Wash up With SOAP Web Services

Now with rich data structures and authentication

BBj® web services received a facelift, which allows you to build web services that handle more complex data structures as well as collections of records. This article presents a brief tutorial on how to create a SOAP web service that uses custom Java types in the parameter set to extend your web service offerings. We have also added Basic authentication access and will show you how to use it. At the end of this article you will find an URL where you can download the examples.

An Overview on Enhanced Data

You can design a typical BBj web service to have parameters of the standard BBj data types BBjString, BBjNumber, and BBjInt, but you may require a greater variety of types. Enhancements to BBj 14.11 now make it possible to create a web service that uses Java types on the parameter list, providing a richer web service capability.

Let's suppose that we want to publish a web service that deals with customers, but the customer record is rather complex, and the main customer record has sub records for the address and the contacts**. Figure 1** shows how this might look.

In our structure, *Address* and *Contact* are complex subtypes, and there could be many contacts records per customer. Okay, so maybe it's not that complex but it does serve to illustrate the point.

Customer		Address		Contact	
BBjInt	ID	BBjString	Street	BBjString	First Name
BBjString	Name	BBjString	City	BBjString	Last Name
Address	Address	BBjString	Zip Code	BBjString	Job Title
BBjVector	Contacts	BBjString	State	BBjString	Phone
				BBjString	Email

Figure 1. Record structure for Customers

To achieve our goals of having a web service that can deal with this data structure, we're going to have to complete several steps.

1. Create the main web service BBj application.
2. Configure the web service in Enterprise Manager.
3. Create a set of JavaBeans that represent the data structure.
4. Configure a classpath for the Java classes.

Additionally, we're probably going to need to do the following:

5. Create utility methods for filling the Java data structures.

Richard Stollar
Software Developer

Before we get into all of that, let's look at how the web services work to understand why we need to do this.

BBj Web Service Classes and Deployment

BBj web services rely on *wsgen*, which comes with Java. The *wsgen* tool parses the web service implementation class and generates the required files for web service deployment.

For *wsgen* to function, it needs to have access to the required Java classes through a classpath that's used during execution.

You have two real options for how to provide *wsgen* with the classes needed.

Option 1: Create a .jar file that contains the classes.

Option 2: Add the folder where the classes were generated in the classpath.

For many reasons, including the fact that a .jar file is far more portable that a whole bunch of class files, Option 1 is the preferred method when dealing with deployment. However, during the development phase, Option 2 is easier as you don't need to recreate the .jar file whenever the code changes.

So, you'll have a project in Eclipse that holds the Java source files and the corresponding class files will be generated into a bin folder, and this is what we're interested in. For example, **`C:\Work\Examples\JavaBeans\bin\`**, but it may vary for you based on the location of your project's bin folder.

Crossing the Bridge from BBj to Java Web Service

When you develop your web service in BBj, there's a certain amount of magic that goes on behind the scenes. Your BBj program, however simple or complex, needs to be wrapped up with a Java implementation. Through Enterprise Manager, you specify the prototypes for your web service's methods. These prototypes are used to generate a Java web service for your BBj code. That front-end web service is sent to *wsgen*; as mentioned earlier in this article, to generate all the classes. Finally, the web service is deployable.

Creating the Main Web Service

Writing the web service's code is beyond the scope of this article and much depends on your requirements. In general terms, it is a BBj program with one or more entry points that each perform some part of the web service's functionality.

Figure 2 shows a small piece of BBj code for a **`getCustomer()`** method in a web service.

```
getCustomer:
    enter customerId%, customer!
    customer! = new Customer()
    customer!.setCustomerID(1)
    customer!.setName("ACME")
    address! = new Address("1 The Street", "The City", "NJ", "52444")
    customer!.setAddress( address! )
    vect! = new BBjVector()
    barny! = new Contact("Barny", "Rubble", "Rock Breaker", null(), "barny@bedrock.com")
    fred! = new Contact("Fred", "Flintstone", "Site Foreman", null(), "fred@bedrock.com")
    vect!.add ( barny! )
    vect!.add ( fred! )
    customer!.setContacts( Util.makeContactArray(vect!.toArray()) )
    exit
```

Figure 2. Sample web service code

This sample creates a customer record with static data, adds the address, and creates a vector containing two contact records which it then converts to an array using a utility class and adds to the customer record. It completes a record with fixed data whereas your web service will most likely be database driven.

Configure the Web Service

Configuring the web service is done through Enterprise Manager as normal, at least up until the point where you need to specify the parameters for your service methods. Rather than selecting the type from the dropdown list you can enter the fully qualified name

of your class as the parameter type. In our example the fully qualified class name is **com.acme.beans.Customer**. **Figure 3** shows this in action.

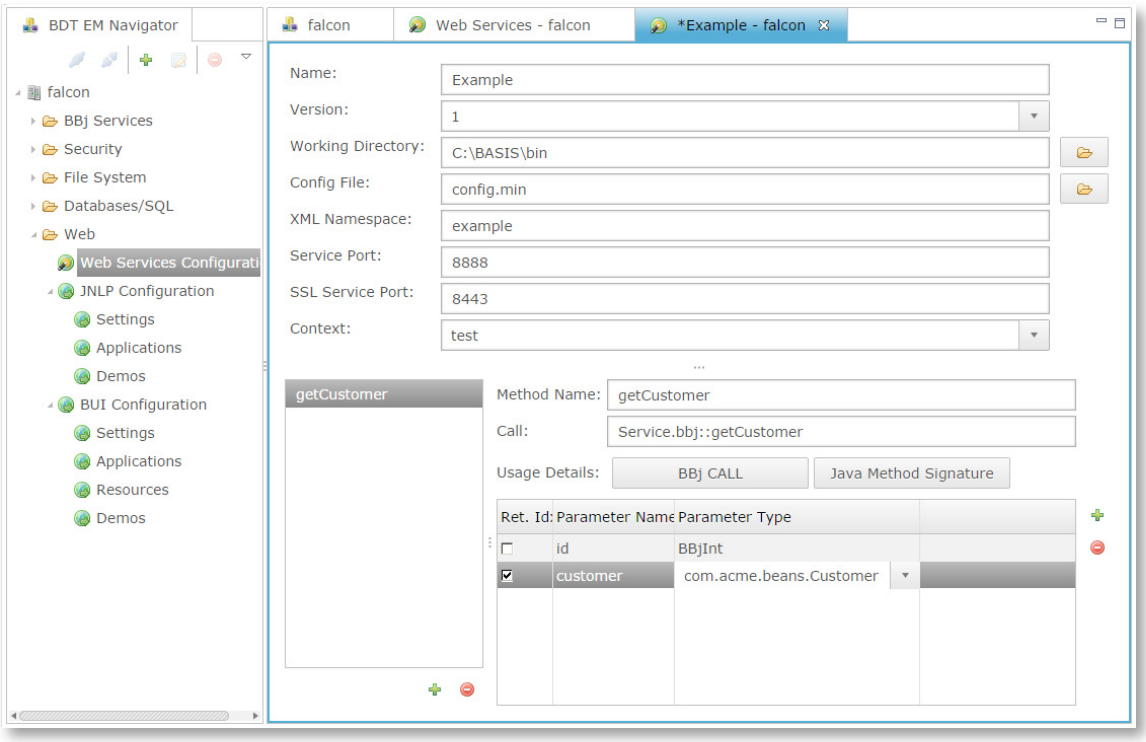

Figure 3. Specifying a custom class

Creating the Java Classes

We need to create a set of JavaBeans that represent our data structure. A JavaBean is a special type of class that encapsulates many objects into a single object (the bean) and, unless your classes conform to the JavaBeans specification, you won't be able to use them in your web service.

Again, I don't want to go too deep into explaining how you should go about creating JavaBeans as the downloadable example should give you all you need. It's beyond the scope of this article to cover how you should write the JavaBeans for the data, just remember that the classes must be serializable, have a zero-argument constructor, and allow access to properties using accessor (getter and setter) methods; this makes them a bean. **Figure 4** shows sample BBj code for the Customer record.

```
package com.acme.beans;

import java.io.Serializable;

public class Customer implements Serializable {
    private Integer m_customerID;
    private String m_name;
    private Address m_address;
    private Contact[] m_contacts;

    public Customer() {
        // Mandatory zero-parameter constructor.
    }

    public Customer(Integer p_customerID, String p_name,
            Address p_address, Contact[] p_contacts) {
        m_customerID = p_customerID;
        m_name = p_name;
        m_address = p_address;
        m_contacts = p_contacts;
    }

    public Integer getCustomerID() { return m_customerID; }
    public void setCustomerID(Integer p_customerID) { m_customerID = p_customerID; }
    public String getName() { return m_name; }
    public void setName(String p_name) { m_name = p_name; }
    public Address getAddress() { return m_address; }
    public void setAddress(Address p_address) { m_address = p_address; }
    public Contact[] getContacts() { return m_contacts; }
    public void setContacts(Contact[] p_contacts) { m_contacts = p_contacts; }
}
```

Setting Up the Classpath

You can choose to add a .jar file to your classpath through Enterprise Manager. To do this:

1. Pick BBj Services > Java Settings from the left menu.
2. Select the 'Classpath' tab.
3. Click the [+] icon above the 'Classpath Names' list.
4. Enter a meaningful name for the classpath entry; I chose **service**.
5. Click [OK] to save it.
6. Ensure that your newly created classpath entry is selected and then click the [Add a Jar] icon above the 'Classpath Entries' list (to the right).
7. Locate the jar file and select it with a double-click or by selecting it and clicking [Open].

Finally you need to add the same entries to the <default> bbj classpath entry and here's how:

8. Select the <default> classpath entry in the list of classpath names.
9. Click the [Add a Jar] icon above the 'Classpath Entries' list, locate the jar file and select it as before.

Alternatively, you can manually add the classes folder to your classpath in your **bbj.properties** file. It's important to remember that all classpath entries must begin with **basis.classpath**. Here is an example of what it should look like:

```
basis.classpath.service=C\:\\Work\\Examples\\JavaBeans\\bin
```

Adding the Classpath to the Context

BBj 14.11 introduced contexts for the deployment of your applications, which is outside of the scope of this article but there is more information in this issue's *Don't Put All of Your Jetty Eggs in One Context*.

You'll need to decide the context to which we're deploying our web service and add the appropriate classpath element to its configuration in **jetty.xml**. For now, let's deploy our web service to the main bbj root context and add the classpath element to the context entry shown in **Figure 5**.

```
<bbj docbase="$basis_home/htdocs" path="/">
    <classpath>service</classpath>
    <welcome-file>index.html</welcome-file>
</bbj>
```

Figure 5. Adding the classpath to the context

Adding the classpath to the context is important as the web service generation will fail without it because this classpath entry is passed to *wsgen*.

Dealing With Collections

Our **Customer** record has many **Contact** records. From BBj's perspective, these contacts are held in a BBjVector but when you want to fill the Java object with the contents of the BBjVector, you'll need to transform the data into an array of objects of the specific class. Enter the need for a utility class I mentioned earlier that will handle this.

Using a utility class on the Java side can be a valuable tool as you can create all sorts of standard methods for converting data. When you have a collection of objects you'll probably need to have a suitable converter similar to the one shown in **Figure 6** which transforms an array of objects into an array of **Contact** objects.

```
public class Util
{
    public static Contact[] makeContactArray(Object[] objects)
    {
        return Arrays.copyOf(objects, objects.length, Contact[].class);
    }
}
```

Figure 6. Converting an array of objects into an array of **Contact** objects

Your BBj program will use this method to convert the BBjVector into an array of **Contact** objects as follows:

```
customer!.setContacts(Util.makeContactArray(vect!.toArray()))
```

Add Basic Authentication

Authenticating clients to a web service can be a rather involved process but thankfully doesn't have to be. If you use the HTTPS protocol for deploying your web services then Basic authentication (introduced for preview in 14.12) should be sufficient and it is widely used. The username and password are encoded in the HTTP request header and then validated server-side.

Applying validation to your web service requires you to write a custom routine in your web service implementation which has three parameters. The input parameters are username, password and a response to indicate if validation was successful or not. **Figure 7** shows a simple implementation, but in a real world you might be validating the user through a database or some other source.

```
myAuth:
    enter user$, pass$, auth%
    auth%=0
    if user$="admin" AND pass$="admin123"
        auth%=1
    endif
    exit
```

Figure 7. Example authentication routine

The final step in this process is to configure the authentication routine through Enterprise Manager. Simply enter the name of your authentication routine in your web service's 'Authentication' field as shown in **Figure 8.** In this example, that would be **myAuth**. Finally, deploy your web service and away you go!

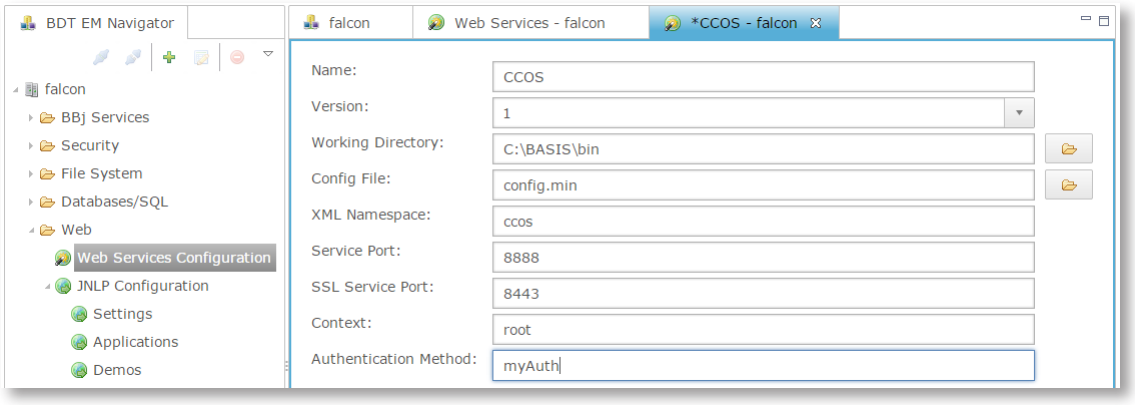

Figure 8. Setting the authentication in Enterprise Manager

Configuring the client application is beyond the scope of this article, but most SOAP test tools provide ways to supply credentials for Basic authentication.

Remember that Basic authentication sends the username and password in a non encrypted form. It is by itself very insecure, but becomes secure when used in conjunction with the HTTPS protocol.

Summary

As you have read, creating more complex web services is much easier as you can now take advantage of rich SOAP web services to send and receive complex data structures as well as collections of records. Your web services will be easily accessible to a range of client technologies and conform to well-known standards. ■

- Read *Don't Put All of Your Jetty Eggs in One Context*
- Download and run the code samples