From Legacy to Enterprise With BBj Web Services

Old is new again!

To continue with this article, download and install the demos from the Optional File section in the BBj download at www.basis.com/products/bbj/download.html. Then select **BBj | Demos | LaunchDock** from the BASIS folder to run the demos as noted in the instructions below.

Running the Chile Company Web Service Demo

1. Verify that BBj Services is running on a Java 6 JDK; we recommend that you also run the Thin Client Proxy Server on a Java 6 JDK.

2. Run the LaunchDock
- On Windows, select **Start | All Programs | BASIS | BBj | Demos | LaunchDock**
- On Mac in the Finder, select **Macintosh HD | Applications | BASIS-LaunchDock**
- On UNIX/Linux distributions in a shell session, change to the `<BBj installation>/bin directory` and type `./LaunchDock`

3. Choose the Language/Interpreter folder. Click the Web Service demo icon that appears as a blue globe with two computers connected with a lightening bolt in **Figure 1**. >>

By Shaun Haney
Quality Assurance Engineer

Figure 1. The "ChileCompany Web Service" demo in the BASIS LaunchDock

The first window that appears displays a tab set with the "Place Order for Customer" tab selected. The grid inside the tab, shown in **Figure 2**, contains inventory data from the Chile Company Web Service.

Figure 2. The order entry screen

In the QUANTITY column, specify the quantity of each item to be ordered. Select the customer from the customer combobox, then press [Place Order] to submit. A confirmation dialog appears (see **Figure 3**), listing information such as the order number that the Chile Company Ordering System assigned. Press [OK] and the dialog disappears. The demo client now queries the Web Service to update its inventory and invoice list.

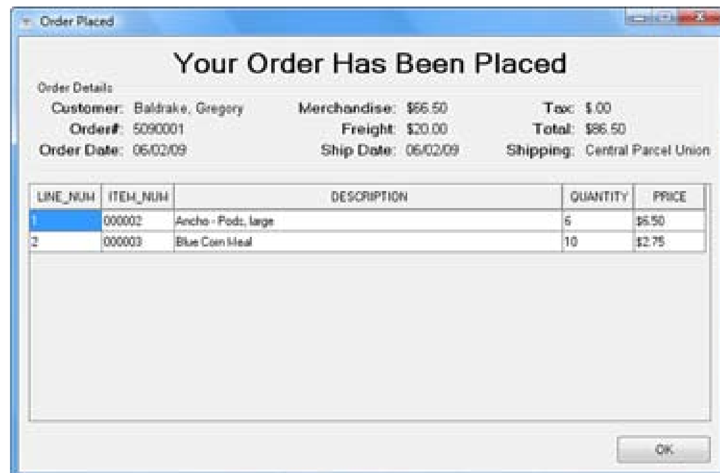

Figure 3. The order confirmation dialog

Next, click the Invoices tab in the main window (**Figure 4**); the order you just placed lists a corresponding invoice.

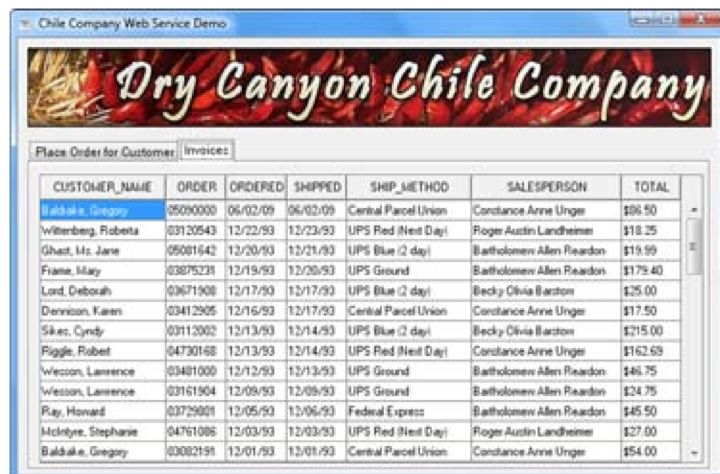

Figure 4. Invoice listing showing the most recent purchases

Exploring the CCOS Web Service

The "Chile Company Web Service" demo shows our client consuming a Web Service written in BBj. What you have seen so far is the client program that handles the user interface. The brains of the operation, however, are in the service that runs on BBj Services' internal Jetty Web Server. BASIS conveniently included the Chile Company Ordering System (CCOS) Web Service configuration inside the Enterprise Manager (EM). By examining the configuration of the CCOS Web Service, you will see how easily you can set up your own Web Service.

The Web Service Program

To begin, find the "Chile Company Web Service" demo directory at **`<BBj Installation>/demos/webservices`**. The **`ServerSide_Legacy.src`** program file contains all the server-side code for the CCOS Web Service. This file is not >>

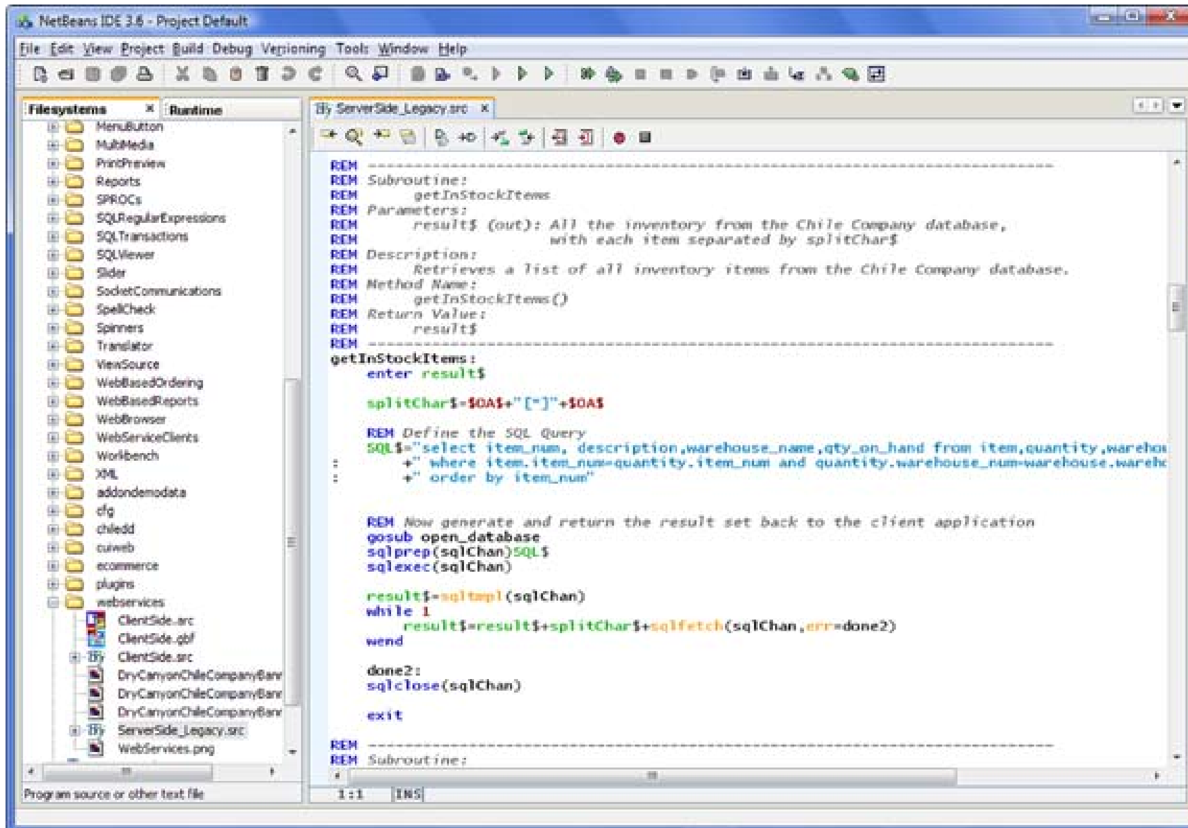

Figure 5. A program listing of the subroutines in **`ServerSide_Legacy.src`**

a stand-alone program, but rather a library of BBx®-style subroutines shown in **Figure 5**.

The Web Service ignores any code in the file that is not contained within a subroutine. Each subroutine consists of a label and an ENTER statement with the list of parameters that each subroutine takes. Additionally, the Service treats one of these parameters as the return value for the subroutine and all other parameters as pass-by-value parameters.

Because the server-side program consists of traditional BBx-style subroutines and does not contain any object-based code or Web Services artifacts, it is possible to transition your old code to run in a BBj Web Service and still continue to use it in its prior capacity.

The CCOS Web Service consists of seven subroutines:

1. **`login`** authenticates users of CCOS and allows anyone into the system and is not ever actually called in this demo, however, developers could easily add code to the routine to authenticate their employees, making the system more secure

2. **`getOrders`** retrieves a list of customer orders from the **`ChileCompany`** database, then appends these orders to a string, separated by an easily identifiable separator string of our choice: **`"<carriage return>+[*]+<carriage return>"`**

3. **`getInStockItems`** retrieves a list of all the items the Chile Company sells, including their quantity and warehouse location, and appends the results to a string with the separator string between each entry

4. **`getOrderDetail`** retrieves all the line items for a given order number, then appends them along with the separator string

5. **`getOrderHeader`** retrieves the order header for a given order number

6. **`getCustomerList`** retrieves the list of customers in the **`ChileCompany`** database

7. **`placeOrder`** accepts an order for a given customer and list of items, and updates several tables in the **`ChileCompany`** database with information from the order

   The Web Service expects the list of items and quantities to be a string with each item in the order separated by the separator string. For simplicity, we assume the customer has already paid and so we will update the customer's information with the total amount of the order, etc. We also update the number of items in stock and the list of invoices as well.

The subroutines in the CCOS server-side code primarily access the ChileCompany database with preset queries.

Exploring the CCOS Web Service Configuration in Enterprise Manager

The Web Service is already configured in the EM. To see this configuration, follow these steps:

1. Launch EM.
   - On Windows, select **Start | All Programs | BASIS | BBj | Enterprise Manager**
   - On Mac in Finder, select **Macintosh HD | Applications | BASIS-Enterprise_Manager** >>

- On UNIX/Linux distributions, type the following at the shell prompt:
  `<BBj Installation>/bin/enterprisemanager`

2. Log in to EM (User Name: **`admin`** Password: **`admin123`** unless the person who installed BBj changed the username and password)

3. Select "Web Service Configuration" from the left pane as shown in **Figure 6**.

4. Select the CCOS Service and click [Edit Service] to display a list of methods (see **Figure 7**).

Figure 7 fully defines the configuration for the CCOS Web Service, including:

- **The name of the service** becomes read-only once the service after its creation.

- **The working directory** is the directory where the program files for the Web Service will reside.

In this case, we left the working directory blank because the directory containing **`ServerSide_Legacy.src`** is already in the PREFIX defined in config.min. We could have also specified **`<BBj Installation>/demos/webservices`** as our working directory. For a given Web Service, you can have any number of program files with callable subroutines. Those program files need to reside in the working directory specified or in the PREFIX defined in the configuration file.

- **The "Config File" field** takes the location of the Web Services' configuration file.

In this case, we specified **`config.min`** located in **`<BBj Installation>/cfg`**. Any relative path specified here will be relative to **`<BBj Installation>/cfg`**. The path must otherwise be an absolute path.

- **The "Namespace" field** specifies the namespace for your service; the package name for the client's service and port classes will be the namespace name in all lowercase. If this field is left blank, the package name for the service and port classes will be "example".

- **A list of methods for the CCOS Web Service** appears in the listbox at the bottom of this window. For easy correlation, each method has the same name as a subroutine in **`ServerSide_Legacy.src`**. Though it is good practice to name the methods this way, it is not required.

To see how the methods correlate to the subroutines in **`ServerSide_Legacy.src`**, select the "placeOrder" method and click [Edit Method]. The result appears in **Figure 8** where the configuration for the subroutine "placeOrder" is shown alongside the subroutine header for placeOrder in **`ServerSide_Legacy.src`**.

The "Edit Method" window in **Figure 8** is where the actual method correlates to its subroutine. >>

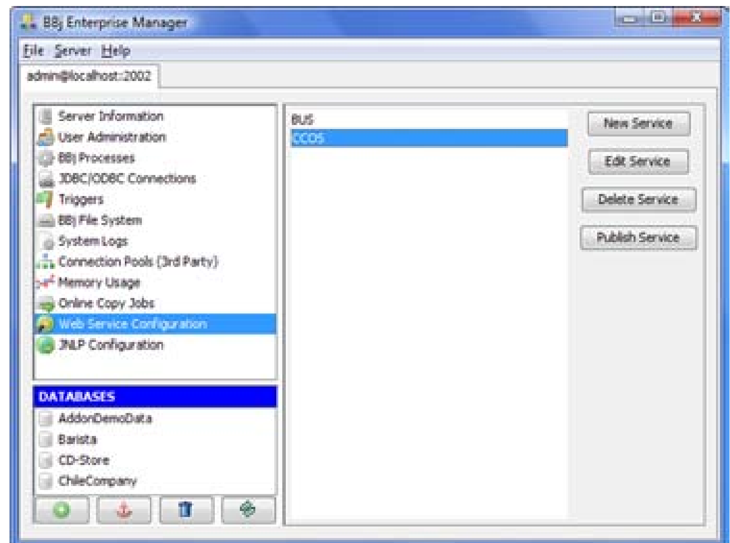

Figure 6. A listing of Web Services defined in EM

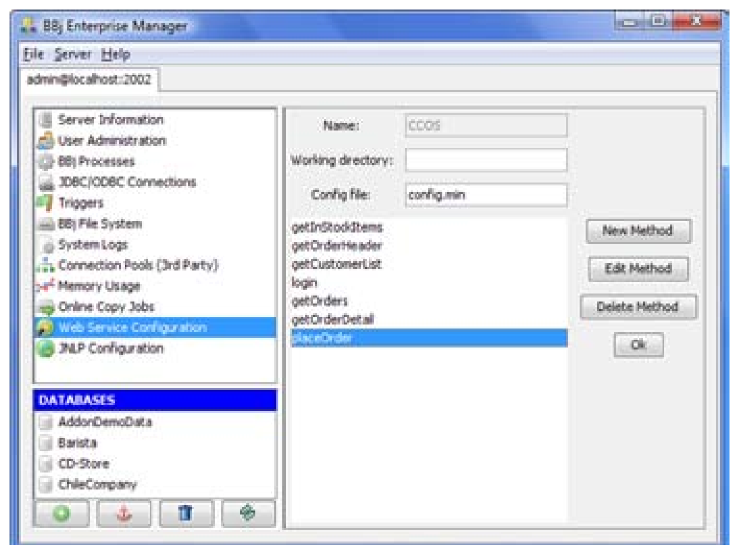

Figure 7. CCOS defined in EM

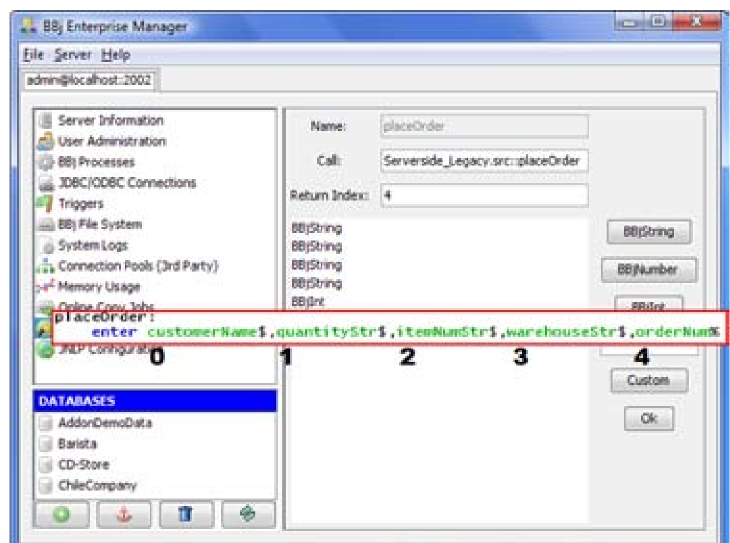

Figure 8. The correlation of a BBx/BBj subroutine to a Web Service method

- The **"Name"** field in this window is read-only after the method has been created.

- The **"Call"** field takes a subroutine that the Web Service will call in a similar fashion to the CALL verb in BBj. The format of the subroutine is **"`<BBj program name>::<label>`"**. The program name should be just the base name of the BBj/BBx program without a preceding path. The label name should correspond to a subroutine's label in that program file.

  As mentioned earlier, it is possible to configure a Web Service to use subroutines from not just one, but several program files. The only requirement is that those program files be in your working directory or PREFIX.

- The **"Return Index"** field is particularly important; subroutines are required to always return a value and one of the parameters must act as the return value.

  Due to these requirements, a subroutine that a Web Service uses must always have at least one parameter. The Web Service will return the parameter specified in the Return Index field as the return value of the method. Return Index is the zero-based position of the parameter in the parameter list. For example, placeOrder takes five parameters. The first parameter has an index of 0, and the last parameter has an index of 4. The method uses orderNum% as the return value. Therefore, our Return Index field is set to 4.

- The listbox at the bottom of the window shows the five parameters: customerName$, quantityStr$, itemNumStr$, warehouseStr$, and orderNum%. While customerName$, quantityStr$, itemNumStr$, and warehouseStr$ are all of type BBjString, orderNum% is a BBjInt.

To add parameters to the listbox, press the [BBjString], [BBjNumber], and [BBjInt] buttons. While BBj Web Services is a work in progress, BBj 9.0 supports these three types but future releases will support more types. In the meantime, we concatenated our results in the **`ServerSide_Legacy.src`** with a separator string rather than using arrays since BBj Web Services does not support arrays in BBj 9.0. The residual textbox and [Custom] button are windows into the future of BBj Web Services and are not currently supported.

The BBj Web Services server-side configuration is now complete.

Once all the Web Services methods correlate with each subroutine in the **`ServerSide_Legacy.src`** file, we went back to the first screen in Figure 6 and clicked [Publish Service]. Now that the Service is available for consumption, the WSDL, which is the document that tells a client what methods it can call on the Web Service, is available at http://localhost:8888/webservice/CCOS?wsdl. This WSDL is exactly what we needed to consume the Web Service in a Web Services client.

If you already familiar with Web Services, you may find it helpful to access the WSDL with a reference to its corresponding xsd file at http://localhost:8888/webservice/CCOS?WSDL=1 and the XSD file at http://localhost:8888/webservice/CCOS?xsd=1.

Consuming the Web Service

BASIS wrote the "Chile Company Web Service" demo client in BBj with some help from tools available in Java. However, the clients can be written in any language that supports Web Services, such as Microsoft's .NET platform (C# and Visual Basic), Perl, and Java. To write a Web Service client, generate client stubs, which are Java libraries for accessing a particular Web Service. The client programs can use these stubs to access the Web Service.

Generating client stubs is really easy using the tools available in Java 6. Locate Java's wsimport tool in the bin directory of your JDK 6 install. To generate client stubs for the "Chile Company Web Service" demo, we switched to a directory in which we wanted to generate a couple of Java packages with Java classes, and then entered the following at the command/shell prompt:

```
wsimport http://localhost:8888/webservice/
CCOS?WSDL
```

This command uses the WSDL document to create two Java packages, "ccos" and "bbjgenerated." BASIS generated two packages found in the **`<BBj installation>/lib/ChileCompanyWebService.jar`** JAR file with this command. The "ccos" package contains two classes known as the *service* and *port* classes. In the case of CCOS, the service class is called **`CCOSServiceService`** and the port class is called **`CCOS`**. The **`bbjgenerated`** directory contains the classes that correspond to the methods that BASIS configured in the EM. These are message classes the port class uses, but not directly by the client.

After generating the client stubs, add the Java classes to the BBjServices classpath. In our case, the **`BBjIndex.jar`** already includes the **`ChileCompanyWebService.jar`**. Other than making use of the service and port classes, the "Chile Company Web Service" demo client is a typical BBj program. Access the Web Service in the client code in these four easy steps:

1. **Reference the generated classes.** The USE statements allow the program to access generated classes in the jar located in the classpath.

```
declare example.CCOS port!
port!=service!.getCCOSServicePort()
```

2. **Instantiate the service class**. The service class is one of two classes in the package and has the name **`<Service Name>+"Service"`**.

```
declare example.CCOSServiceService service!
service!=new example.CCOSServiceService()
```

3. **Obtain the port from the service.** The port is the other class that is included in the ccos package and has the same name as the service. Rather than instantiating this class directly, call the **`get<service name>ServicePort()`** method on your service object. >>

4. **Call any of the methods defined in EM on the port object** once you obtain the port object. Since `placeOrder` was defined in **Figure 8** as taking four BBjStrings, you will need to pass four string values to the method. Configuration in EM defines this method as returning a BBjInteger, which the following statement assigns to `orderNum%`.

```
orderNum%=port!.placeOrder(customerName$,quantityStr$,itemNumStr$,warehouseStr$,orderNum%)
```

How it all Connects

You have come full circle seeing a BBj Web Services client in action and the server-side code that the service invokes, and how the client Web Services program can connect to the Web Service. The "Chile Company Web Service" demo showed that when the user clicks the [Place Order] button, the demo invoked port!.placeOrder(), sending a placeOrder message to the Web Service configured in EM. This resulted in a CALL to the subroutine ServerSideLegacy.src::placeOrder upon receiving the "placeOrder" message. Web Services then executed the subroutine and passed back orderNum% as the return value. Inside the demo client's code, orderNum% was assigned to that return value.

Summary

So now, when you are looking at the current weather update on your day planner application, you will know just how BBj provides a similar live data connection to your sales force, inventory crew, customers, etc. BBj makes creating a Web Service a breeze, saving days or even months of development time, otherwise needed to distribute your current functionality over a network. Your users do not even need a LAN or VPN connection to be connected to your company! ■