HostBridge and WebSphere: Integrating CICS with IBM's Application Server

---

A HostBridge® White Paper

7/23/2002

Toll-free: (866) 965-2427
Email: info@hostbridge.com

Copyright Notice

Revision date: 7/23/2002

Trademarks

Table of Contents

List of Figures and Screen Shots

Appendix

HostBridge and WebSphere: Integrating CICS with IBM's Application Server

Overview: WebSphere, CICS and HostBridge

IBM's WebSphere Application Server is a platform for the development and deployment of web applications composed of Servlets, enterprise Java beans (EJBs), and Java server pages (JSPs) along with supporting communication standards such as Simple Object Access Protocol (SOAP), Universal Description, Discovery and Integration (UDDI), and Web Services Description Language (WSDL). It is fully J2EE compliant, providing support for Java's open web standards. WebSphere runs on many different platforms and enables connectivity among many environments. The open standards allow businesses to rapidly deploy new solutions developed from the vast Java resources available without reengineering for a non-standard system.

WebSphere belongs to an emerging category of eBusiness applications based on open standards such as XML and Java. However, the existing category of mainframe CICS applications is certainly not dead. In fact, according to statistics from IBM and others, CICS has never been more successful:

- 30 years and $1 trillion (per IDC) invested in CICS applications
- 14,000+ CICS customers worldwide
- 20,000+ CICS/390 licenses worldwide
- CICS is used by 490+ of IBM's top 500 customers
- 30 million end users of CICS applications
- 150,000+ concurrent users/system
- 5,000 CICS software packages from 2,000 ISVs
- 950,000 programmers earn their living from CICS
- CICS handles >30 billion transactions/day valued at >$1 trillion/week

For companies with large investments in mainframe CICS applications, the ability to integrate these applications with IBM's WebSphere is imperative.

HostBridge is a patent-pending software product that XML-enables existing CICS applications. HostBridge does this without requiring modification to the existing applications, and without screen-scraping. As a result, HostBridge is an ideal tool for integrating CICS applications with WebSphere, allowing quick, robust and reliable integration of CICS applications into the WebSphere Application Server environment.

This White Paper presents a case study on how you can use HostBridge to integrate existing CICS applications with WebSphere.

Types of CICS Applications

CICS applications operate in various ways. As a result, the integration approach will depend on how the CICS transaction operates. The following diagram shows a high-level taxonomy of CICS transactions.

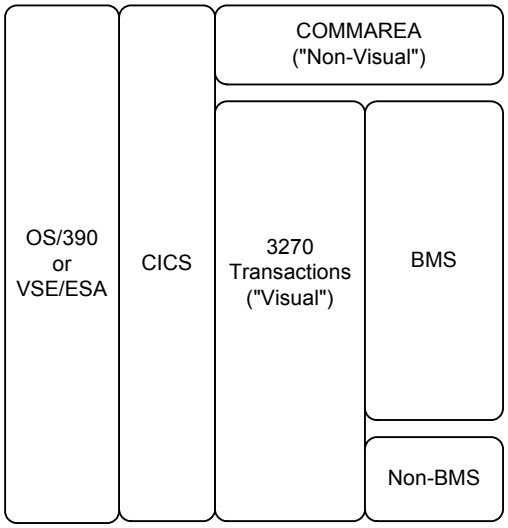


Figure 1. CICS Application Access Taxonomy

"Visual" vs. "Non-Visual" Transactions

CICS transactions fall into two broad categories: "visual" and "non-visual." A "visual" transaction is one that presents an interface to an end-user at a terminal. You could also refer to a "visual" transaction as a "terminal-oriented" transaction. In contrast, "non-visual" transactions do NOT interact with an end-user. Instead, another program invokes these transactions. (This type of transaction is also referred to as "COMMAREA transaction" because the input/output parameters are passed to/from the transaction using an area of storage referred to as the "communication area," or COMMAREA.)

The distinction between non-visual and visual transactions is important because integration possibilities exist for non-visual transactions that do not exist for visual transactions. As you might imagine, non-visual transactions are far easier to integrate with other programs than are visual transactions.

Understanding CICS "Visual" Transactions

In order to understand the integration possibilities for visual transactions, we need to further define this category.

CICS application developers have always had a number of choices in how to design their transaction to interact with an end-user at a terminal. The majority of applications use a component of CICS called Basic Mapping Support (BMS). The basic function of BMS is to provide data format and

device independence. BMS essentially handles the presentation logic of the transaction and relieves the application developer from having to encode and decode 3270 terminal data streams. The minority of applications that do not use BMS either include code to process 3270 data streams, or rely upon a non-IBM solution to handle presentation logic.

What is HostBridge?

HostBridge is a patent pending software product that XML-enables existing CICS transactions. HostBridge does this without requiring modification to your existing applications, and without screen-scraping. HostBridge supports all three types of CICS applications. As a result, HostBridge is the perfect tool for integrating CICS applications with WebSphere.

Integration Architecture

Given the flexibility of both HostBridge and WebSphere, there are numerous ways in which a WebSphere application can integrate with a CICS transaction using HostBridge.

HostBridge supports two basic interfaces: HTTP (POST or GET) and a LINK interface. Regardless of the interface used, a remote application can articulate its request to HostBridge using either a command string or an inbound XML document. When a command string is sent to HostBridge using an HTTP GET request, it is included as the query string on the URL. Alternatively, requests sent to/from HostBridge can be formatted using Simple Object Access Protocol (SOAP). The output generated by HostBridge is always an outbound XML document.

WebSphere supports numerous application-level components (e.g., JSP, Java servlet and EJB). A program using any or all of these components can invoke the services of a CICS application using HostBridge.

The sample WebSphere application described in this document will use HTTP GET requests for transport and HostBridge command strings to describe the service request (we will assume the use of the CICS HTTP listener, which is a standard component of CICS TS 1.3 or later). The sample WebSphere application will use Java Server Pages.

This basic architectural approach can be diagrammed as follows:

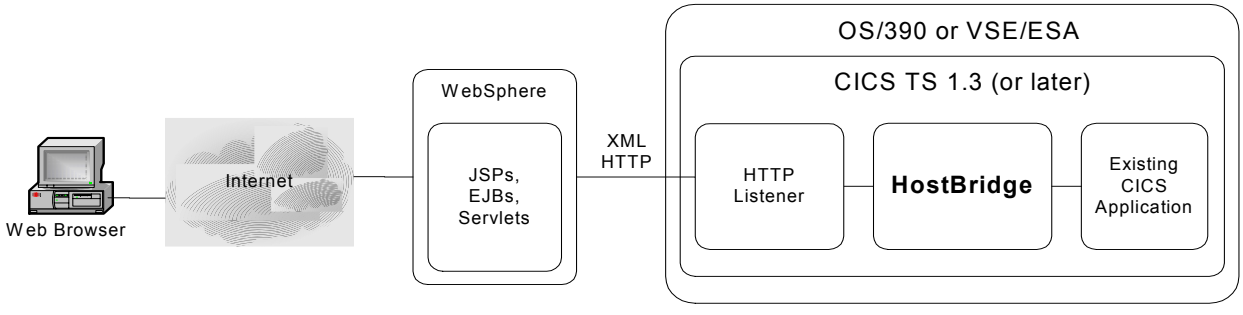


Figure 2. Sample Integration Architecture.

As illustrated above, the end user at a browser invokes the WebSphere application (or "Web Module"). When the WebSphere application requires the services of a CICS application, it sends an HTTP GET request to HostBridge. The request will be received by the HTTP Listener and authenticated. If valid, it will be passed to HostBridge for processing. HostBridge will examine the command string (received as the query string on the URL) to determine the transaction to be invoked. HostBridge returns the results of the transaction to the WebSphere application as an outbound XML document. The communication between the WebSphere application and CICS transactions continues through HostBridge until all required information is obtained, or updates are performed. The WebSphere application then replies to the end user accordingly.

Design Time Activities

The designer of a WebSphere application that uses HostBridge to access terminal-oriented transactions will need to know the following:

- The names of the CICS terminal-oriented transactions that contain the data elements you need in your WebSphere application;
- Whether the transactions use BMS or not (this will determine how the data elements are described by HostBridge in its XML output document);
- Which transaction, and/or which leg of a transaction composed of multiple interactions, outputs the data elements your WebSphere application requires.
- The navigation path, if any, required through a transaction to obtain the required data elements.

Discovering Field Names

When communicating with HostBridge, input and output fields are described using name/value pairs. And, when using HostBridge to interact with a CICS BMS transaction, the name of an input or output field is the same as the field's name in the BMS map. Therefore, we need to know the BMS field names that correspond to the data we wish to input and/or output.

BMS field names can be easily discovered by inspecting the BMS map files or by manually interacting with the BMS transactions. To test interactions with a CICS transaction using HostBridge, you can either use a web browser or the HostBridge Request Tool. (The request tool is a simple program that submits requests to HostBridge and displays the XML output.) When using a web browser, you simply enter the appropriate URL and query string required by HostBridge to invoke and/or interact with the CICS transaction. Examine the XML documents returned by HostBridge to discover the BMS field names. The HostBridge Request Tool allows you to do essentially the same thing, but in a more automated and convenient manner.

HostBridge Requests and Responses

To communicate with HostBridge, our WebSphere application will build command strings and send them as query strings in an HTTP GET request. A command string is composed of name/value pairs formatted as "name=value" with each pair separated by "&". Any field names proceeded by `HB_` are interpreted as HostBridge parameters; all others are presented to the CICS application as input data fields. For a complete description of command

strings, consult the HostBridge User Manual. Following are commonly used HostBridge parameters:

- `hb_tranid` – Specifies the name of the CICS transaction to be invoked
- `hb_token` – Specifies the session token generated by HostBridge and used to continue execution of a pseudo-conversational transaction
- `hb_aid` – Specifies the attention identifier (AID) key which is to be presented to the CICS transaction (as though it were a key pressed on a terminal)

Since the HostBridge command string will be sent using an HTTP request, it must be appropriately encoded. Our WebSphere application will do this using the `java.net.URLEncoder` class.

After sending a request, the WebSphere application receives the XML response from HostBridge (contained in the http response body). The XML response can be parsed with any of the XML parsers available; for example, Apache Xerces has both DOM and SAX parser implementations. HostBridge's XML response includes each field name/value pair as separate <field> elements. Each <field> element contains a child element <name> and a child element <value> that contains the name and value. Figure 3 shows an abbreviated XML response from the CICS transaction we will be using. Consult the User Manual for a detailed description of the output XML.

Sample Application

In this portion of the document we will use IBM WebSphere Studio Application Developer to create a JSP that will use HostBridge to interact with a sample CICS BMS application.

The sample application is a stock trading application that we refer to as the Trader application. Trader is included with CICS TS 1.3+ and is fully described in the HostBridge Interfaces Guide. The Interfaces Guide also contains detailed examples of how a client application (such as the JSP we will create) can interact with HostBridge. This document assumes that the reader is familiar with this information in the Interfaces Guide.

Like the majority of CICS applications in use today, the Trader application is a pseudo-conversational, terminal-oriented application that uses BMS (Basic Mapping Support) to handle the presentation logic. By automatically XML-enabling the Trader application, HostBridge will allow our JSP to easily communicate with it.

In this sample WebSphere application, we will use HostBridge to:

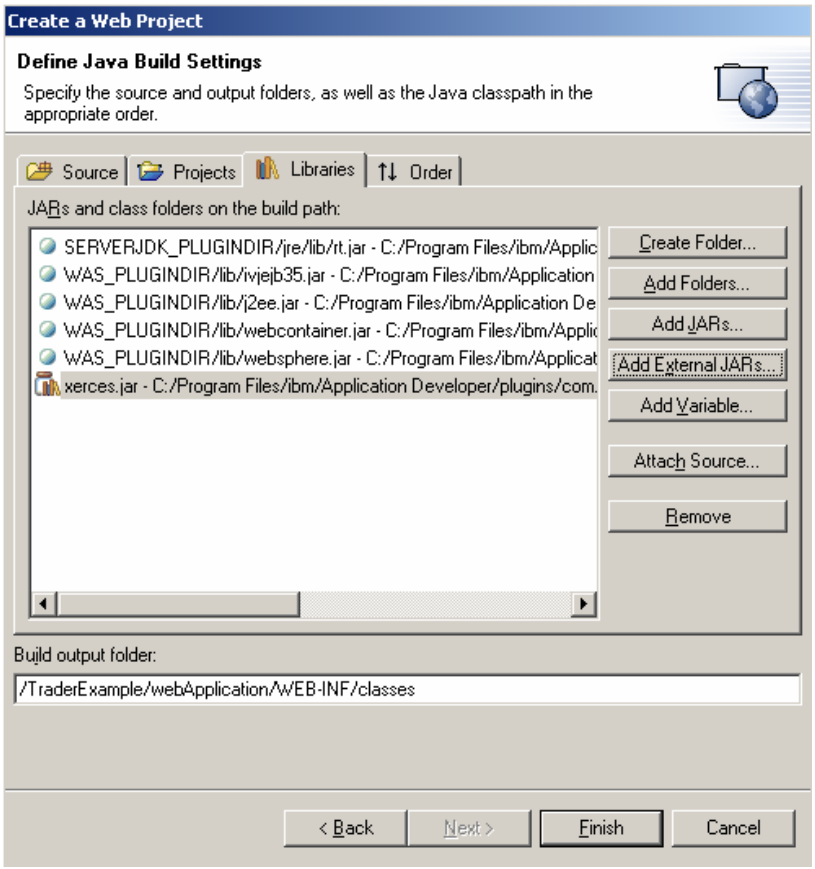
- get a list of company names for which stock quotes are available,
- create/display an HTML form listing the company names,
- allow the users to select a company, and
- display the stock quote for the selected company.

Developing the JSP

This section steps through development of the JSP with WebSphere Application Developer. Finished code for the example can be found in Appendix A.

The JSP will use the `HBRequest` class for communicating with HostBridge. `HBRequest` is a small Java class that simplifies communication with HostBridge; it exchanges http requests and responses with HostBridge and extracts the field name/value pairs from the returned XML documents. The `HBRequest` class communicates with HostBridge using a command string sent via an HTTP GET request. The command string is included in the URL as the query string. Additional information regarding `HBRequest` can be found in Appendix D, and the source code is available upon request.

After starting Application Developer, the first step is to create a new Web Project. Since the `HBRequest` class uses the Xerces XML parser, the project needs to import `xerces.jar`. This can be specified on the "Define Java Build Settings" screen of the Create a Web Project wizard. `xerces.jar` is located in `/plugins/com.ibm.etools.websphere.runtime/lib`



Screen 1. Importing the jar for the Xerces XML parser

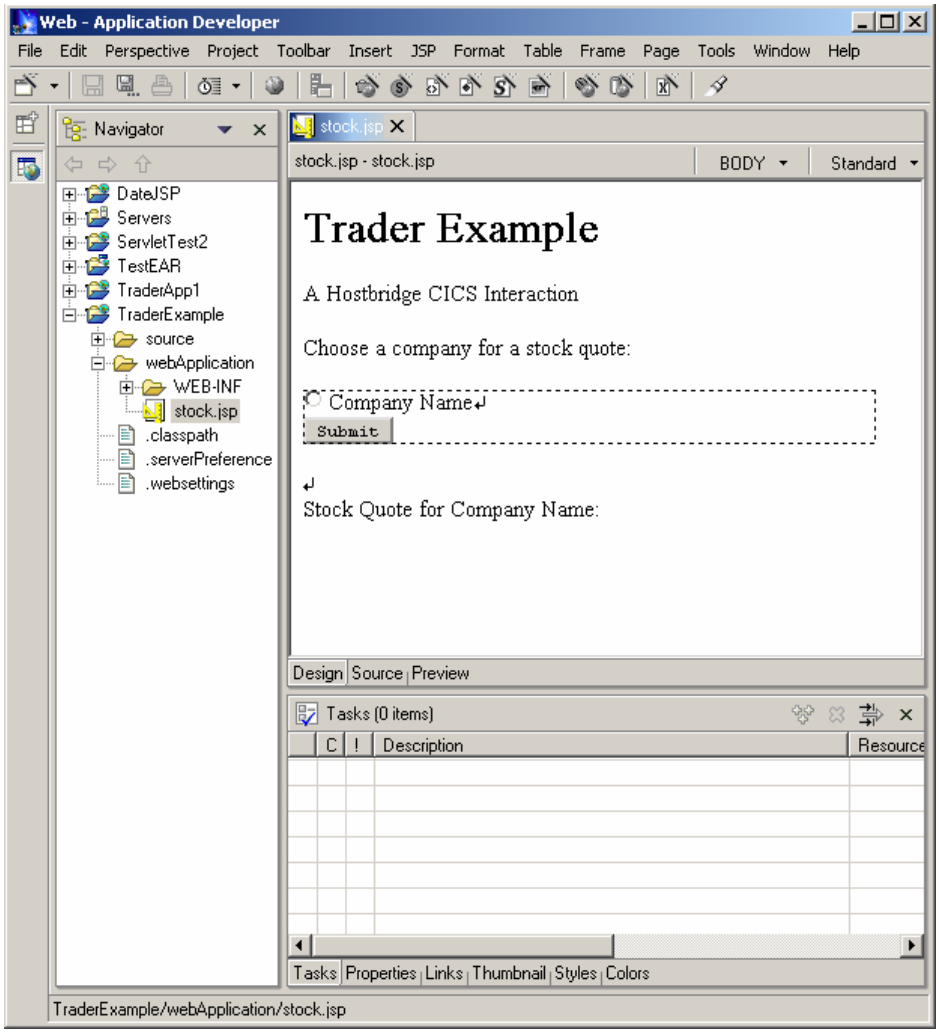
Part 1: Laying out the text

After generating the new project, Application Developer opens in Web View. From the file menu, click import to load in the source files for `HBRequest`. Also, create the file stock.jsp. After creating a new JSP, Application Developer opens to design mode which allows for easy layout of HTML components and Java scriptlets. Design mode is one of the three viewing

modes for a JSP; it allows for WYSIWYG editing of HTML elements and insertion of individual blocks of Java code (called scriptlets). The other two modes are source mode which shows the source code for the page and preview mode which gives a preview of the HTML elements of the page.

Insert a form for user input (it will show up on the screen as a dotted box). Insert a radio button inside the form. Finally, insert a submit button in the form.

Next, add descriptive text to the page so that it looks like the page illustrated in Screen 2.



Screen 2. The JSP with text and input form.

Part 2: First HostBridge Request

The first scriptlet in the page will contain the declaration and initialization of `HBRequest`; it will also send the first request to HostBridge.

Following is the code for the first request:

```
final String HOST = "demo.hostbridge.com";
final int PORT = 3029;
final String PROGRAM = "trad";
```

```
HBRequest hbRequest = new HBRequest();
hbRequest.setHost(HOST);
hbRequest.setPort(PORT);
hbRequest.setTranid(PROGRAM);
hbRequest.send();
```

Here is what the complete URL for the first request will look like:

```
http://hostaddress:port/         IP address and port assigned to HB
            hostbridge?          indicates this is a hostbridge request
            HB_TRANID=trad&      the CICS transaction name
            HB_AID=enter         AID key used with a transaction
```

The `hbRequest.send()` function formulates the http request, sends it, and receives the XML response from HostBridge. Following is an abbreviated version of the XML response:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!--HostBridge Copyright 2000, 2001 HostBridge Technology, U.S. Patent Pending-->
<hostbridge>
  <token>86d946b5</token>
  <fields>
    <field name="COMP1">
      <value>Casey_Import_Export</value>
    </field>
    <field name="COMP2">
      <value>Glass_and_Luget_Plc</value>
    </field>
    <field name="COMP3">
      <value>Headworth_Electrical</value>
    </field>
    <field name="COMP4">
      <value>IBM</value>
    </field>
    <field name="OPTION">
      <value></value>
    </field>
    <field name="MESS2">
      <value></value>
    </field>
  </fields>
</hostbridge>
```

Figure 3. XML response from HostBridge after invoking the Trader application

The Trader transaction returns the companies in its database in the fields named COMP1, COMP2, COMP3, and COMP4. Also included is the HostBridge session token, which HostBridge generated and uses to keep track of the transaction's state. After parsing the response, the `HBRequest` object contains a list of the field name/value pairs. `HBRequest` also maintains the session token.

We will write the script to load a variable number of companies so that if the Trader app is ever changed to include more companies the script will not need to be updated. The following `while` loop encompasses the radio button (but not the submit button) so that a radio button is generated for each company.

```
int i = 1;
while(hbRequest.getField("COMP" + i) != null){
     [Radio Button in Between]
 i++;
}
```

In design mode, double clicking the radio button will bring up the attributes dialog. Enter `i` (the name of the loop variable) under "value", and check "Specify by property" to set the value associated with the radio button to the current value of `i`. Next, insert the JSP expression *`hbRequest.getField("COMP" + i)`* after the tag for the radio button; this will cause the name of the company to be displayed after the radio button.

This completes the part of the JSP to display the company list. The JSP source produced in this section follows, with scriptlets in blue:

```
<P>Choose a company for a stock quote:</P>
<FORM>
<% int i = 1;
   while(hbRequest.getField("COMP" + i) != null){ %>
<INPUT name="CompanyList" type="radio" value="<%= i %>">
<%= hbRequest.getField("COMP" + i) %><BR>
<% i++;
} %>
<INPUT type="submit" name="submit" value="Submit"></FORM>
```

Figure 4. JSP source for getting company names.

Part 3: Getting Company Quote

To conditionally display the quote, surround the text used for the quote with the following scriptlets. This detects input to the page caused by clicking the submit button. (Note: "CompanyList" is the group name used for the radio buttons):

```
if(request.getParameter("CompanyList") != null){
     [Original Text is between scriptlets]
}
```

Inside the if statement, use the following expression to load the name of the company that was selected:

```
hbRequest.getField("COMP" + request.getParameter("CompanyList"))
```

Next the JSP gets to the main processing of the quote request. If a user were at the terminal, the user would select the company number, then select "1" to get a stock quote. In the script, set the `option` field to select the company, and then set `opt2` to "1" to receive the quote.

```
//First, select the company the query
hbRequest.setField("option", request.getParameter("CompanyList"));
hbRequest.send();
//Next, we select "1" to get the stock quote
hbRequest.setField("opt2", "1");
hbRequest.send();
```

After the last transaction `HBRequest` now has access to the share price. Use the expression *`hbRequest.getField("shrnow")`* to load the stock price. After the close brace of the previous `if` statement place the last scriptlet to close the Trader Application.

```
hbRequest.setAid("pf12");
hbRequest.send();
```

The JSP source produced in this section follows (Note: Appendix A contains the complete source to the JSP):

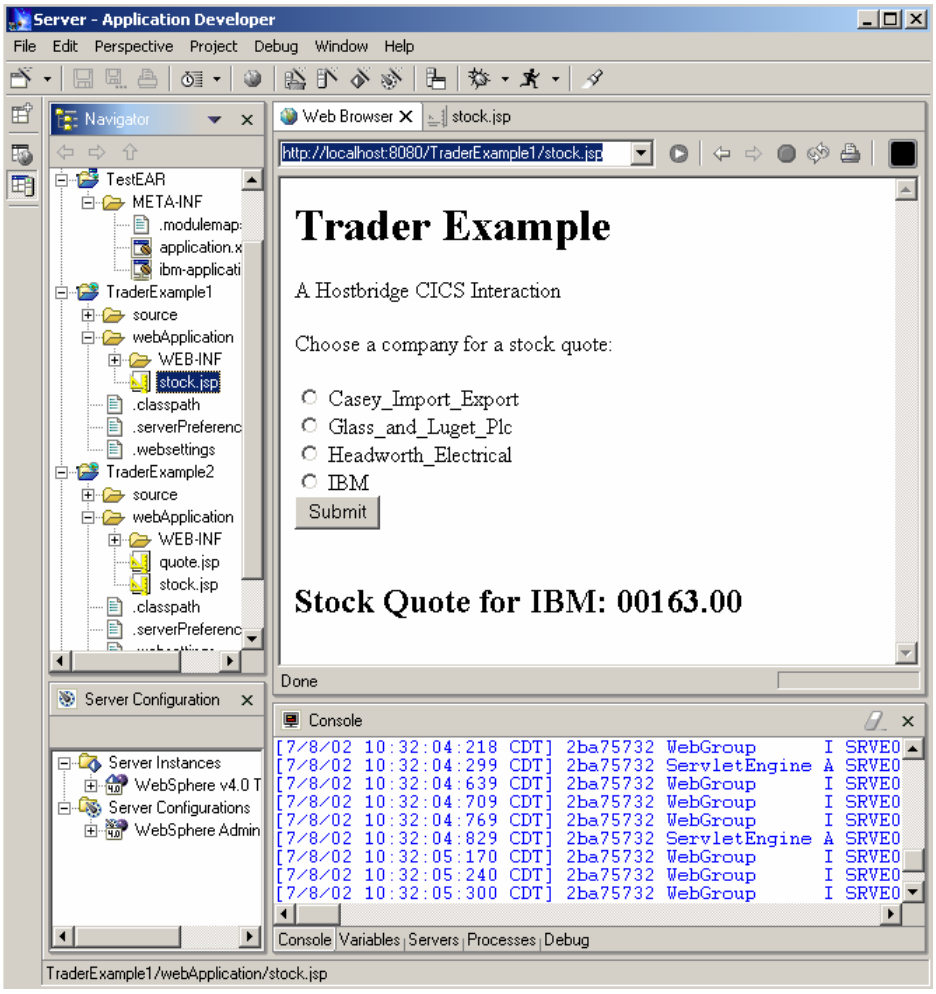
```
<% if(request.getParameter("CompanyList") != null){ %>
<BR>
<B>
<FONT size="+2">Stock Quote for

<%= hbRequest.getField("COMP" + request.getParameter("CompanyList")) %>:
<% //First, select the company the query
hbRequest.setField("option", request.getParameter("CompanyList"));
hbRequest.send();
//Next, we select "1" to get the stock quote
hbRequest.setField("opt2", "1");
hbRequest.send(); %>
<%= hbRequest.getField("shrnow") %>
</FONT>
</B>
<% }//Close out loop and end connection.
hbRequest.setAid("pf12");
hbRequest.send();
%>
```

Figure 5. JSP source for getting the stock quote.

Testing in Application Developer

Now it is time to test. Application Developer provides a test server function in order to test the pages along the way; this is brought up by right clicking on the JSP and selecting "Run on Server" or selecting it from the toolbar above. This process will most likely take a while, since it must first initialize the test server and then have the server dynamically compile the JSP into a servlet. The process of dynamically compiling occurs when the page is first loaded after a change to the JSP (subsequent loads will be much faster).

Here is the sample output:

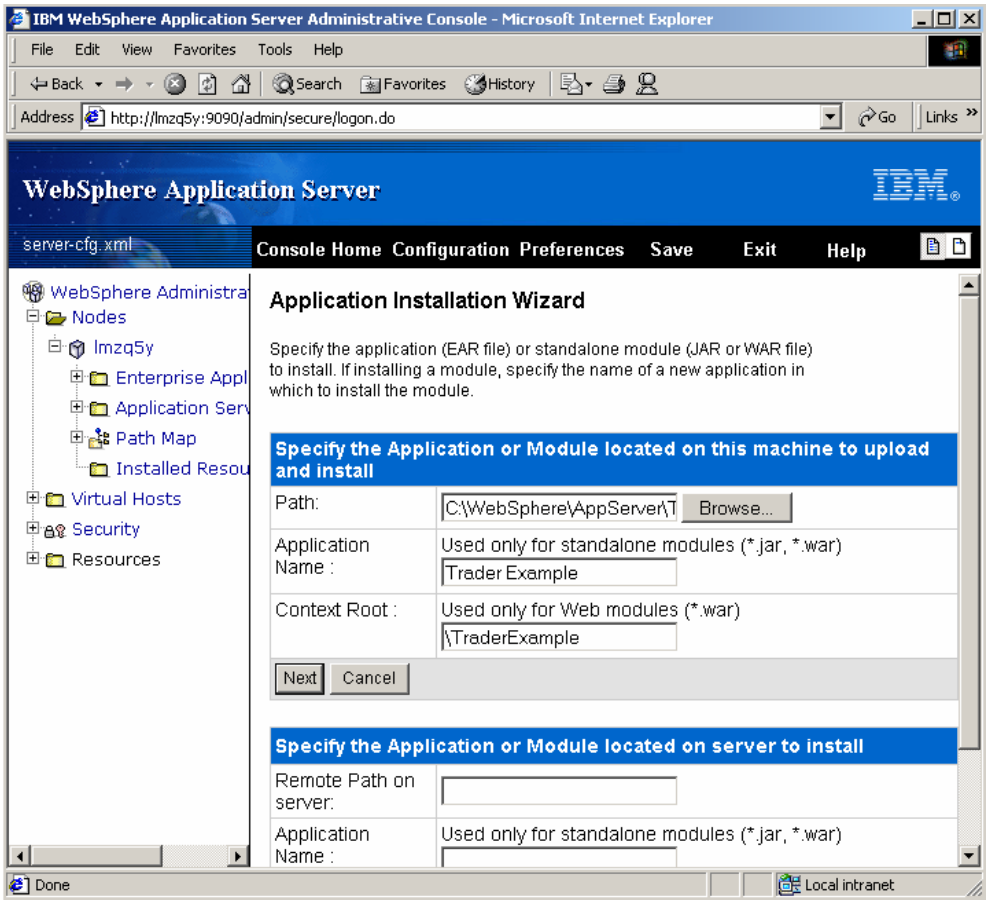


Screen 3. Testing application in WebSphere Application Developer

Publishing to Application Server

With the application successfully tested, it is time to publish the application. Publishing a Java Server Page involves first creating a web archive then an enterprise archive. Application Developer provides an easy method for creating a Web Archive (.WAR) file. Simply right click the whole project and select "Export WAR".

Make sure WebSphere Application Server is running, and run the Administrative Console (this can be done from the WebSphere Application Server - First Steps program). After opening the administrative console and logging in, select Nodes->[server to install to]->Enterprise Applications. The console brings up a list of applications installed. Choose Install to add the new application. Select the WAR file you created and enter the name and context root. The context root specifies the http path that the web module will occupy. WebSphere will generate an Enterprise Archive (.EAR) file for the web module.



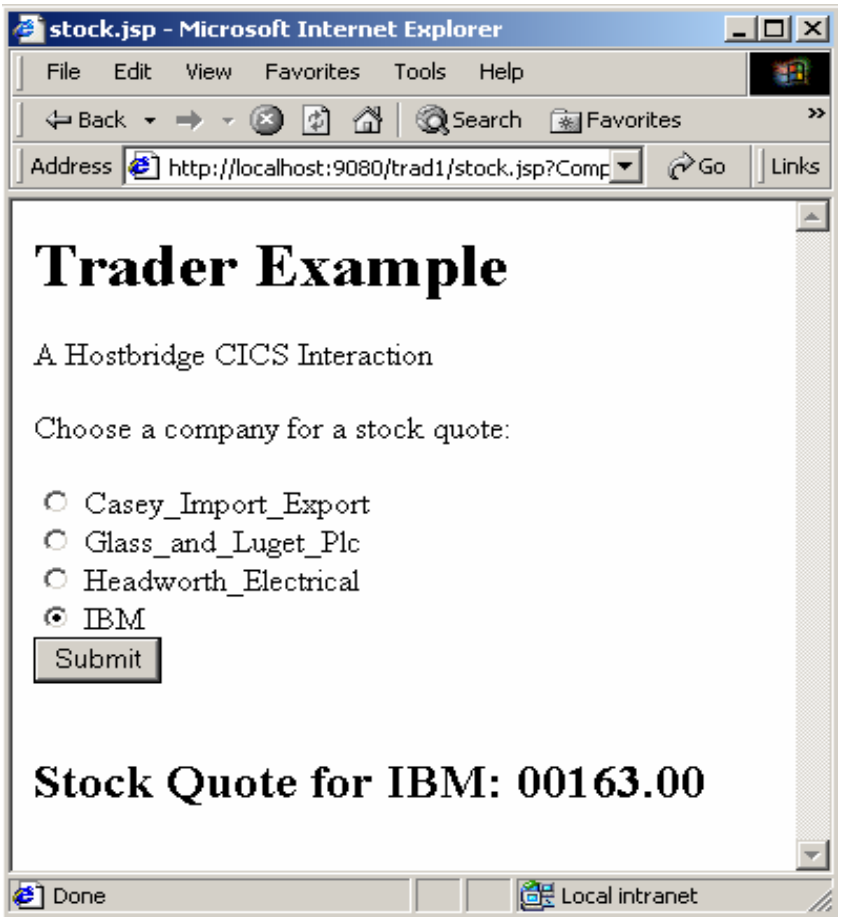
Screen 4. Installing the Web Archive file into the application server.

After filling out the information, click Next, then click Finish.

Now we must update the mappings for the Application Servers. Under the Application Servers Folder choose the application server installed to, and then click the link Web Server Plug-In Configuration, then hit generate. Finally, save the settings (using the menu bar above), and then exit. Before accessing the newly installed application, the Application Server must be restarted. (Done from the First Steps console.) To access the resulting application, simply open a browser and enter the appropriate URL; generically, the URL should be specified as: http://ApplicationServerName:port/ContextRoot/file.jsp

Assuming that WebSphere Application Server is installed on our workstation and is listening on port 9080, the appropriate URL to access our application would be: http://localhost:9080/TraderExample/stock.jsp

Screen 5 illustrates the output of our sample application when a quote for IBM is requested.



Screen 5. Successful communication between JSP and CICS application code.

Modifying the Sample Application

In the previous example, every time our JSP is loaded it creates a new `HBRequest` object. This was done for purposes of simplicity. However, it would be far more efficient (for the WebSphere application, HostBridge and CICS) if `HBRequest` persists over multiple page loads. Since `HBRequest` saves the HostBridge session token, persistence of the `HBRequest` object allows our JSP to interact with the CICS transaction within the context of a single CICS session. It will also provide much greater flexibility for our WebSphere application. In this section, we will modify our sample JSP to persist the `HBRequest` object over multiple page loads.

It will probably be easiest to do this in source view. First, create a second page called quote.jsp. In our first example, the stock quote was displayed as part of the stock.jsp page. In this example, we will use a separate page for displaying the stock quote. Next, move the code for loading the stock quote from stock.jsp to quote.jsp:

```
<% if(request.getParameter("CompanyList") != null){ %>
<BR>
<B><FONT size="+2">Stock Quote for
<%= hbRequest.getField("COMP" +
request.getParameter("CompanyList")) %>:
<%
```

```
//First, select the company the query
hbRequest.setField("option", request.getParameter("CompanyList"));
hbRequest.send();
//Next, we select "1" to get the stock quote
hbRequest.setField("opt2", "1");
hbRequest.send();
%>
<%= hbRequest.getField("shrnow") %>
</FONT></B></P>
<%
} //Close out loop and end connection.
hbRequest.setAid("pf12");
hbRequest.send();
%>
```

Persistant Sessions

To make `HBRequest` persist from page to page, use the implicitly defined `HttpSession` object `session` to store the `HBRequest` object as an attribute. In stock.jsp we store the request with *`session.setAttribute("HB_REQUEST", hbRequest)`* and retrieve it in quote.jsp with *`session.getAttribute("HB_REQUEST")`*. Finally, to link stock.jsp with quote.jsp insert an action into the form so that it looks like: *`<FORM ACTION = "quote.jsp">`*. Note that the only crucial information that `HBRequest` holds is the HostBridge session token, which allows HostBridge to process multiple requests within the same CICS session context. We could simply save the token from page to page, but storing the whole object is more convenient.

quote.jsp contains the code to load in the stock request. Above the copied code, retrieve the `HBRequest` object from the session as described in the previous paragraph. Checking the value retrieved from the session to make sure it is not null is a good idea, as a user clicking the "back" or "refresh" buttons on a browser can cause problems. Optionally, if the HB_REQUEST attribute is null, display a message for the user to not use browser buttons. After closing the transaction by sending the "pf12" command, remove the `HBRequest` object from the session by calling *`session.removeAtrribute("HB_REQUEST")`*.

Finally insert a link back to `stock.jsp` so the user can select a new company.

Finishing Touches

To provide the user with more information, add lines to get additional information from the stock quote screen, such as:

```
<BR>Current stocks held: <%= hbRequest.getField("held") %>
<BR>Total value of holdings: <%= hbRequest.getField("value") %>
```

This shows the number of stocks held and the total value of the stocks held. After modification, run the files on the test server in the application developer. If desired follow the same steps to publish to the application server as used for the first example.

Note: Appendix B contains the complete source to the JSPs

Summary

In this document we walked through the development and deployment of two simple WebSphere applications (JSPs) which use HostBridge to allow an end user to interact with a CICS transaction. HostBridge allowed the JSPs to communicate with the CICS transaction using simple HTTP requests and XML documents. This created a layer of abstraction between the presentation logic within the JSP and the data/business logic within the CICS transaction.

Using a persistent session between the WebSphere application and HostBridge allowed us to improve processing efficiency and application flexibility.

Appendix C contains a third WebSphere application that extends the session to be open- ended. This is a more realistic example, as many applications will need to be more interactive with the user. HostBridge allows the application to provide a higher level of interaction and responsiveness to the user.

Appendix A: Trader Example Code without Persistent Sessions

This is the code for the first example developed in the "Sample Application" section of the text. It loads the company names, and upon clicking the submit button produces a stock quote for the selected company. Note that the example uses demo.hostbridge.com, HostBridge's test server. This server is available for public demonstration, but should not be used for performance testing.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<META name="GENERATOR" content="IBM WebSphere Studio">
<TITLE>stock.jsp</TITLE>
</HEAD>
<BODY>
<P><B><FONT size="+3">Trader Example</FONT></B></P>

<%
//Setup code for the first HostBridge request
final String HOST = "demo.hostbridge.com";
final int PORT = 3029;
final String PROGRAM = "trad";
HBRequest hbRequest = new HBRequest();
hbRequest.setHost(HOST);
hbRequest.setPort(PORT);
hbRequest.setTranid(PROGRAM);
//The first send request invokes the trader app and returns the list of
// company names.
hbRequest.send();
%>

<P>A HostBridge CICS Interaction</P>
<P>Choose a company for a stock quote:</P>
<FORM>

<%
//Generate the list of companies by getting the COMP fields.
int i = 1;
while(hbRequest.getField("COMP" + i) != null){
%>
  <INPUT name="CompanyList" type="radio" value="<%= i %>">
  <%= hbRequest.getField("COMP" + i) %>
  <BR>
<%
  i++;
}
%>
<INPUT type="submit" name="submit" value="Submit"></FORM>
<P>
<%
if(request.getParameter("CompanyList") != null){
%>
  <BR>
  Stock Quote for <%= hbRequest.getField("COMP" +
  request.getParameter("CompanyList")) %>:
```

```
  <%
  //First, select the company to query
  hbRequest.setField("option", request.getParameter("CompanyList"));
  hbRequest.send();
  //Next, we select "1" to get the stock quote
  hbRequest.setField("opt2", "1");
  hbRequest.send();
  %> <%= hbRequest.getField("shrnow") %></P>
<%
}
//Close out loop and end connection.
hbRequest.setAid("pf12");
hbRequest.send();
%>
</BODY>
</HTML>
```

Figure 1. First stock.jsp handles company selection and displaying stock quote

Appendix B: Trader Example with Persistant Sessions

This is the code from "Modifying the Application" section. This example persists the session from one page boundary to the next. It splits the operations of the previous example so that the first page provides input list of companies, and the second page gives the stock quote.

stock.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<META name="GENERATOR" content="IBM WebSphere Studio">
<TITLE>stock.jsp</TITLE>
</HEAD>
<BODY>
<P><B><FONT size="+3">Trader Example</FONT></B></P>

<% final String HOST = "demo.hostbridge.com";
final int PORT = 3029;
final String PROGRAM = "trad";
HBRequest hbRequest = new HBRequest();
hbRequest.setHost(HOST);
hbRequest.setPort(PORT);
hbRequest.setTranid(PROGRAM);
hbRequest.send();
session.setAttribute("HB_REQUEST", hbRequest);
%>

<P>A Hostbridge CICS Interaction</P>
<P>Choose a company for a stock quote:</P>
<FORM ACTION = "quote.jsp">

<%
   int i = 1;
   while(hbRequest.getField("COMP" + i) != null){
%>
<INPUT name="CompanyList" type="radio" value="<%= i %>">
<%= hbRequest.getField("COMP" + i) %> <BR>
<%
   i++;
}
%>
<INPUT type="submit" name="submit" value="Submit">
</FORM>
</BODY>
</HTML>
```

Figure 2. Second version of stock.jsp only handles company selection.

quote.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<META name="GENERATOR" content="IBM WebSphere Studio">
<TITLE>quote.jsp</TITLE>
</HEAD>
<BODY>

<%  HBRequest hbRequest = null;
   if((hbRequest = (HBRequest)session.getAttribute("HB_REQUEST")) != null){
   if(request.getParameter("CompanyList") != null){
%>
<BR>
<B><FONT size="+2">Stock Quote for
<%= hbRequest.getField("COMP" + request.getParameter("CompanyList")) %>
</FONT></B>

<%
//First, select the company the query
hbRequest.setField("option", request.getParameter("CompanyList"));
hbRequest.send();
//Next, we select "1" to get the stock quote
hbRequest.setField("opt2", "1");
hbRequest.send();
%>

<BR>Price per share: <%= hbRequest.getField("shrnow") %>
<BR>Current stocks held: <%= hbRequest.getField("held") %>
<BR>Total value of holdings: <%= hbRequest.getField("value") %>

<%
  }//Close out loop and end connection.
  hbRequest.setAid("pf12");
  hbRequest.send();
}
%>
<BR>
<a href = "stock.jsp">Back to company selection</a>
</BODY>
</HTML>
```

Figure 3. quote.jsp displays the stock price and other information.

Appendix C: Trader Example with Extended Persistance

In the first example application, each time the page loaded we treated it as an autonomous session, loading any information necessary and ending the session afterwards. The second example maintained the session from one page to another.

This example extends the idea of persisting a transaction over multiple page loads and across several JSPs. It retrieves a company list and provides a quote as in the other examples, but now the second page provides the option of buying and selling.

Since the transaction can be open ended, the HBRequest object is kept in the HttpSession until a user clicks a log out link to exit. This design creates the issue of needing to keep track what state the transaction is in; otherwise, the JSP could be trying to select a company when it was actually on the stock quote page. In this example, the pages keep track of state based on if HBRequest is present in the HttpSession and if so, any query arguments provided to the page. A more scalable approach might involve an object abstracting the request that would maintain the current state, and could encapsulate the HBRequest as well.

There are four files: `stock.jsp`, `buysell.jsp`, `exit.jsp`, and `error.jsp` . See header comments within each file for a description of the file.

stock.jsp

`stock.jsp` should be the first file loaded, it initiates the first HostBridge request, and creates a company list, similar to the one created in the main part of the document. It has additional intro code that processes a transaction that occurs in `buysell.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<META name="GENERATOR" content="IBM WebSphere Studio">
<TITLE>stock.jsp</TITLE>
</HEAD>
<BODY>
<%--  This is the base page for the Trader Application interaction with
   HostBridge. It will be loaded initially and will list the company
   names, allowing the user to select one for a stock quote.  It will
   send the request to buysell.jsp for lookup.  buysell.jsp will send
   the user back here after a transaction request or the user hits
   "Select New Company".

   First, we check to see if a HostBridge interaction (encapsulated by
   a HBRequest object) has already been initiated. If we were referenced
   from buysell.jsp we check the parameters and process a transaction
   request as necessary.  Otherwise, we just reset the interaction.

   If no interaction has already been initiated or if we reset the
   previous one, we create a new HBRequest object, set the Tranid to "trad"
   and send the request.  HostBridge will receive the request and begin the
   trader application.
```

```
   After opening it, we are now on the initial screen (listing company names).
   We loop through, and load each company name into a radio button list
   with values equalling the company numbers.

   --%>

<%@ page
   errorPage = "error.jsp"
   %>
<%! HBRequest hbRequest; %>
<%  hbRequest = null;
if(session.getAttribute("HB_REQUEST") != null){

   // Check to see if we are reacting to an action from buysell.jsp
   if(request.getParameter("BackToStart") != null){
      hbRequest = (HBRequest)session.getAttribute("HB_REQUEST");
      hbRequest.setAid("pf3");
      hbRequest.send();
   }
   else if(request.getParameter("numShares") != null){

      //Note, we assume that if we return to this screen,
      //we are back on the Quote/Buy/Sell page.
      hbRequest = (HBRequest)session.getAttribute("HB_REQUEST");
      //We first process the request.
      hbRequest.setField("opt2", request.getParameter("buysell"));
      hbRequest.send();

      if( request.getParameter("buysell").equals("2") ){ //BUY
         hbRequest.setField("SHRBUY", request.getParameter("numShares"));
      }
      else{
         hbRequest.setField("SHRSELL", request.getParameter("numShares"));
      }
      hbRequest.send();
      // After transmitting the request, we are again on the QUOTE/BUY/SELL page.
      // Submit pf3 to return to the opening page.
      hbRequest.setAid("pf3");
      hbRequest.send();
      %>

      <%-- Give user feedback on outcome of transaction --%>
      <FONT color="#0000cc">Previous transaction successful.</FONT> <%
   }
   else{
       // If there is no action, begin a new transaction.
      hbRequest = null;
      session.removeAttribute("HB_REQUEST");
   }
}

session.setMaxInactiveInterval(30); //in seconds
if(hbRequest == null){
   final String HOST = "demo.hostbridge.com";
   final int PORT = 3029;
   final String PROGRAM = "trad";
   hbRequest = new HBRequest();
   session.setAttribute("HB_REQUEST", hbRequest);
   hbRequest.setHost(HOST);
   hbRequest.setPort(PORT);
   hbRequest.setTranid(PROGRAM);
   hbRequest.send();
} %>

<P><B><FONT size="+3">Trader Example</FONT></B></P>
<P>A Hostbridge CICS Interaction</P>
<P>Choose a company for a stock quote:</P>
```

```
<FORM ACTION = "buysell.jsp">

<%
   int i = 1;
   while(hbRequest.getField("COMP" + i) != null){
%>
       <INPUT name="CompanyList" type="radio" value="<%= i %>">
       <%= hbRequest.getField("COMP" + i) %><BR>
<%     i++;
   }
%>
<INPUT type="submit" name="submit" value="Submit">
</FORM>
<A HREF = "exit.jsp">Log Out</A>
</BODY>
</HTML>
```

Figure 4. stock.jsp displays a company list and carries out any transaction from buysell.jsp

buysell.jsp

buysell.jsp loads the share price and share count for the company selected in stock.jsp. See comments in the file for details

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<META name="GENERATOR" content="IBM WebSphere Studio">
<TITLE>buysell.jsp</TITLE>
</HEAD>
<BODY>
<%--
     This page displays the stock quote requested from the start page (stock.jsp).
     It takes the hbRequest from the current HttpSession and loads the stock quote
     by making two requests to HostBridge.  It first selects "option" which
     corresponds to the company number to be selected and sends a request.  Once
     we select a company the trader app presents the following options for field
     opt2:
     1. New Real-Time Quote
     2. Buy Shares
     3. Sell Shares

     To get the quote, we send a request to HostBridge selecting 1.  The share
     price and number of shares are stored in the fields "SHRNOW" and "HELD".

     Since choices 2 and 3 for opt2 corresponds to Buy and Sell respectively, the
     Buy and Sell radio buttons in this page have the values 2 and 3. This makes
     processing logic in stock.jsp simpler.

   --%>
<%@ page
   errorPage = "error.jsp"
   %>
<%! HBRequest hbRequest; %>
<% hbRequest = (HBRequest)session.getAttribute("HB_REQUEST");
// If the request object is null, or no CompanyList input, we only load the
// link to Log Out
if(hbRequest != null && request.getParameter("CompanyList") != null){
   session.setAttribute("COMPANY_NUMBER", request.getParameter("CompanyList"));
%>

Stock quote for <%= hbRequest.getField("COMP" +
session.getAttribute("COMPANY_NUMBER")) %>

<%  //Send request selecting company that user chose on first page.
```

```
   hbRequest.setField("option", request.getParameter("CompanyList"));
   hbRequest.send();
   //Send request for "Real Time Quote"
   hbRequest.setField("opt2", "1");
   hbRequest.send();
%>
   : $<%= hbRequest.getField("SHRNOW") %>
   <BR>
   You currently have <%= hbRequest.getField("HELD") %> shares.

<%  //Exit back one screen
   hbRequest.setAid("pf3");
   hbRequest.send();
%>
<FORM ACTION = "stock.jsp">Fill in number of shares to buy or sell, then hit
submit.<BR>
<INPUT type="radio" checked name="buysell" value="2"> Buy
<INPUT type="radio" name="buysell" value="3"> Sell
<INPUT size="10" type="text" maxlength="10" name="numShares" value="0"> Shares
<INPUT type="submit" name="Submit" value="Submit">
</FORM>
<BR>
OR
<BR>
<FORM action="stock.jsp">
<INPUT type="submit" name="BackToStart" value="Select New Company"></FORM>
<% } %>

<A HREF = "exit.jsp">Log Out</A>
</BODY>
</HTML>
```

Figure 5. buysell.jsp displays the stock quote and allows user to buy or sell shares

exit.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<META name="GENERATOR" content="IBM WebSphere Studio">
<TITLE>exit.jsp</TITLE>
</HEAD>
<BODY>
<%--
     This page exits by retrieving the hbRequest object, and sending a
     request with "pf12" as the aid (this simulates hitting the pf12 key
     which exits the application.  Afterwards, we remove the request
     object from the session.
--%>
<% HBRequest hbRequest = (HBRequest)session.getAttribute("HB_REQUEST");
   hbRequest.setAid("pf12");
   hbRequest.send();
   session.removeAttribute("HB_REQUEST");
   %>
We thank you for your business!
<P><A href="stock.jsp">Click here to return</A></P>
</BODY>
</HTML>
```

Figure 6. exit.jsp ends the session by sending “pf12” aid.

error.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML><HEAD>
```

```
<META name="GENERATOR" content="IBM WebSphere Studio">
<TITLE>error.jsp</TITLE>
</HEAD>
<BODY>
<%--
     Standard error page for the other three files.  In case there is a problem
     with the transaction, we tell the user not to use the back button, and give
     them a link to the front page to load a new session.  If the session is not
     null, we try to exit with the “pf12” Aid.
--%>
<%@ page
isErrorPage="true"
%>
An Error occurred processing your request.  Make sure you don't reload
or hit the back button.

<BR>
<P><A href="stock.jsp">Click here to return</A></P>
<%-- Only print exception in debugging mode.
<%= exception %>
<%  exception.printStackTrace( new PrintWriter(out) );%> --%>
<% HBRequest hbRequest = (HBRequest)session.getAttribute("HB_REQUEST");
   if( hbRequest != null){
      session.removeAttribute("HB_REQUEST");
      hbRequest.setAid("pf12");
      hbRequest.send();
   }
   %>
</BODY>
</HTML>
```

Figure 7. error.jsp catches exceptions and ends the session of HBRequest if present

Appendix D: HBRequest Description

This sections shows select functions from HBRequest.java and provides a description of each. HBRequest illustrates programmatically interacting with HostBridge. Functions have been edited for brevity (typically only logging is removed). Also, before the first use of a non-String instance variable, a comment indicates its declared type.

send()

`send()` is called after setting the fields to invoke a transaction with HostBridge. In `doHttpRequest()` It sends the request and validates the response header . Next, in `readXml()`, the response is read from HostBridge. Finally, in `parseXml()` a DOM document is created from the received XML.

```
   public void send() throws HBException
   {
      doHttpRequest();
      readXml();
      parseXml();

      // Save off the token
      m_Token = getToken();
      clearParameters( true );
   }
```

Figure 8. HBRequest.send() sends a request, and receives and parses the response.

doHttpRequest()

`doHttpRequest()` first builds the url and opens the connection with the server. After setting the username and password by calling `setRequestProperty()`, it calls connect and then checks the response code. If the response isn't "HTTP_OK" it throws an exception. The response is read in `readXml()`.

```
private void doHttpRequest() throws HBException
{
   URL Url = null;
   String UrlText = buildUrl();
   try
   {
      Url = new URL( UrlText );
   }
   catch ( java.net.MalformedURLException ex )
   {
      throw new HBException( "URL is malformed." );
   }
   // HttpURLConnection m_Request
   m_Request = null;
   try
   {
      m_Request = (HttpURLConnection) Url.openConnection();

      // GET, POST, etc
      m_Request.setRequestMethod( m_Method );
```

```
      // Security credentials
      if ( m_User != null )
      {
         String userPass = m_User + ":" + m_Password;
         String userPass64 = new sun.misc.BASE64Encoder().encode(
userPass.getBytes());
         // Add in Basic authentication header
         m_Request.setRequestProperty( "Authorization",
                                 "Basic " + userPass64 );
      }
      // Send request onto network
      m_Request.connect();
   }
   catch ( java.io.IOException ex )
   {
      throw new HBException( "Could not connect to URL " +
                         Url.toString() );
   }
   try
   {
      // Verify that the HTTP response is OK
      if ( m_Request.getResponseCode() != HttpURLConnection.HTTP_OK )
      {
         throw new HBException( "Web server responded with " +
                            m_Request.getResponseCode() + " " +
                            m_Request.getResponseMessage() );
      }
      // Make sure we got back an XML document
      String ContentType = m_Request.getHeaderField( "Content-type" );
      if ( ( ContentType == null ) ||
          ( ContentType.compareTo( "text/xml" ) != 0 ) )
      {
         if ( ContentType != null )
         {
            throw new HBException( "Document is not XML.  Content type is "
+ ContentType );
         }
         else
         {
            throw new HBException("No Content-type HTTP header exists.");
         }
      }
   }
   catch ( java.io.IOException ex )
   {
      throw new HBException("IO Exception while reading HTTP response.");
   }
}
```

Figure 9. HBRequest.doHttpRequest() sends the request and checks reponse header.

buildURL()

`buildURL()` builds the request URL from the http address and the command string. The http address is build from the protocol, host, port, and resource properties. The command string is built from HostBridge parameters and any optional fields set with the `setField()` function in `HBRequest`

The http address usually looks like:
`http://demo.hostbridge.com:3029/hostbridge`

The HostBridge command string is build out of the following components, separated by ampersands. Since `HBRequest` uses http requests, the command string is appended as a query string to the http address. The command string has the following components:

- hb_tranid=<Tranid> this lists the program name being executed.
- hb_aid=<aid> an aid for the request, defaults to enter.
- hb_token= <token> this is the HostBridge session token that tracks a transaction.
- <Name>=<Value> any name/value pairs passed into the setField function

See the HostBridge User Manual for a complete description of command string arguments. In the connection it uses the specified connection method (such as GET or POST.) Here is an example command string for a HostBridge request URL:

`.../hostbridge?hb_tranid=trad&hb_token=ca445fb5&hb_aid=enter&option=1`

This would correspond to the request after receiving the first screen and selecting Casey_Import_Export (option 1). (See Appendix A for description of trader application).

```
private String buildUrl()
{
   //StringBuffer m_TempUrl
   if ( m_TempUrl == null )
   {
       m_TempUrl = new StringBuffer();
   }
   else
   {
      m_TempUrl.delete( 0, m_TempUrl.length() );
   }
   m_TempUrl.append( m_Protocol );
   m_TempUrl.append( "://" );
   m_TempUrl.append( m_Host );
   m_TempUrl.append( ":" );
   m_TempUrl.append( m_Port );
   m_TempUrl.append( m_Resource );
   if ( m_QueryString == null )
   {
      m_QueryString = new StringBuffer();
   }
   else
   {
      m_QueryString.delete( 0, m_QueryString.length() );
   }
   //
   // Transaction id
   //
   if ( m_Tranid != null )
   {
      addPair( "hb_tranid", m_Tranid );
   }
   //
```

```
   // Attention identifier key
   //
   if ( m_Aid != null )
   {
      addPair( "hb_aid", m_Aid );
   }
   //
   // HostBridge token to indicate whether this is a
   // new request or continuing session
   //
   if ( m_Token != null )
   {
      addPair( "hb_token", m_Token );
   }

   // Add in any updated fields
   if ( m_Fields != null )
   {
      for ( int Index = 0; Index < m_Fields.size(); ++Index )
      {
         HBPair Pair = (HBPair) m_Fields.get( Index );
         addPair( Pair.first, Pair.second );
      }
   }
   if ( m_QueryString.length() > 0 )
   {
      m_TempUrl.append( "?" );
      m_TempUrl.append( m_QueryString );
   }
   return m_TempUrl.toString();
}
```

Figure 10. HBRequest.buildURL() builds the HostBridge command string & URL

readXml()

`readXml()` simply reads the response document into a buffer, to be parsed in `parseXml()` the contents have already been verified in `doHttpRequest()`, so little error checking is necessary here.

```
private void readXml() throws HBException
{
   // We should have an XML document at this point
   // Read in the entire document
   // ByteArrayOutputStream m_XmlDocument
   if ( m_XmlDocument == null )
   {
      m_XmlDocument = new ByteArrayOutputStream();
   }
   else
   {
      m_XmlDocument.reset();
   }
   try
   {
      //HttpURLConnection m_Request
      InputStream Input = m_Request.getInputStream();

      final int BUFFER_SIZE = 2048;
      //Byte[] m_XmlBuffer
```

```
      if ( m_XmlBuffer == null )
      {
         m_XmlBuffer = new byte[ BUFFER_SIZE ];
      }
      int Count = Input.read( m_XmlBuffer, 0, BUFFER_SIZE );

      while ( Count != -1 )
      {
         int i = 1;
         m_XmlDocument.write( m_XmlBuffer, 0, Count );
         Count = Input.read( m_XmlBuffer, 0, BUFFER_SIZE );
      }
   }
   catch ( java.io.IOException ex )
   {
      throw new HBException( "IO Exception reading XML document." );
   }
}
```

Figure 11. HBRequest.readXml() reads the XML document from the HttpConnection

parseXml()

`parseXml()` uses a default document builder to parse the document retrieved by `readXml()` after this returns, calls can be made to `getField()` to retrieve field name/value pairs from the XML document.

```
private void parseXml() throws HBException
{
   // We've read the document
   // parse it and find a specified node
   // Document m_XmlDom
   m_XmlDom = null;
   // DocumentBuilder m_XmlDocBuilder
   m_XmlDocBuilder = null;
   try
   {
      // DocumentBuilderFactory m_XmlDocumentBuilderFactory
      if ( m_XmlDocBuilderFactory == null )
      {
         // Step 1: create a DocumentBuilderFactory
         m_XmlDocBuilderFactory = DocumentBuilderFactory.newInstance();
      }

      // Step 2: create a DocumentBuilder
      m_XmlDocBuilder = m_XmlDocBuilderFactory.newDocumentBuilder();
   }
   catch ( javax.xml.parsers.ParserConfigurationException ex )
   {
      throw new HBException( "XML Parser configuration error" );
   }

   // Step 3: parse the input file to get a Document object
   StringReader stringReader = new StringReader(m_XmlDocument.toString());
   InputSource inputSource = new InputSource( stringReader );

   try
   {
      m_XmlDom = m_XmlDocBuilder.parse( inputSource );
```

```
   }
   catch ( org.xml.sax.SAXException ex )
   {
      throw new HBException( "SAX Exception occurred while parsing XML
document: " + ex.getMessage() );
   }
   catch ( java.io.IOException ex )
   {
      throw new HBException( "IO Exception occurred while parsing XML
document." );
   }
}
```

Figure 12. HBRequest.parseXml() creates a Document object from the response

getField(String Name)

`getField(String Name)` this returns the value of any field returned in the XML document. HostBridge returns field name/value pairs as defined in the BMS map. It loops through the field elements in the document, looking for the one with a particular name. Note that in DOM, the text inside an element is a child TextNode of the element. To extract "HostBridge" out of <name>HostBridge</name>, assuming we had the name element, we'd call *`nameElement.getFirstChild().getNodeValue()`*.

```
public String getField( String FieldName )
{
   String Return = null;
   NodeList List = m_XmlDom.getElementsByTagName( "field" );
   if ( List.getLength() > 0 )
   {
      boolean Done = false;
      for ( int Index = 0; !Done && Index < List.getLength(); ++Index )
      {
         //Item holds an individual field element the name and value of
         //the field are contained as child elements
         Node Item = List.item( Index );
         String Name = null;
         String Value = null;
         NodeList Children = Item.getChildNodes();
         boolean haveName = false;
         boolean haveValue = false;
         for ( int Child = 0; Child < Children.getLength(); ++Child )
         {
            Node Current = Children.item( Child );
            if ( Current.getNodeName().equals("name"))
            {
               //Get the text inside the element
               Name = Current.getFirstChild().getNodeValue();
               haveName = true;
            }

            if ( Current.getNodeName().equals("value"))
            {
               if ( Current.hasChildNodes() )
               {
                  Value = Current.getFirstChild().getNodeValue();
               }
               else
               {
```

```
                  Value = "";
               }
               haveValue = true;
            }

            // Don't want to make any order assumptions
            // so we use this scheme
            if ( haveName && haveValue )
            {
               if ( Name.compareToIgnoreCase( FieldName ) == 0 )
               {
                  Return = Value;
                  Done = true;
                  continue;
               }
            }
         }
      }
   }
   return Return;
}
```

Figure 13. HBRequest.getField(String Name) returns a value from the XML document