBjAuctionImpl** calls **invokeBBjReturnMap()** and does not need to understand templated strings. The map that it receives contains serialized **AuctionItems**, rather than serialized templated strings. **BBjAuctionImpl** deserializes these **AuctionItems** and uses them as the return value for the methods invoked by the Web service consumer.

The conversion from a templated string to an **AuctionItem** uses the following code:

From **auction_util.bbj**

```
0330 templateToItem:
0340   ENTER item!, A$
0350   item! = new techcon03.AuctionItem()
0360   item!.setItemID(num(A.id$))
0370   item!.setDescription(A.description$)
0380   item!.setHighBid(A.highBid)
0390   item!.setHighBidder(A.bidder$)
0400   item!.setNumberBids(A.numBids)
0410   now = int(new java.util.Date().getTime()/60000)
0420   remaining = A.endTimeMinute - now

0500   item!.setMinutesRemaining(remaining)
0510   item!.createString()
0520   techcon03.AuctionItem.setBiddingClosed(item!,A.biddingClosed)
0530 exit
```

One additional conversion remains. The method **invokeBBjReturnMap()** returns a Java HashMap. Each value in the HashMap contains bytes that are the serialization of an **AuctionItem**. However, we want to return an array of **AuctionItems** to the consumer of the Web service. Looking back at the code for **BBjAuctionImpl.listItems()**, notice that it calls **BBjAuctionImpl.convertToAuctionItems()**. See the code for this final conversion below.

From **techcon03.BBjAuctionImpl.java**

```
private AuctionItem[] convertToAuctionItems(HashMap p_map)
{
        int size = p_map.size();
      AuctionItem ret[] = new AuctionItem[size];

      String key;
      String value;
        Iterator next = p_map.keySet().iterator();
      while(next.hasNext())
      for(int q+0;q{
           key = (String)next.next();
           value = (String)p_map.get(key);
           ret[q] = AuctionItem.fromByteString(value);
        }
        Arrays.sort(ret,getSorter());
              return ret;
}
```

This code iterates over the Map returned by invoking **BBjReturnMap()** and converts each value to an **AuctionItem** by calling **AuctionItem.fromByteString()**. Furthermore, this code demonstrates how to exchange Java Objects between a BBj program and the Web service. The only requirement is knowing how to change that Java Object to bytes and back again.

Increased Security and Functionality

BBj 3.x provides the classes necessary for the immediate deployment of BBj programs as a Web service. By writing a minimal amount of Java code, a developer can write a Web service that is more secure and functional than the provided classes. Because BBj can create and manipulate Java objects, BBj programs and Web services can exchange complex data structures giving BBj programmers the ability to write very sophisticated BBj powered Web service providers. For more information about BBj and Web services, see http://www.basis.com/advantage/mag-v7n1/webservices.html .

Click HERE for the "Writing a Web Service in BBj" source code.