Using JMS and WebSphere Application Server

To Interact with CICS over the MQ/CICS Bridge

Ron Lotter
lotter@us.ibm.com

IBM Software Services for WebSphere
WebSphere Enablement Team
Raleigh, NC

November, 2005

 Version 12/8/2005
"http://www.ibm.com/support/Techdocs"http://www.ibm.com/support/Techdocs

Introduction

This paper, and the associated code samples present an approach to invoking CICS transactions that are DPL-enabled (use COMMAREAS) using JMS running in a WebSphere Application Server. There may be other papers that have discussed this functionality, yet the approach shown here uses the RAD V6 tooling to significantly simplify the build and test phases.

Conclusions

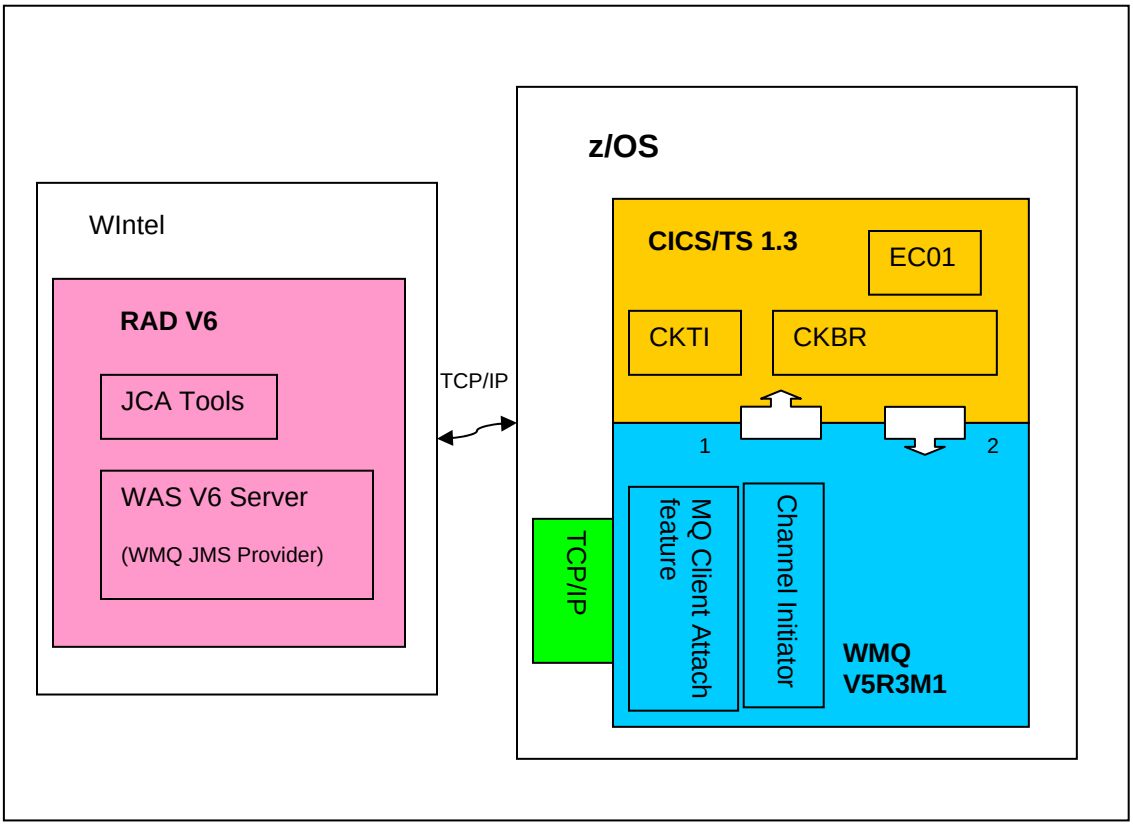
After implementing the functionality and testing an end-to-end scenario, the following observations and conclusions were drawn:

- Setting up the MQ/CICS Bridge is well documented and fairly straightforward. It helps to understand that an application will put a message on the CICS Bridge Queue, at which time MQ will fire a trigger task by writing a message on its INITQ. The trigger task makes sure the needed monitor is running in CICS which reads the message off the CICS Bridge Queue, invokes the specified program with its input COMMAREA, obtains the output COMMAREA and writes it back to the ReplyTo Queue specified in the request
- In order to provide maximum flexibility when interacting with CICS, a client application should include the MQCIH (CICS Header Structure) as part of the message request.
- Even if the client application does not send an MQCIH with the request, it must be enabled to receive a message which does include an MQCIH in the response, as this is the mechanism CICS uses to inform the client of certain error conditions.
- RAD V6 can be used to rapidly develop and test the JMS client application.
- The J2EE Connector Tools Feature in RAD V6 is useful to generate java code from COBOL code, used to mask the complexity of constructing a COMMAREA.
- The sample code runs in WAS V6 and uses a WebSphere MQ (external) JMS Provider. It should run unchanged in a V5 or V5.1 environment. (Additionally, a sample application is included that uses the MQ APIs directly. The hard-coded connection properties in this application must be updated before it can be used successfully.)
- The sample code can be used to test the correct configuration of WAS V6 (or V5, V5.1), WMQ V5.3.1, and CICS TS 1.3 (or higher).
- The sample code could be used to build a framework for this type of message processing with CICS that would insulate client applications from knowing these details, or may potentially be a part of a MediationHandler deployed to the SI Bus in WAS V6.

Configuration Overview

The development and runtime environments are illustrated and discussed here.

Development Configuration



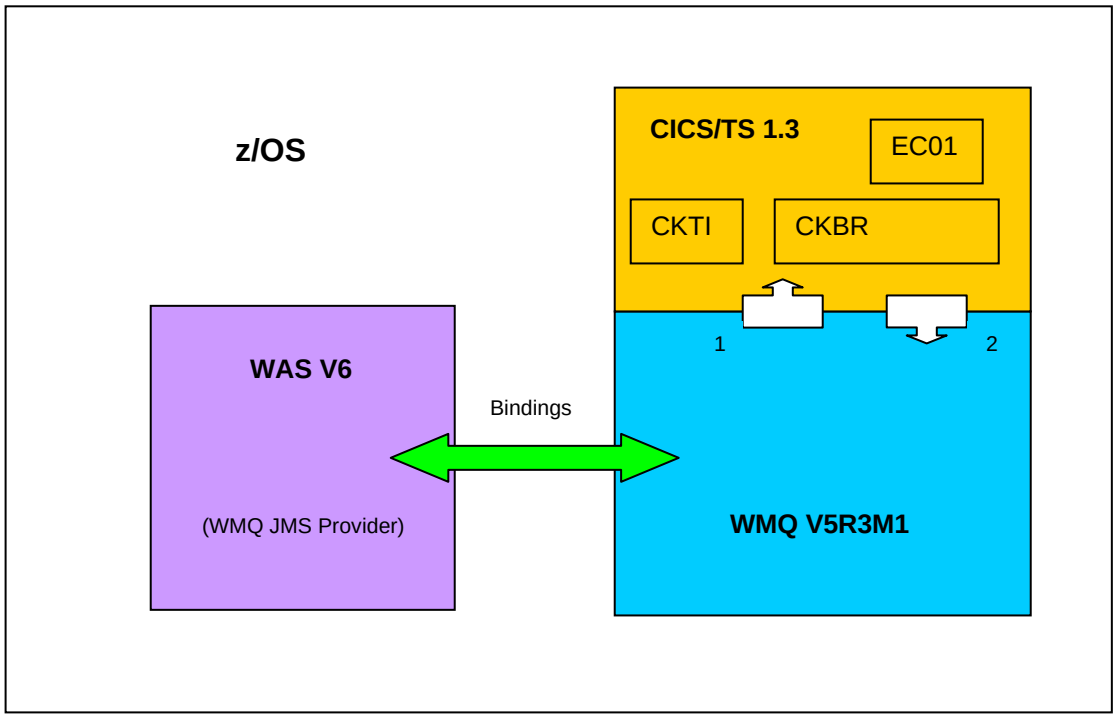
The development environment is RAD V6 with the J2EE Connector Tools Feature (download using the Rational SDP Product Updater). On z/OS the MQ Client Attachment Feature is installed, and a TCP/IP Listener is started so that we can connect to WMQ. There are other connectivity options. An instance of WMQ could be installed and configured on the development machine, which would enable the two QMgrs to interoperate.

In z/OS, the WMQ CICS Adapter (CKTI) and CICS Bridge (CKBR) are running in CICS. There is a SYSTEM.CICS.BRIDGE.QUEUE(1) defined to WMQ, where messages for CICS are placed, and a CICSB(2) queue where responses are obtained.

The DPL-enabled program that we will use to test with is named EC01. It is the ECIDateTime sample program that ships with the CICS Transaction Gateway. If it is unavailable, any simple program will do. It should use an 18 byte COMMAREA, expect it to be uninitialized on input, and it should return a simple text string. ECIDateTime is good since it returns the date and time, so you know immediately if you've gotten a good response.

Runtime Configuration

The runtime environment used to deploy the solution included the same WMQ and CICS subsystems used in development (no configuration changes were made). WAS V6 for z/OS was used as that would be a common platform choice, given the architecture. However, no functions specific to V6 were used, so a V5 or V5.1 AppServer would work. The supplied ear file would have to be repackaged as a J2EE 1.3 application, of course. No changes were made to the ear file to deploy on z/OS. The JMS resources were defined indentically to those in the RAD Test Server, except we used Bindings for the transport.



Design and Code Overview

When writing JMS applications, the first important concept to understand is that there are several kinds of messages that can be sent. The two message types that we will deal with are:

- JMSTextMessage – this is the simplest message to deal with and most JMS samples will typically start with this message. When sending a cross-platform TextMessage, translation is done automatically by the QMgr.
- JMSBytesMessage – this message provides much greater flexibility in terms of its content, and will generally be the more useful. Messages destined for CICS will likely be of this type, as integer, decimal and floating point fields are often present in addition to text. When sending cross-platform BytesMessages, the client application is responsible for the code page translation as well as using the proper field encodings for the target, in our case, z/OS.

A second important concept is that of a Managed Connection. A Managed Connection is obtained by asking the WebSphere runtime for it using a JNDI lookup. This is how all of the code, with one exception, obtains a Handle to the QMgr. The exception is found in the MQWrite class that is included in the package. This class is included mostly for reference. Since this class creates its own connections to WMQ, and since those connection properties have been hard-coded, to use it, you will have to modify the code. That, of course, is one of the great features of a Managed Connection; its properties are external from the code. Furthermore, Managed Connections are recommended because the runtime provides Connection Pooling, and Security Services.

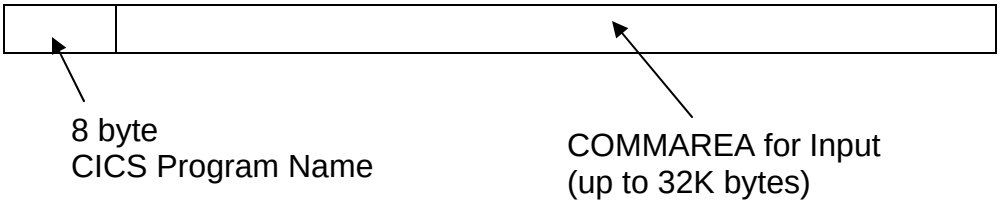
With that background, we will look in more detail at the solution.

What does the Developer Need to Know to use the Bridge?

It is helpful to have a basic understanding of sending and receiving messages using JMS. We will assume the basic concepts and functions of the JMS classes are familiar to the reader.

First Iteration

On a first iteration of this function, you simply need to understand that the message needs to have a minimum format such as the following:



Our sample uses a simple 18 byte COMMAREA of Strings, so we can use a JMS TextMessage. The following 3 lines of code set up the message:

```
TextMessage tmsg = sess.createTextMessage();
String smsg = new String("EC01                ");
tmsg.setText(smsg);
```

There are two properties of the message that must be set. The first tells the CICS Bridge to start a new transaction to run the program. The second tells the Bridge where to put the response message. These 2 lines of code do that:

```
tmsg.setJMSCorrelationID(new String(MQC.MQCI_NEW_SESSION));
tmsg.setJMSReplyTo(incomingQ);
```

It really is that simple. After coding the method to read a message off the queue, we have completed the first iteration and we can successfully test this function.

Second Iteration

Once we have the TextMessage working, it is natural to try using a BytesMessage. Why? Well, if the COMMAREA has any complexity to its format (and most of them do) treating it all as text will cause errors, during code page translation. So, when we try the BytesMessage and it fails, we realize there are a few more things to consider. Now, when we setup the COMMAREA, we need to translate it to the code page of the target QMgr:

```
BytesMessage bmsg = sess.createBytesMessage();
String smsg = new String("EC01                ");
byte[] bytes = smsg.getBytes("Cp500");
bmsg.writeBytes(bytes);
```

We set the 2 properties as we did for the TextMessage:

```
bmsg.setJMSCorrelationID(new String(MQC.MQCI_NEW_SESSION));
bmsg.setJMSReplyTo(incomingQ);
```

There are 3 additional properties as well,

```
bmsg.setStringProperty("JMS_IBM_Character_Set", "500" );
bmsg.setStringProperty("JMS_IBM_Format", MQC.MQFMT_NONE);
bmsg.setIntProperty("JMS_IBM_Encoding", 785);
```

With these changes, and updates to our receiver method to handle a BytesMessage returned from the Bridge, we can successfully interact with CICS.

During this iteration, however, two important questions arose:

1. When our first attempt to send the BytesMessage failed, we got a message returned that we didn't quite understand. We discovered that it contained an MQCIH header.
2. If one has complex COMMAREAS, doesn't the java code to build and format them get fairly cumbersome and error prone?

Seems we need a Third Iteration.

Why Include an MQCIH?

It turns out, as we discovered above, we need to be able to understand an MQCIH header in our receiver code, because whether we send one or not, with our request, the Bridge may send us one back with the response.

Since we must understand an MQCIH header, it may be interesting to add one to the input message as well. Doing so allows us much greater flexibility when interacting with CICS. For example, the MQCIH is required if you want to modify the name of the Transaction ID the program will run under in CICS.

What RAD V6 Features Can Help me Build the Messages?

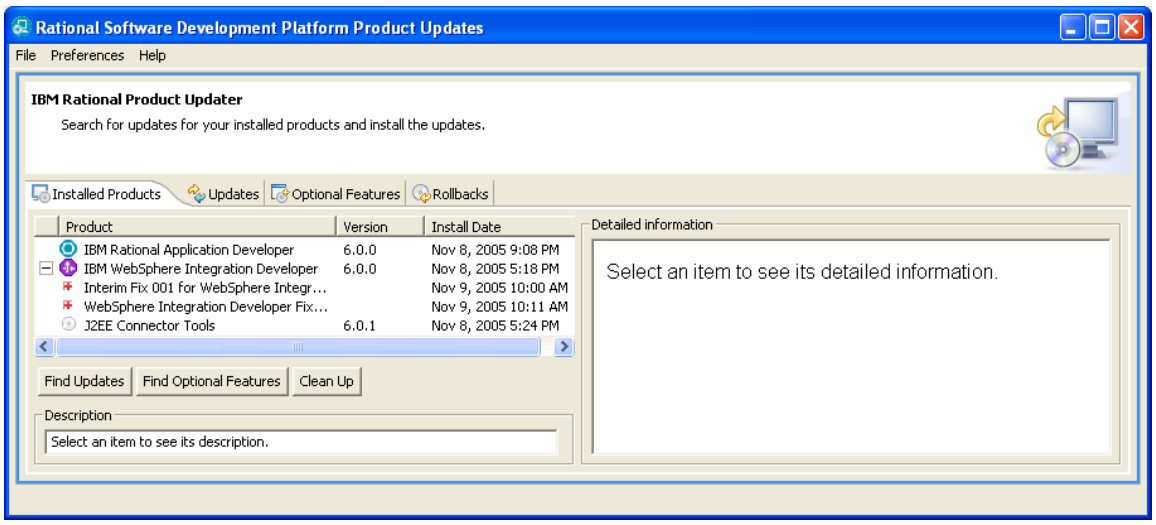
The J2EE Connector Tools provide the capability to generate java code from COBOL code. Specifically, the COBOL code that defines the COMMAREA layout in the CICS program can be processed by the tool and a java class is generated with simple setters and getters for all the fields. Under the covers, the class has all the behavior needed to properly format the fields so they can be used accurately by the program running in CICS. This feature cam be downloaded from the web, and the Rational Product Updater can find and install it for you.

We have located the COBOL definition of the structure in one of the libraries shipped with WMQ:

```
MQSERIES.V5R3M1.SCSQCOBC(CMQCIHL)
```

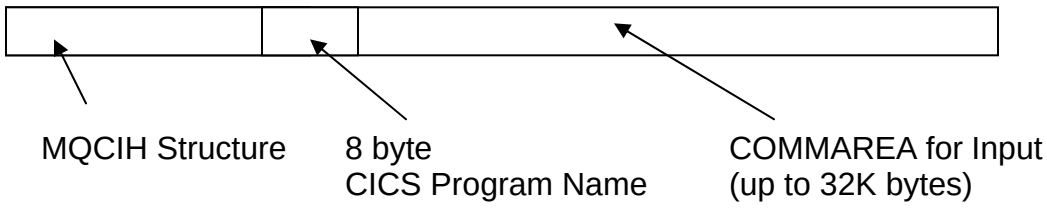
So, we can approach sending and receiving the MQCIH much like any other COMMAREA definition and use the J2EE Connector Tools to simplify the build process.

Verify you have the J2EE Connector Tools Feature installed by launching the Rational Software Development Platform Product Updater:



Third Iteration

So, on our final iteration, we will handle sending and receiving the MQCIH in the message, which now looks as follows:



The first thing we need to know is how to determine if we have one? The following line of code checks for that (where obm is the BytesMessage):

```
if((obm.getStringProperty("JMS_IBM_Format")).equals(MQC.MQFMT_CICS))
```

So, receiving a BytesMessage, we will have to check this property, and when sending the MQCIH, we will have to set it.

Further, at this time we don't need to do a great deal more with the MQCIH when we receive it, so we will just print out the fields, extracting the portion after the MQCIH for return to the browser.

When we send the MQCIH, we will initialize the fields according to the default values mentioned in the WMQ Application Programming Reference.

The MQCIH structure is represented in java as an array of bytes. We can generate the class that will help us to properly construct the array using the J2EE Connector Tools. Once we have the class, when we receive a message with an MQCIH, we simply create a new instance of the MQCIH class, and use the setBytes method: (here, cihBuffer is that portion of the message that has the structure)

```
MQCIH cih = new MQCIH();
cih.setBytes(cihBuffer);
```

When we want to send a message with a MQCIH, we create a new instance of the MQCIH class, and set the properties as illustrated in the following code segment:

```
MQCIH cih = new MQCIH();
cih.setMqcih__strucid(MQConstants.MQCIH_STRUC_ID);
cih.setMqcih__struclength(MQConstants.MQCIH_LENGTH_2);
cih.setMqcih__transactionid("    ");
cih.setMqcih__uowcontrol(MQConstants.MQCUOWC_ONLY);
cih.setMqcih__version(MQConstants.MQCIH_VERSION_2);
```

When we include this new behavior in our program, our third iteration can be tested successfully and is complete.

Reviewing the code – Using the RAD Project Interchange file

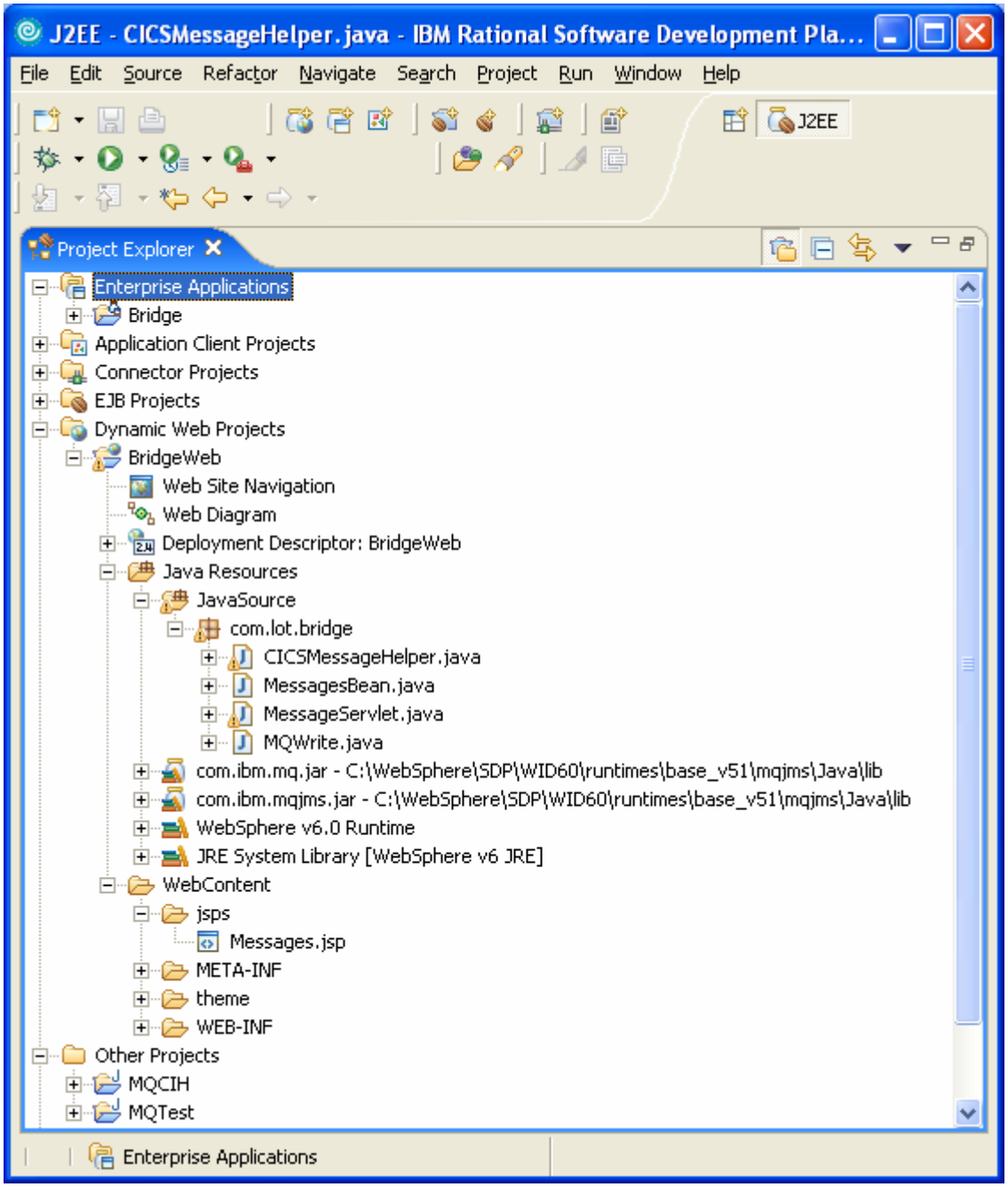
In order to review the code included in this package, the best option is to use RAD V6 and import the Project Interchange file. The following discussion will assume that's the development environment, however, you could unzip the Project Interchange file and browse the various artifacts with any tool desired.

It is best to start RAD with a new workspace. Use File→ Import to import the Project Interchange file. This will be targeted for a V6 Server by default.

After importing the file, switch to the J2EE Perspective, and examine the contents:

- Enterprise Application named Bridge – this is the EAR file which contains our Web Application and a Project Utility Jar named MQCIH.
- Dynamic Web Application named BridgeWeb – This is the web application and is composed of a Servlet, JSP, and 3 plain java classes which implement the desired functions.
- Other Projects
    - MQCIH – this class was generated by RAD, using the J2EE Connector Tools Feature. It wraps the MQCIH Structure so that we can use simple getters and setters on the structure, and not be concerned with formatting and positioning in the buffer.
    - MQWrite – this is a stand-alone java application that uses the MQ APIs directly. There is a second version of this class in the web application, so that it can be invoked from the Servlet. You can use RAD to launch this application, or you can export the class file, FTP it to z/OS, and invoke it from the command line in OMVS (USS). We won't discuss using the MQ APIs or this program in any further detail in this document, as our focus is JMS.

Use the Project Explorer to become familiar with the package.



There are three classes of primary interest in the package:

- CICSMessageHelper - where all of the behavior for interacting with WMQ via JMS has been localized.
- MQCIH –a java wrapper on the MQCIH Structure

- MessagesBean – this is a data holder class used by the servlet to pass dynamic data to the JSP.

We will discuss these three classes in some detail now.

MessagesBean

This is a plain java object. It holds data that will be passed to the JSP to be displayed in the Browser. It has two fields of interest:

- message – holds the response COMMAREA or error string extracted from the message read from the CICSB queue.
- correlID – holds the messageID retrieved from the last message that was sent. This can be used to read the specific message that is returned for this request.

CICSMessageHelper

This class has 6 methods of primary interest:

- initJMS() – this method is used after the class is first instantiated. It does JNDI lookups for the QueueConnectionFactory(qcf), a QueueDestination that is used to put messages on the CICS Bridge Queue(outgoingQueue), and a QueueDestination that is used to receive messages form the CICS response queue(incomingQueue).

- dequeueMessages(aCorrelID) – this method will read a message from the incomingQueue. If the argument is null, then the next message in the queue is obtained. If the argument is non-null, it is used to construct a JMSCorrelationID, which is used to read a specific message form the queue. This method looks at the Message that is read and determines if it is a TextMessage or a BytesMessage. If it's a BytesMessage, it is further inspected to determine if it contains an MQCIH header, and processes the structure if it is present. Regardless of the message type, the portion that is of interest to the browser end-user is extracted and returned.

- enqueueMsgJMS(aProgramName) – this method will write a JMSTextMessage to the CICS Bridge queue. It returns the messageID of the message that was sent.

- enqueueMsgJMSBytes(aProgramName) – this method will write a JMSBytesMessage to the CICS Bridge queue. It returns the messageID of the message that was sent.

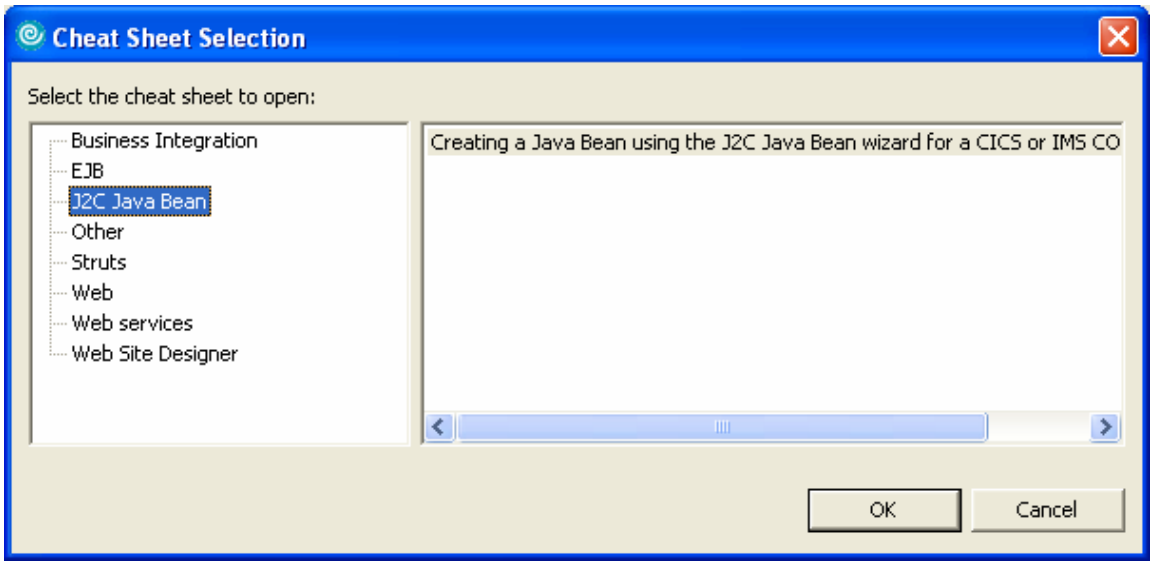
- enqueueMsgCIH(aProgramName) – this method will write a JMSBytesMessage which includes the MQCIH header. It returns the messageID of the message that was sent.

- buildCIH() – this method will create an instance of the MQCIH wrapper class, and set all of the fields to default values for WMQ. If you want CICS to use a different Transaction ID when running the program, you would change the value of that attribute in this method.

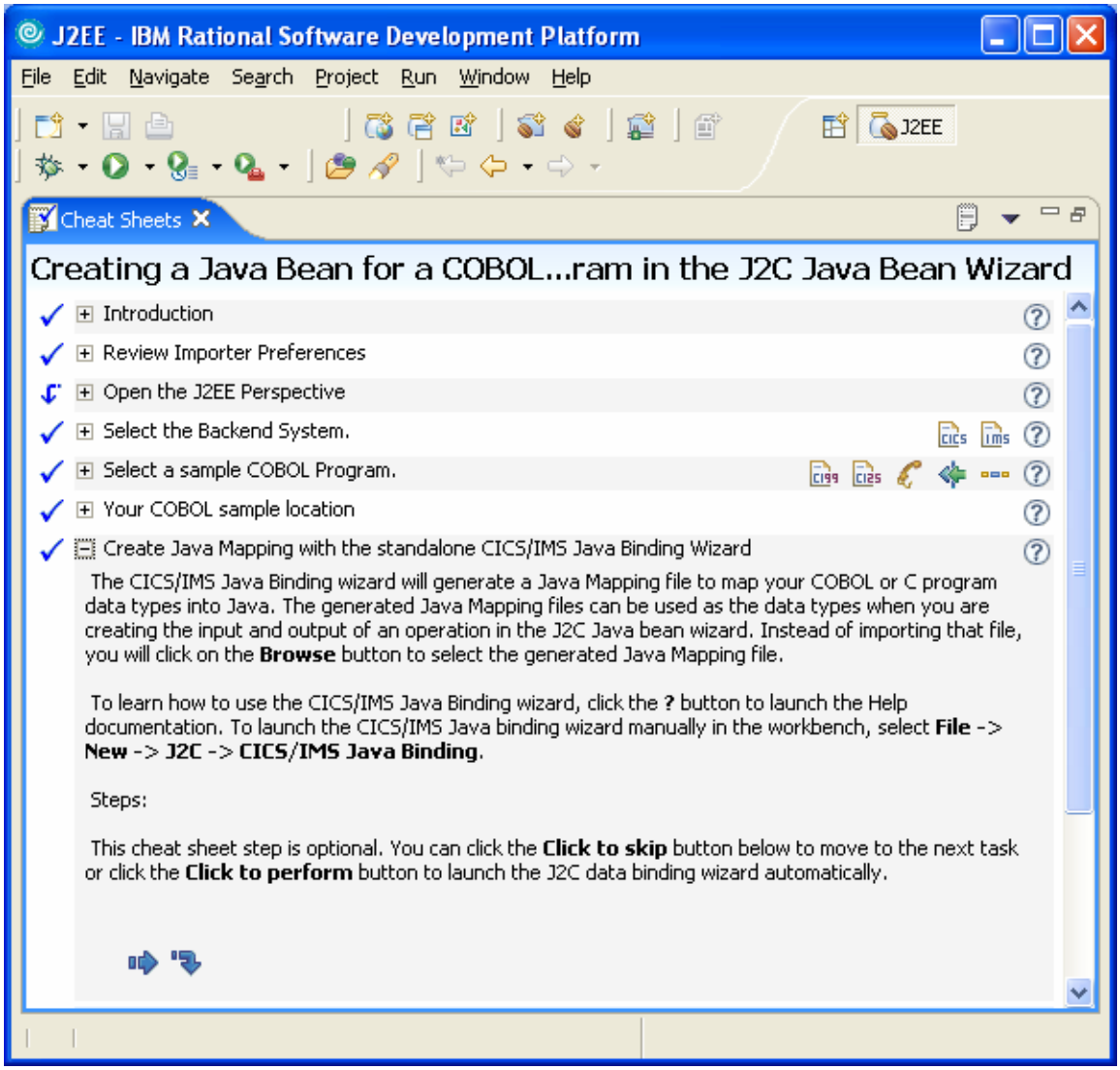
MQCIH

This class makes dealing with an MQCIH structure extremely straightforward. In Appendix A, you can see the COBOL code that defines the structure. We will briefly step through the process of generating this class using the J2EE Connector Tooling.

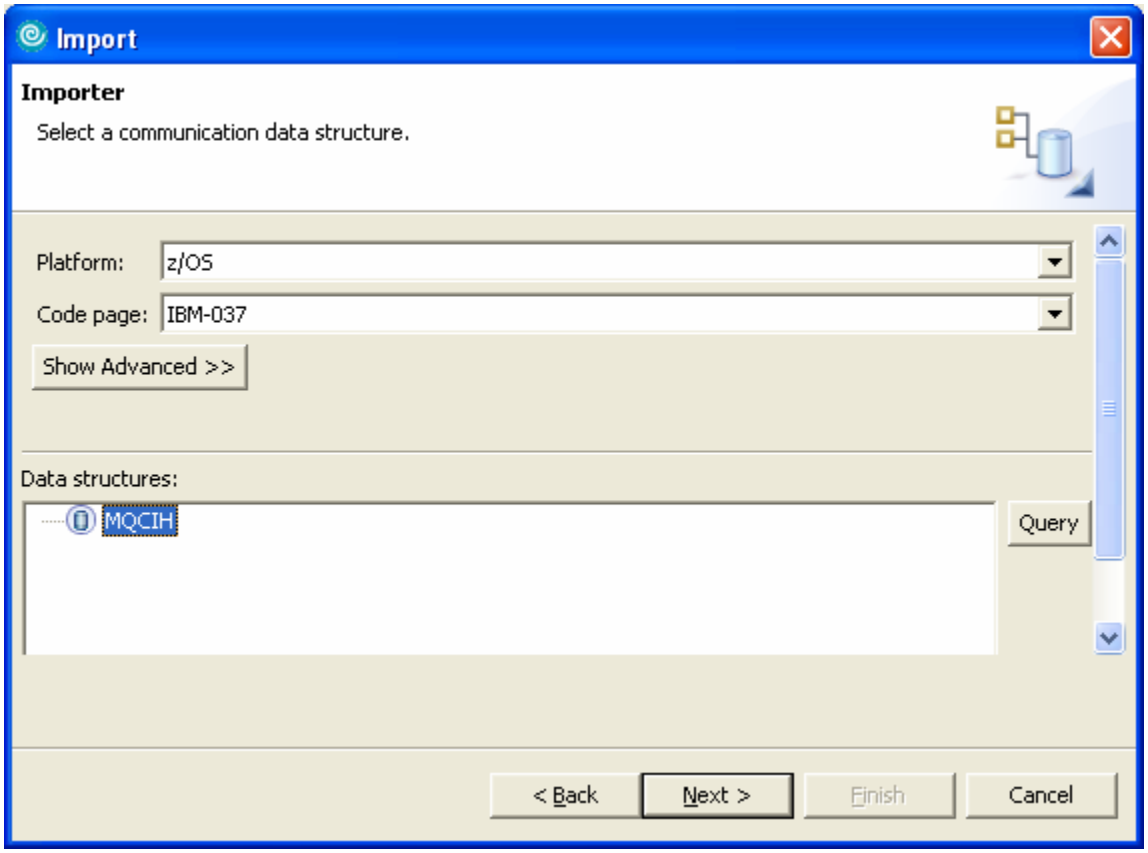
From the RAD Help menu, select Cheat Sheets:



The wizard will open and display a set of steps that it will guide you through. For our purposes, we only need to perform the first several steps, as once we have generated the java mapping, we are finished.

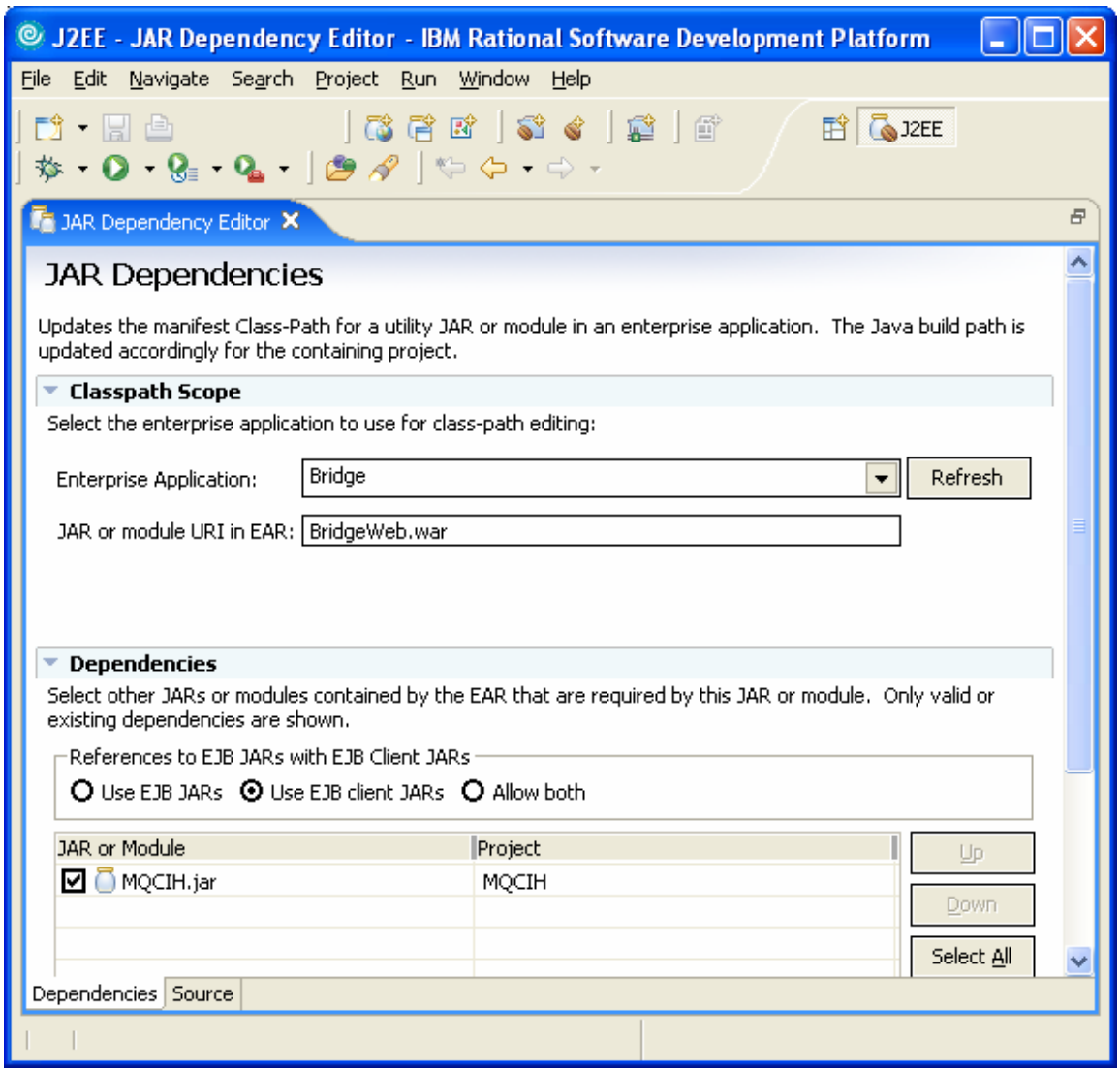


When the wizard prompts for the Cobol file, use the mqcih.cpy file that is included in this package (unless you want to build a version of the header other than WMQ V5R3M1). Make sure the Platform and Code page is set as shown. When you press the 'Query' button, the wizard will read the file, parse it, and discover the 01 Level structure.



We generated the code to a separate java project with the package name of com.messaging.cics, and named the class MQCIH.

We then invoked the Jar Dependency Editor to add the MQCIH class to our Web Application.



We will use the individual getters and setters in the MQCIH class in our code. These 3 methods are also very useful:

- getSize() – the size of the structure is 180 bytes, but we will use this method in our code to increase the ease with which we can change it later.
- getBytes() – this method returns the array of bytes that is the MQCIH. We use this when we want to set the bytes into the JMSBytesMessage.
- setBytes(aByteArray) – this method allows us to take a subset of the bytes in the JMSBytesMessage that was read form the queue and initialize an MQCIH structure from them.

Notes on the Code

Hopefully, the following two points are fairly apparent after browsing the code:

1. If you want to build messages that have complex COMMAREAS, you would use the J2C Java Bean Wizard in much the same way we have done for the MQCIH. You could generate wrapper classes for all of your COMMAREAS, and then modify the behavior of the CICSMessageHelper class to simply copy those bytes to a buffer after the MQCIH bytes have been added.

2. It may simplify the message creation code if the 8 byte program name were actually part of the COMMAREA definition used.

3. The technique used here to create an instance of the CICSMessageHelper class, which itself creates instances of the QueueConnectionFactory and QueueDestinations may not be the best approach for your application. A more common technique would be to use a Factory class which would create a Singleton of the class that has these instances. These are reusable, thread safe objects, so we really only want single instances of them.

Running the code – Deploying the EAR file

The Bridge.ear file can be immediately deployed and used to interact with the CICS program EC01. The following steps discuss how to update the server configuration and deploy the ear file.

Configuring the WAS V6 Server Environment

You can run the code inside of RAD or in a Stand Alone Server. For our test, this work was done inside of RAD, in a V6 Server. The AdminConsole panels would be identical in any V6 server. The steps are applicable to a V5 or V5.1 Server as well, although the panel content would be slightly different.

Before we deploy the ear file, we need to configure 3 JMS resources in the Server. We are using a WebSphere MQ JMS Provider:

- jms/MQM1 – a Queue Connection Factory that specifies the Host Name and Port where the WMQ Listener is running
- jms/CICS01 – a Queue Destination that will be used as the **OutgoingQueue** by the application. In our WMQ configuration, this is **SYSTEM.CICS.BRIDGE.QUEUE**
  - Be sure the Queue Destination has TargetClient set to MQ, not JMS, since the receiving program is not prepared to handle JMS Headers. configured as non-JMS
- jms/CICSB – a Queue Destination that will be used as the **IncomingQueue** by the application. This is **CICSB.**
  - Be sure the Queue Destination has TargetClient set to MQ, not JMS, since the receiving program is not prepared to handle JMS Headers. configured as non-JMS

After these definitions are made, stop and restart the server. When the server starts back up, you should see these messages in the log:

```
WSVR0049I: Binding MQM1 as jms/MQM1
WSVR0049I: Binding CICS01 as jms/CICS01
WSVR0049I: Binding CICSB as jms/CICSB
```

Now Bridge.ear can be deployed. There are two ways to do this, depending on the runtime environment:

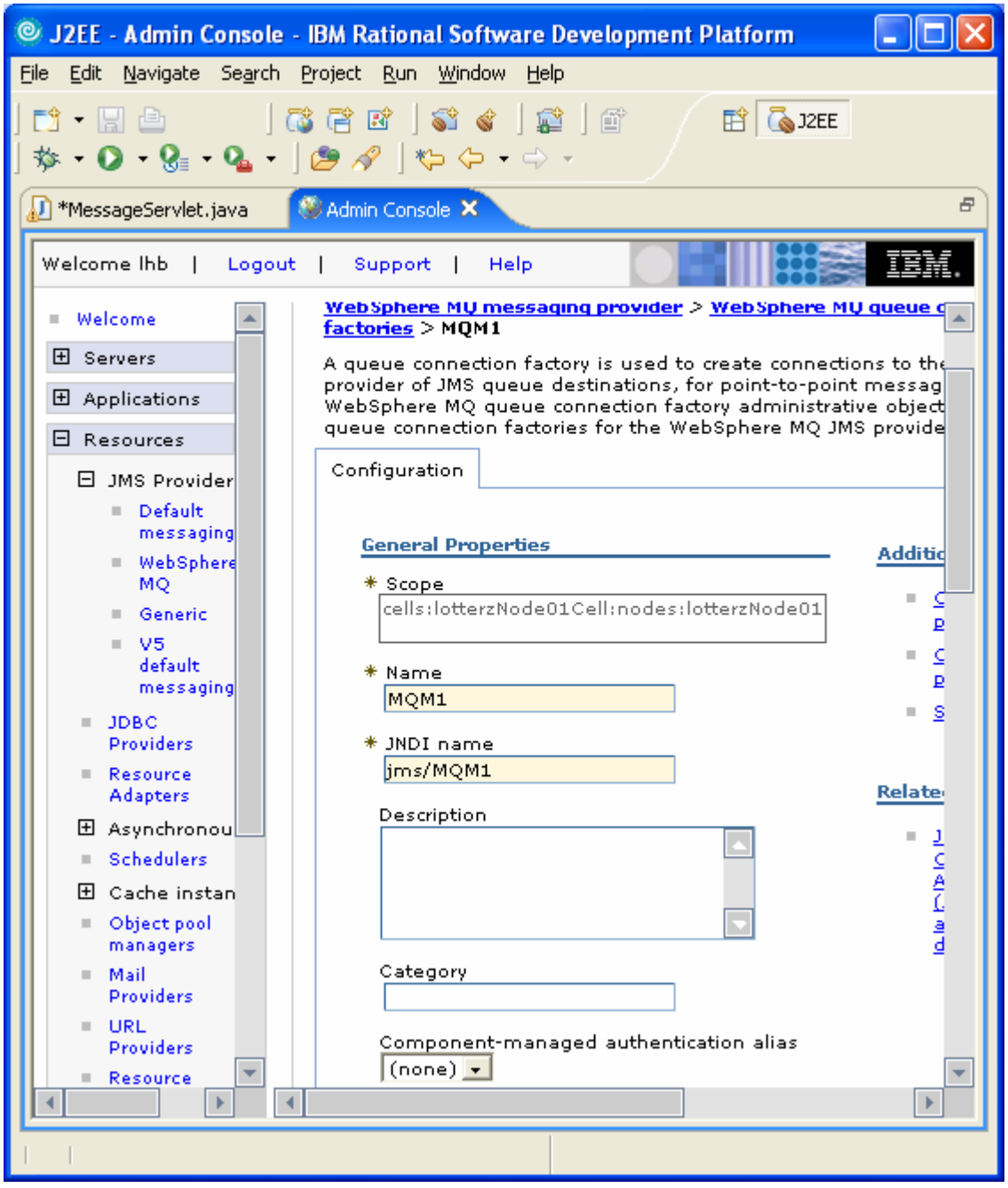
- If you are running a Test Server inside of RAD, you can use the server pop-up to Add and Remove projects. Since the JMS Resource References in the Deployment Descriptor, have the same JNDI Names that we just

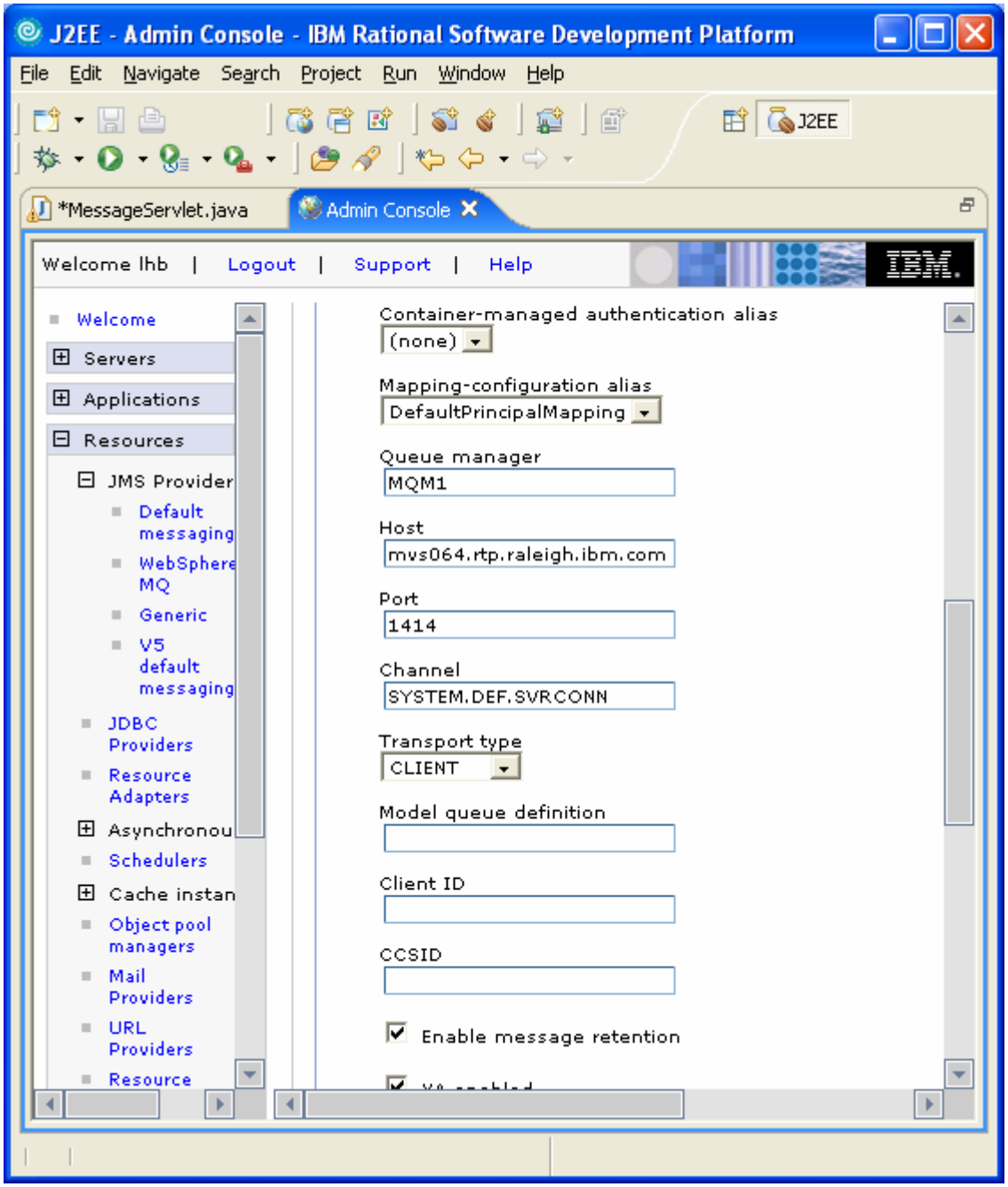
configured, this deployment will complete properly and the application will be started.

- If you are running a stand alone server, logon to the AdminConsole and deploy the application manually. You can take all of the defaults during deployment. You should notice that the Map Resource References screen shows the proper mappings. If you chose different JNDI Names for your JMS resources when you configured them, make sure you update the mappings. After you save the updated configuration, start the application.

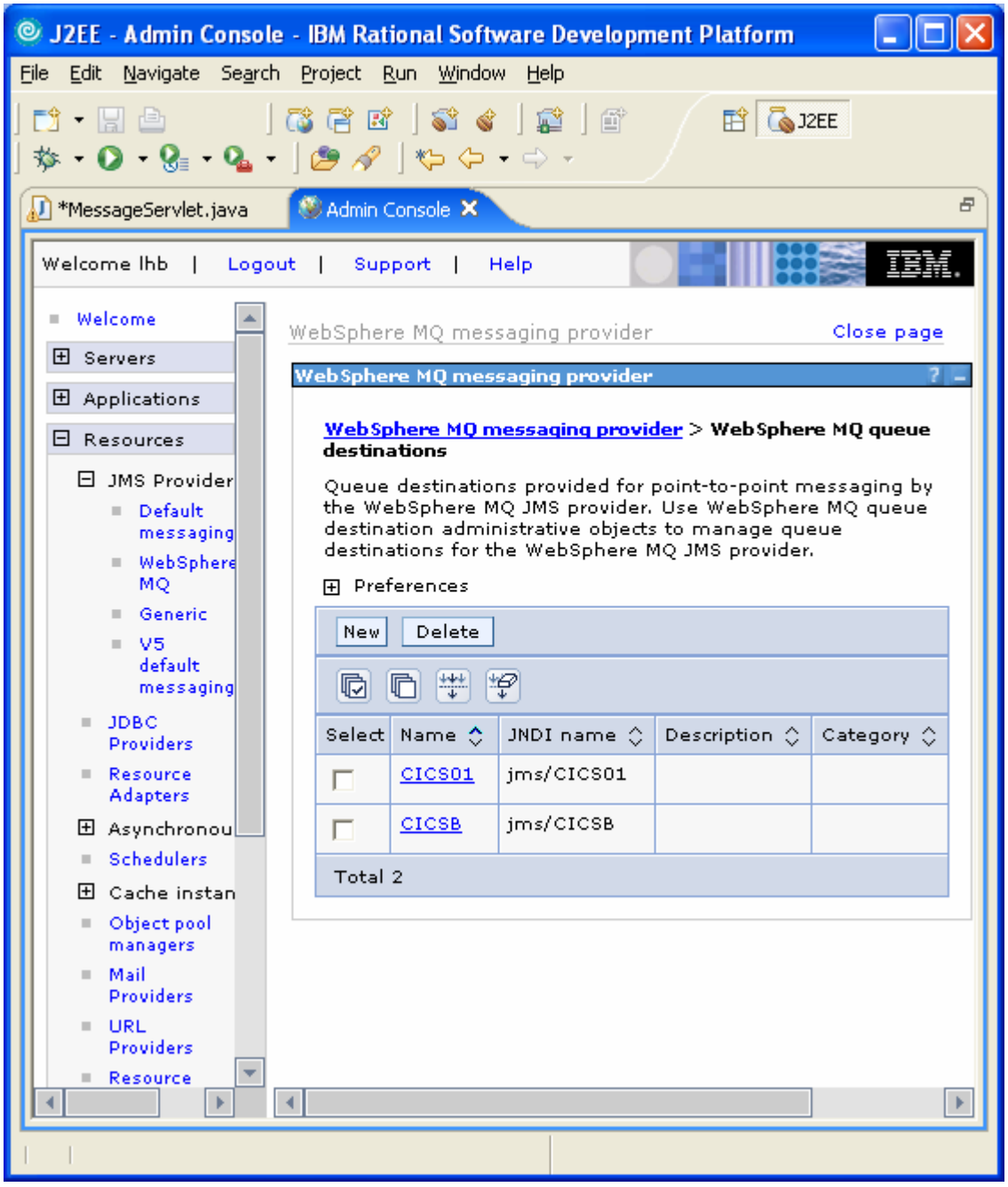
The AdminConsole panels with the resource definitions follow.

The Queue Connection Factory is shown on the next 2 pages:

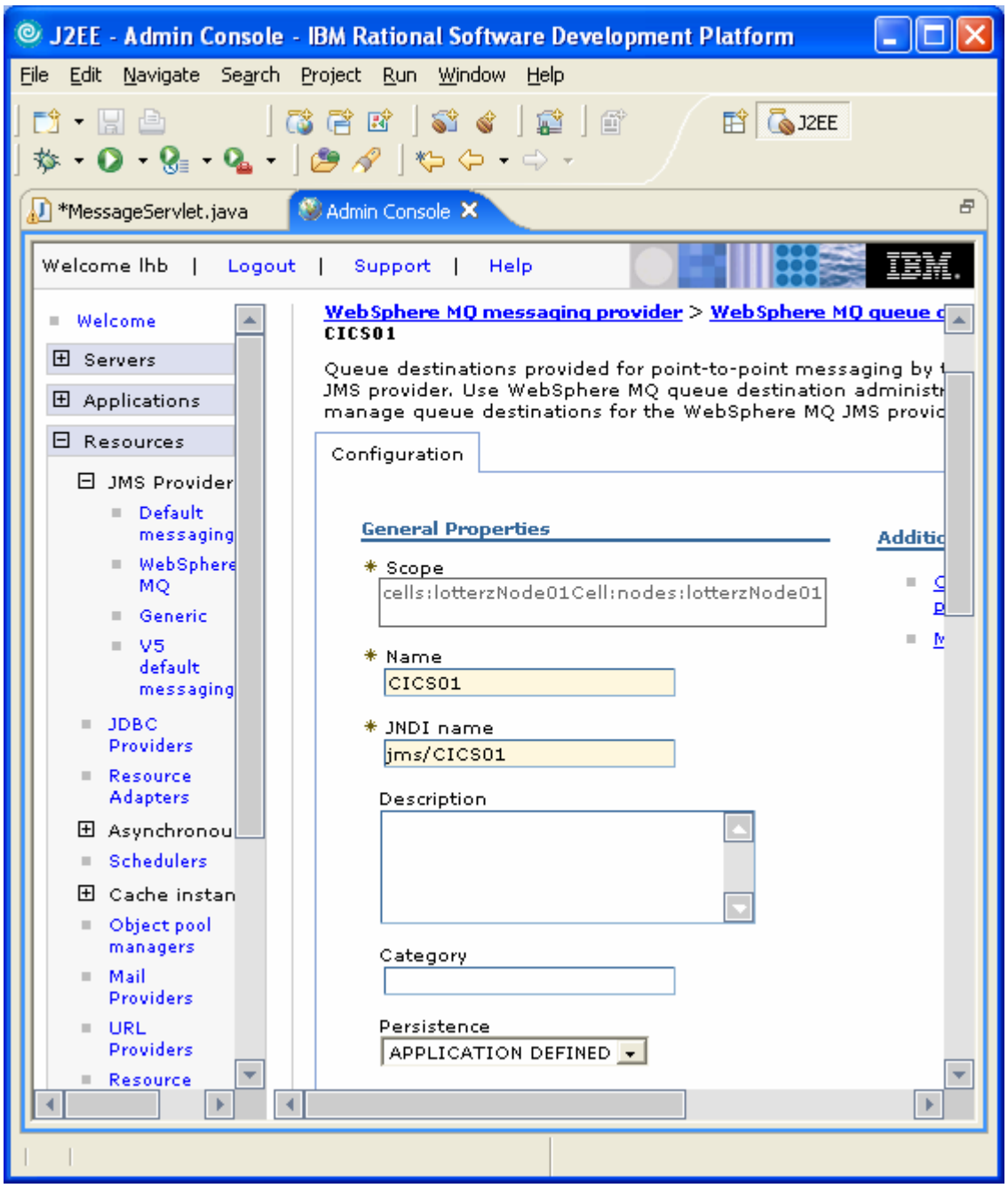


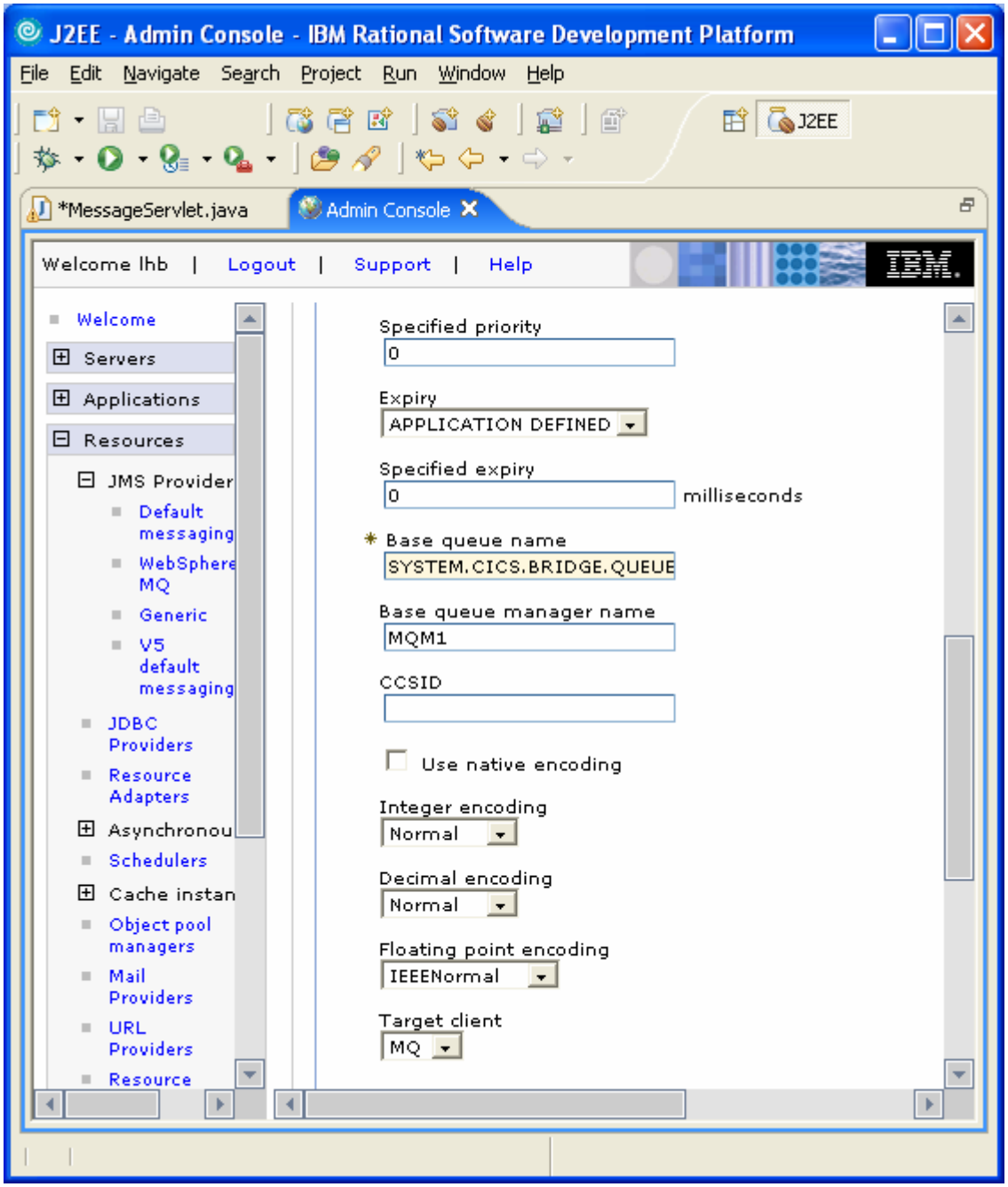

J2EE - Admin Console - IBM Rational Software Development Platform
File Edit Navigate Search Project Run Window Help
J2EE
*MessageServlet.java
Admin Console
Welcome lhb | Logout | Support | Help
IBM.
Welcome
Servers
Applications
Resources
JMS Provider
Default messaging
WebSphere MQ
Generic
V5 default messaging
JDBC Providers
Resource Adapters
Asynchronou
Schedulers
Cache instan
Object pool managers
Mail Providers
URL Providers
Resource
Container-managed authentication alias
(none)
Mapping-configuration alias
DefaultPrincipalMapping
Queue manager
MQM1
Host
mvs064.rtp.raleigh.ibm.com
Port
1414
Channel
SYSTEM.DEF.SVRCONN
Transport type
CLIENT
Model queue definition
Client ID
CCSID
Enable message retention

There are 2 Queue Destinations that must be defined:



The following panels show the details of the first Queue Destination. The second one is defined in a similar manner.




J2EE - Admin Console - IBM Rational Software Development Platform
File Edit Navigate Search Project Run Window Help
J2EE
*MessageServlet.java
Admin Console
Welcome lhb | Logout | Support | Help
IBM.
Welcome
Servers
Applications
Resources
JMS Provider
Default messaging
WebSphere MQ
Generic
V5 default messaging
JDBC Providers
Resource Adapters
Asynchronou
Schedulers
Cache instan
Object pool managers
Mail Providers
URL Providers
Resource
Specified priority
0
Expiry
APPLICATION DEFINED
Specified expiry
0
milliseconds
Base queue name
SYSTEM.CICS.BRIDGE.QUEUE
Base queue manager name
MQM1
CCSID
Use native encoding
Integer encoding
Normal
Decimal encoding
Normal
Floating point encoding
IEEENormal
Target client
MQ

Configuring WMQ and CICS on z/OS

In order to invoke a CICS transaction using the CICS Bridge function in WMQ on z/OS, we need to accomplish the following tasks:

- Configure CICS for Inter-Region Communication (Make sure IRC is Open…this was already done since we've been running the CICS Transaction Gateway)
- Deploy EC01 (ECIDateTime) sample program in CICS (again, already done)
- Configure CICS to use the Bridge, which also requires configuration of the CICS Adapter (although we have no programs that use the MQI directly)
- Configure CICS to start a connection to WMQ automatically
- Configure WMQ queues and process required for the Adapter and the Bridge

We made extensive use of the MQ Publications during this activity. In the appendix is a reference to the MQ InfoCenter, which can be downloaded and installed on the development desktop, which eased access considerably.

Using the MQ System Setup Guide as a starting point, the following tasks were completed.

Setting up the CICS Adapter

The MQ System Setup Guide takes you through the steps you must perform to define the resources for the CICS Adapter.

1. You must use resource definition online (RDO) to add new groups to the CSD dataset. The new groups must contain definitions of:
   - The supplied adapter programs
   - The supplied adapter management transactions
   - The supplied sets of BMS maps, required for the adapter panels

To update the CSD, run the CICS offline utility program, DFHCSDUP, with the supplied sample input datasets:

- thlqual **SCSQPROC(CSQ4B100)** - CICS adapter

  ```
  * NAME: CSQ4B100
  *
  * CICS resource definitions for the WebSphere MQ CICS adapter
  ADD GROUP(CSQCAT1) LIST(yourlist)
  ```

2. Add the following WebSphere MQ libraries to the STEPLIB concatenation in your CICS procedure in the following order:

- thlqual.SCSQANLx
- thlqual.SCSQAUTH

Where x is the language letter for your national language.

3. Add the following WebSphere MQ libraries to the DFHRPL concatenation in your CICS procedure in the following order, even if they are in the LPA or link list:

- thlqual.SCSQANLx
- thlqual.SCSQCICS
- thlqual.SCSQAUTH

Where x is the language letter for your national language.

Setting up the CICS Bridge

The MQ System Setup Guide takes you through the steps you must perform to define the resources for the CICS Bridge.

1. Run the resource definition utility DFHCSDUP, using the sample thlqual.**SCSQPROC(CSQ4CKBC)** as input, to define the bridge transactions and programs:

- CKBR Bridge monitor transaction
- CSQCBCDI Data conversion exit
- CSQCBR00 Bridge monitor program
- CKBP Bridge ProgramLink transaction
- CSQCBP00 Bridge ProgramLink program
- CSQCBP10 Bridge ProgramLink abend handler program
- CSQCBE00 3270 bridge exit for WebSphere MQ (CICS Transaction Server, Version 1.2)
- CSQCBE30 3270 bridge exit for WebSphere MQ (CICS Transaction Server, Version 1.3)

```
* NAME: CSQ4CKBC
*
* CICS resource definitions for the WebSphere MQ-CICS bridge
ADD GROUP(CSQCKB)  LIST(yourlist)
```

2. Add the group, CSQCKB, to your startup group list.

Notes:

1. The bridge uses CICS temporary storage IDs with the prefix CKB. You should make sure these are not recoverable.
2. By default, your CICS DPL programs are run under transaction code

CKBP. The transaction to be run can be specified in the MQCIH CICS-bridge header in the message. For more information, see *WebSphere MQ Application Programming.*

Updating the SIT in CICS

Use the INITPARM parameter in the CICS system initialization table (SIT), or the SYSIN override, to set the default connection parameters.

- MQCONN={NO|YES} specifies whether you want CICS to start the MQSeries for z/OS connection automatically during initialization.
  - **NO** Do not automatically invoke CSQCCODF, the MQSeries attach program, during initialization.
  - **YES** Invoke the MQSeries attach program, CSQCCODF, automatically during CICS initialization. The other information CICS needs for starting the attachment, such as the MQSeries queue manager subsystem name, is taken from the CSQCPARM operand of an INITPARM system initialization parameter.

As an example, you might code the initparm as follows:

```
INITPARM=(CSQCPARM='SN=CSQ1,TN=001,IQ=CICS01.INITQ')
```

Setting up WMQ

There are 3 sets of objects that must be configured.

1. Make sure you have the initiation queue used by the trigger task defined. **SCSQPROC(CSQ5INYG)** has a sample:

```
DEFINE QLOCAL( 'CICS01.INITQ' ) +   *****CSQ5INYG******
```

2. Next, define the CICS Bridge Queue and the Process to run when the trigger even occurs. **SCSQPROC(CSQ4CKBM)** has a sample:

```
* NAME: CSQ4CKBM
*
* CSQINP2 sample for using the WebSphere MQ-CICS bridge
************************************************************************
  DEFINE QLOCAL('SYSTEM.CICS.BRIDGE.QUEUE') REPLACE +  …
*                               Trigger on 1st message
*                               using default initq
       TRIGGER TRIGTYPE(FIRST) +
       PROCESS('CICS_BRIDGE') +
       INITQ('CICS01.INITQ')
*
```

```
DEFINE PROCESS('CICS_BRIDGE') REPLACE +
    DESCR('CICS BRIDGE MONITOR') +
    APPLICID('CKBR') +
    APPLTYPE(CICS) +
    USERDATA('AUTH=LOCAL,WAIT=20')
```

3. Finally, define a Local Queue to use as the ReplyTo Queue by the application. This is where the Bridge will place the output from CICS.

```
DEFINE QLOCAL( 'CICSB' ) +
```

.

Testing the Bridge

We need to verify the z/OS environment is ready, and make sure the WAS Server is running.

Starting the z/OS Environment

First, be sure that the QMgr is running on z/OS. We used the following commands:

- +MQM1 START QMGR
- +MQM1 START CHINIT
- +MQM1 START LISTENER

You will see these messages in the JES2 log for the Channel Initiator:

CSQX022I +MQM1 CSQXSUPR Channel initiator initialization complete
CSQX251I +MQM1 CSQXSTRL Listener started, TRPTYPE=TCP INDISP=QMGR
CSQX023I +MQM1 CSQXLSTT Listener started for port 1414 address *,
TRPTYPE=TCP INDISP=QMGR

Later, when the WAS V6 Server connects, you will see:

CSQX500I +MQM1 CSQXRESP Channel SYSTEM.DEF.SVRCONN started
CSQX500I +MQM1 CSQXRESP Channel SYSTEM.DEF.SVRCONN started
CSQX501I +MQM1 CSQXRESP Channel SYSTEM.DEF.SVRCONN is no longer active

Next, after CICS is started, look in the JES2 log to verify the Adapter and the Bridge have been defined:

CICS01 Install for group CSQCAT1 has completed successfully
CICS01 Install for group CSQCKB has completed successfully

Also, look in the JES2 log for messages pertaining to the connection with MQ:

+CSQC307I CICS01 CSQCCON Successful connection to (qm) MQM1
+DFHSI8441I CICS01 Connection to MQ MQM1 successfully completed
+DFHSI1517 CICS01 Control is being given to CICS.

Later, when the first message is placed on the SYSTEM.CICS.BRIDGE.QUEUE you will see the Bridge Monitor get started, This is because the Adapter (CKTI) was informed by WMQ that a message arrived.

CSQC700I CKBR 0000028 IBM WebSphere MQ for z/OS V5.3.1 - CICS bridge.
CSQC702I CKBR 0000028 Monitor initialization complete
CSQC703I CKBR 0000028 Auth=LOCAL, WaitInterval=20000,

Q=SYSTEM.CICS.BRIDGE.QUEUE

During your testing, if you want to verify how many messages are on the queues, you can issue commands similar to the following:

```
+MQM1 DISPLAY QLOCAL(CICSB) CURDEPTH
+MQM1 DISPLAY QLOCAL(SYSTEM.CICS.BRIDGE.QUEUE) CURDEPTH
```

Starting the WAS Test Environment

Make sure the test server is running, and that the Bridge application is started.

You should be certain you have TCP/IP access to the z/OS Host. (e.g. can resolve the host name and get through any firewalls).

Test Results

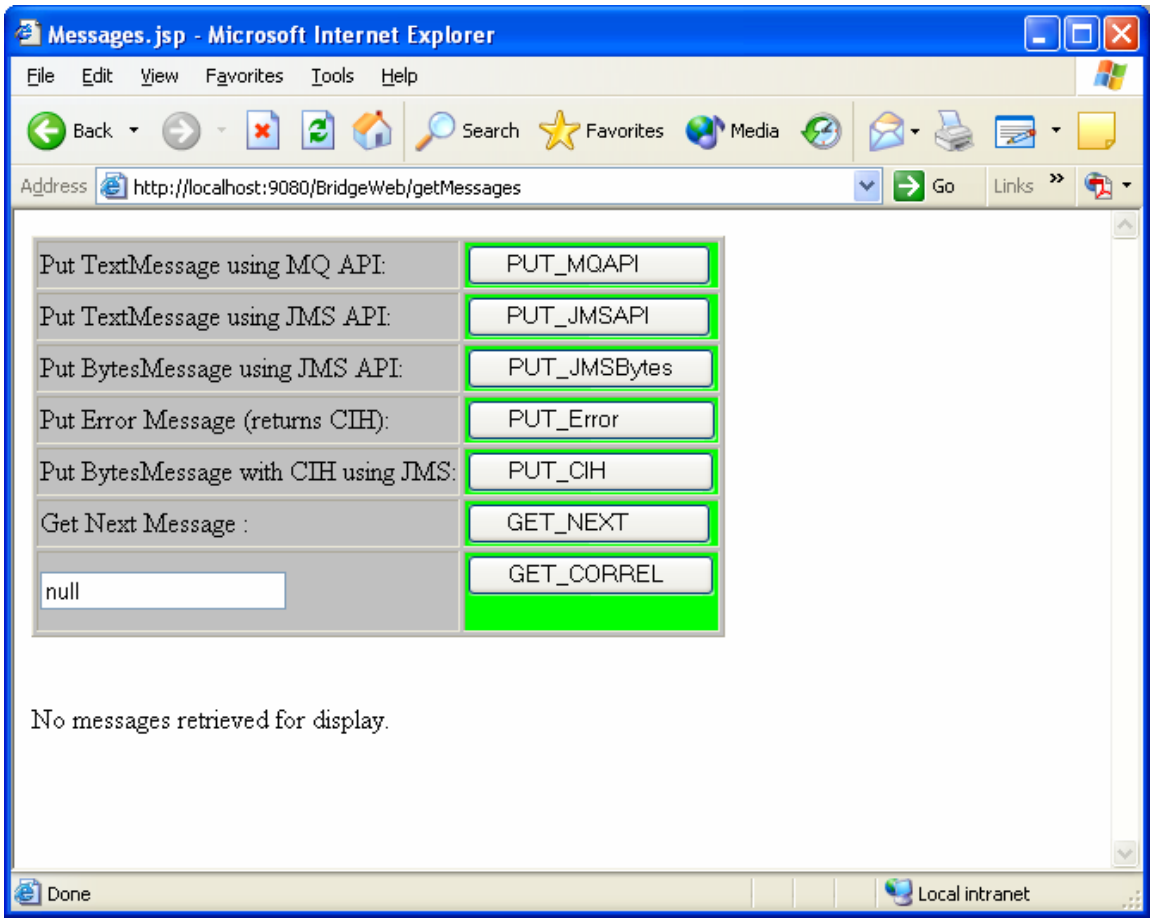
The following sections discuss the initial test steps you can perform to validate the configuration of the z/OS subsystems and the functioning of JMS in WAS. After successfully executing those steps, a test scenario is presented which allows a custom Transaction ID to be used in CICS. Lastly, several observations about the testing are made.

Basic Tests

Invoke the application with the following URL

http://<your_host:port>/BridgeWeb/getMessages

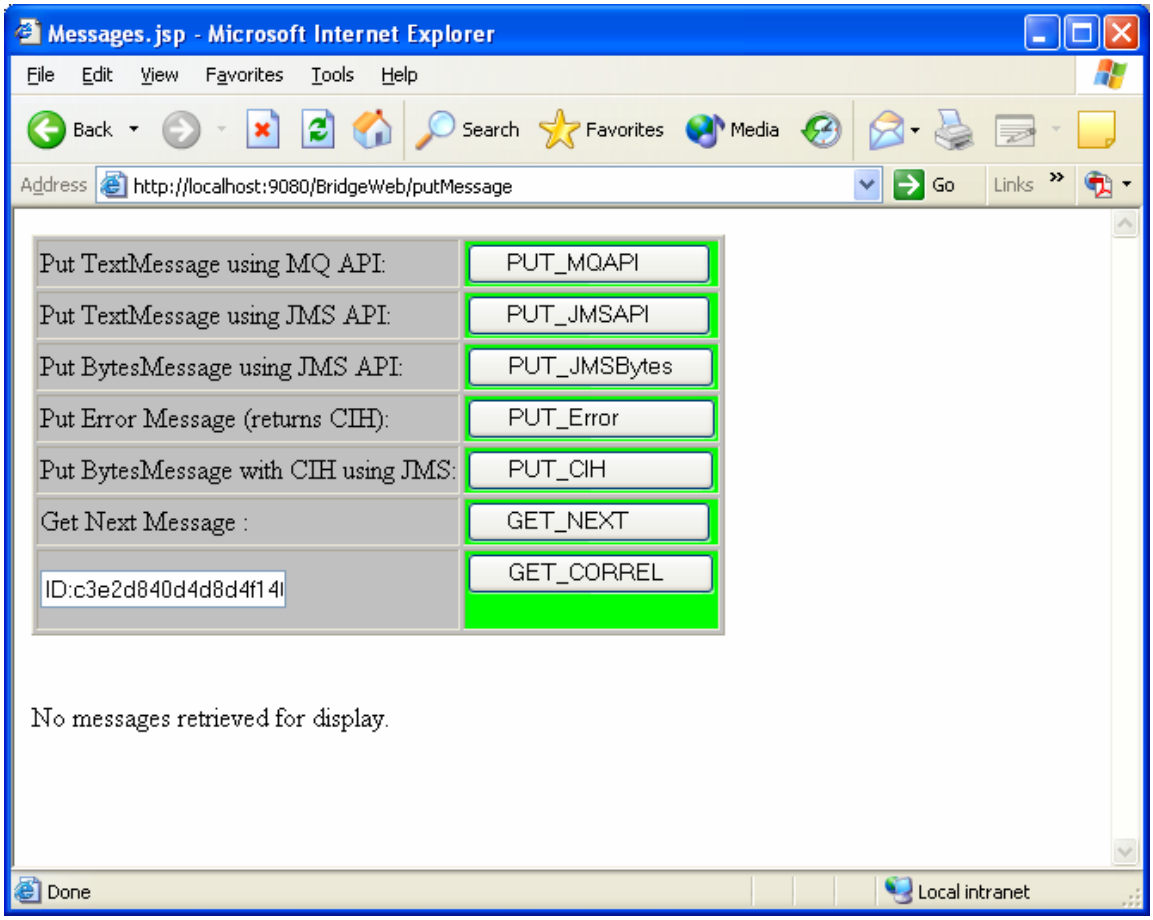
The initial browser page should appear:



If you press the “PUT_JMSAPI” button, you will see the following in the log:

```
SystemOut     O Sent a TextMessage
```

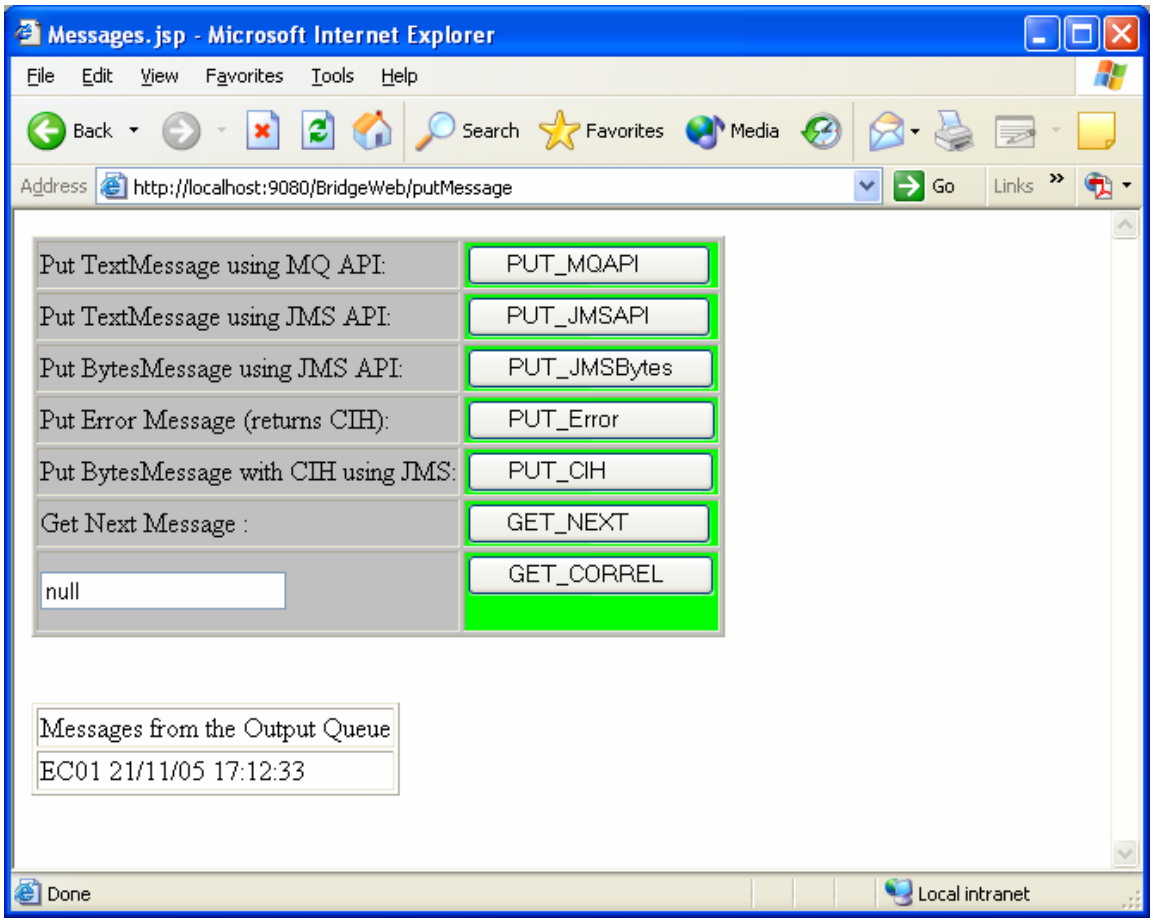
And the browser will be refreshed with the following screen (note that the MessageID is filled in (this is the MessageID from the message that was just sent)



If you press “GET_CORREL”, a GET request to the CICSB queue will be made, using the MessageID as the correlation ID. The following will be displayed in the log:

```
SystemOut     O Using selector:JMSCorrelationID =
'ID:c3e2d840d4d8d4f14040404040404040bdf2ade32d00a3ae'
SystemOut     O Read a TextMessage
SystemOut    O Correl= ID:c3e2d840d4d8d4f14040404040404040bdf2ade32d00a3ae
```

The browser will contain:



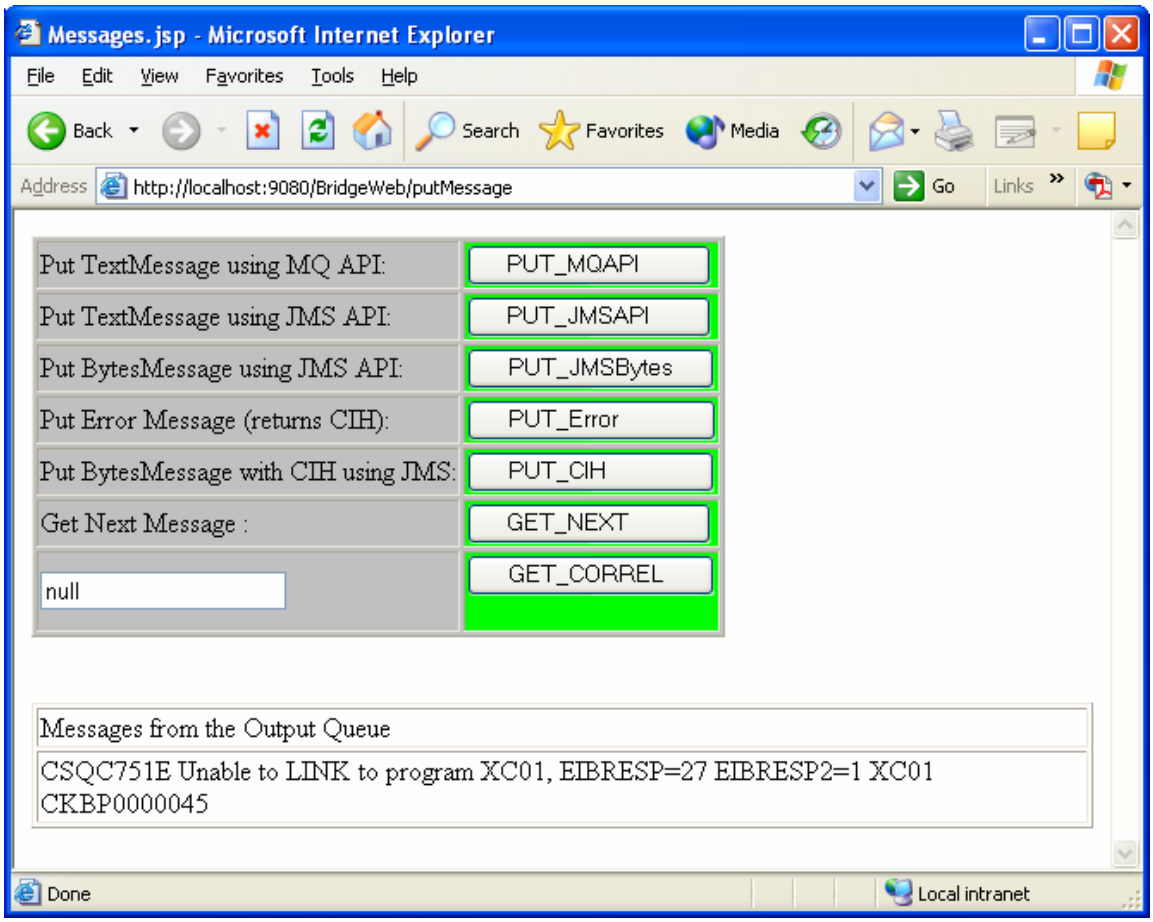
The message returned is displayed. You can see that the Bridge keeps the 8-byte Program Name at the beginning of the message, and appends the COMMAREA returned from the DPL request. In our case, the 18 bytes are the date and time from EC01.

The “GET_NEXT” function does not use a correlation ID when it does the GET, so it’s a good way to empty the queue.

Next, press the “PUT_Error” button. This function will send a TextMessage, just as the prior function did. The console will display:

```
SystemOut     O Sent a TextMessage
```

However, the returned message is NOT a TextMessage as it was prior.
Press the “GET_CORREL” button and the browser will display:



A look at the console log shows:

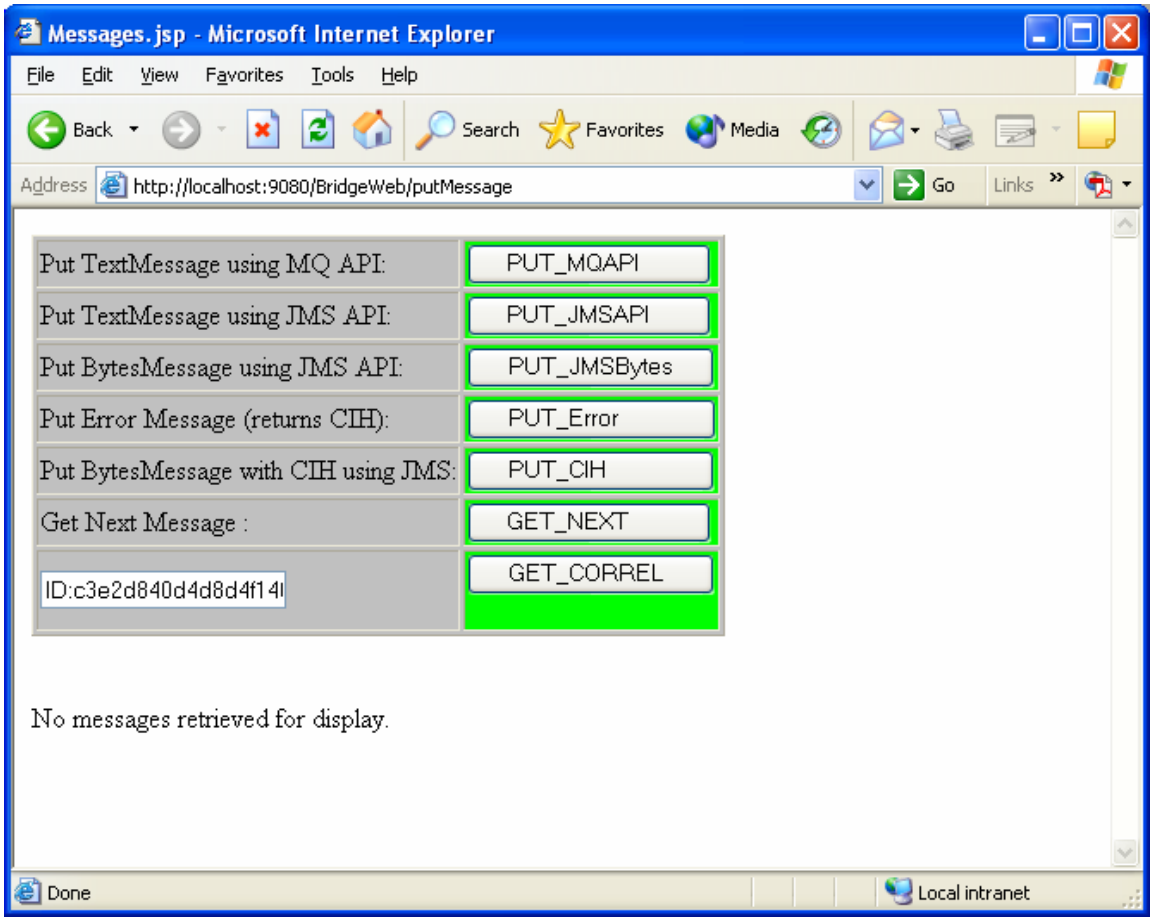
```
SystemOut     O Using selector:JMSCorrelationID =
'ID:c3e2d840d4d8d4f14040404040404040bdf2b1182d28882e'
SystemOut     O
SystemOut     O ------------------------------
SystemOut     O OUTPUT BYTES MESSAGE
SystemOut     O ------------------------------
SystemOut     O
JMS Message class: jms_bytes
  JMSType:          null
  JMSDeliveryMode: 2
  JMSExpiration:   0
  JMSPriority:     0
  JMSMessageID:    ID:c3e2d840d4d8d4f14040404040404040bdf2b1182d28882e
  JMSTimestamp:    1132612014270
  JMSCorrelationID:ID:c3e2d840d4d8d4f14040404040404040bdf2b1182d28882e
  JMSDestination:  null
```

```
  JMSReplyTo:      null
  JMSRedelivered:  false
  JMSXDeliveryCount:1
  JMS_IBM_MsgType:2
  JMSXAppID:CICS01  CKBP
  JMS_IBM_Format:MQCICS
  JMS_IBM_Encoding:785
  JMS_IBM_PutApplType:1
  JMS_IBM_Character_Set:500
  JMSXUserID:CICSUSER
  JMS_IBM_PutTime:22265427
  JMS_IBM_PutDate:20051121
Integer encoding: 1, Floating point encoding 768
c3c9c840000000020000000b4000000000000000d4d8e2e3d940404000000000000000070000001b
0000000100000110ffffffe00000001ffffffff0000000000000000000000000000000000000000
00000000f0f2f0f8004040405c5c5c5c5c5c5c5c404040404040404040404040404040404040404
4040404040404040404040404040404040404040404040404040404040404040404040404040404
4040404000000000000000000000000000000000c3e2d8c3f7f5f1c540e4958182938540a39640d3
c9d5d240a396409799968799819440e7c3f0f16b40c5c9c2d9c5e2d77ef2f740c5c9c2d9c5e2d7f2
7ef1400040404040404040404040404040404040404040404040404040404040404040404040404
4040404040404040404040404040404040404000e7c3f0f140404040c3d2c2d7f0f0f0f0f0f4f500
```

This illustrates that the code properly recognized the BytesMessage that was returned. In fact, it returns the interesting portion to the browser. Further, it determined that the message contained an MQCIH Structure, and displays those fields, as well:

```
SystemOut     O Read a MQCIH BytesMessage
SystemOut     O Number of bytes = 319
SystemOut     O
SystemOut     O Abend Code =
SystemOut     O Comp Code = 27
SystemOut     O Format = MQSTR
SystemOut     O Ouput Data Length = -1
SystemOut     O Reason Code = 1
SystemOut     O Transaction ID =
SystemOut     O Task End Status = 0
SystemOut     O ADS Descriptor = 0
SystemOut     O Attention ID =
SystemOut     O Authenticator = ********
SystemOut     O Cancel Code =
SystemOut     O CCSID = 0
SystemOut     O Conversational Task = 0
SystemOut     O Cursor Posiition = 0
SystemOut     O Encoding = 0
SystemOut     O Error Offset = 0
SystemOut     O Facility =
SystemOut     O Facility Keep Time = 0
SystemOut     O Facility Like =
SystemOut     O Flags = 0
SystemOut     O Function = 0208
SystemOut     O Get Wait Interval = -2
SystemOut     O Input Item = 0
SystemOut     O Link Type = 1
SystemOut     O Next Transaction ID =
SystemOut     O Remote System ID =
SystemOut     O Remote Trans ID =
SystemOut     O ReplyTo Format =
SystemOut     O Return Code = 7
SystemOut     O Start Code =
SystemOut     O Structure ID = CIH
SystemOut     O Structure Length = 180
SystemOut     O UOW Control = 272
SystemOut     O Version = 2
```

Now press the ‘Put_CIH’ button:



Look at the console log to view the message that is built and PUT to the SYSTEM.CICS.BRIDGE.QUEUE:

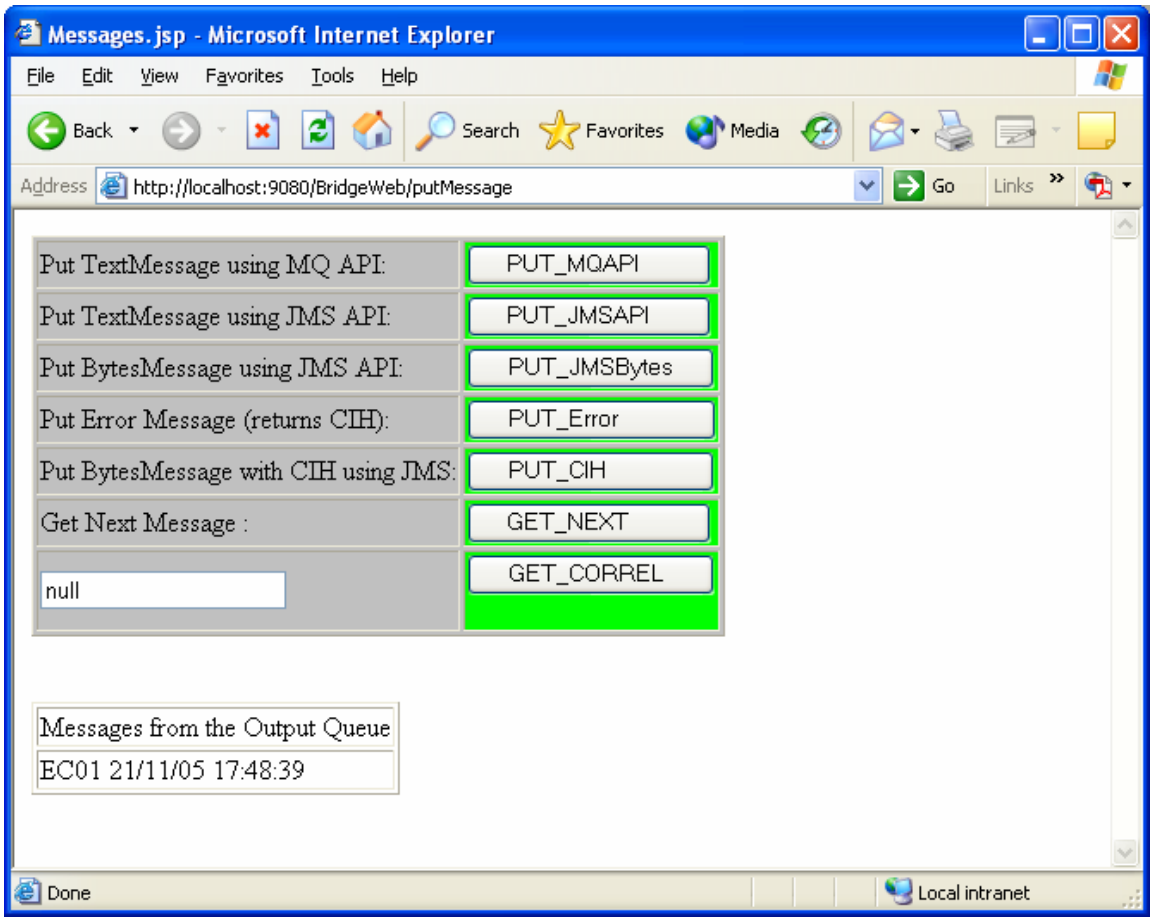
```
SystemOut     O CIH Request processing...
SystemOut     O
SystemOut     O ----------------------------
SystemOut     O INPUT BYTES MESSAGE
SystemOut     O ----------------------------
SystemOut     O
JMS Message class: jms_bytes
  JMSType:          null
  JMSDeliveryMode: 2
  JMSExpiration:   0
  JMSPriority:     4
  JMSMessageID:    null
  JMSTimestamp:    0
  JMSCorrelationID:AMQ!NEW_SESSION_CORRELID
  JMSDestination:  null
  JMSReplyTo:      queue://MQM1/CICSB?targetClient=1
  JMSRedelivered:  false
  JMS_IBM_Encoding:785
  JMS_IBM_Format:MQCICS
```

```
  JMS_IBM_Character_Set:500
Integer encoding: 1, Floating point encoding 256
c3c9c840000000020000000b400000000000000004040404040404040000000000000000000000000
000000000000011100000000000000001ffffffff000000000000000000000000000000000000000
000000004040404040404040404040404040404040404040404040404040404040404040404040
40404040e7d2c2d7404040404040404040404040404040404040404040404040404040404040
404040400000000000000000000000000000000c5c3f0f1404040404040404040404040404040
4040404040404040
```

```
SystemOut     O Abend Code =
SystemOut     O Comp Code = 0
SystemOut     O Format =
SystemOut     O Ouput Data Length = -1
SystemOut     O Reason Code = 0
SystemOut     O Transaction ID = XKBP
SystemOut     O Task End Status = 0
SystemOut     O ADS Descriptor = 0
SystemOut     O Attention ID =
SystemOut     O Authenticator =
SystemOut     O Cancel Code =
SystemOut     O CCSID = 0
SystemOut     O Conversational Task = 0
SystemOut     O Cursor Posiition = 0
SystemOut     O Encoding = 0
SystemOut     O Error Offset = 0
SystemOut     O Facility =
SystemOut     O Facility Keep Time = 0
SystemOut     O Facility Like =
SystemOut     O Flags = 0
SystemOut     O Function =
SystemOut     O Get Wait Interval = 0
SystemOut     O Input Item = 0
SystemOut     O Link Type = 1
SystemOut     O Next Transaction ID =
SystemOut     O Remote System ID =
SystemOut     O Remote Trans ID =
SystemOut     O ReplyTo Format =
SystemOut     O Return Code = 0
SystemOut     O Start Code =
SystemOut     O Structure ID = CIH
SystemOut     O Structure Length = 180
SystemOut     O UOW Control = 273
SystemOut     O Version = 2
```

Actually, the Transaction ID will be all blanks if you use the code as it is currently packaged.

Now press one of the GET buttons:



The console log will contain:

```
SystemOut     O Using selector:JMSCorrelationID =
'ID:c3e2d840d4d8d4f14040404040404040bdf2b5f4d1b0046a'
SystemOut     O
SystemOut     O ----------------------------
SystemOut     O OUTPUT BYTES MESSAGE
SystemOut     O ----------------------------
SystemOut     O
JMS Message class: jms_bytes
  JMSType:         null
  JMSDeliveryMode: 2
  JMSExpiration:   0
  JMSPriority:     4
  JMSMessageID:    ID:c3e2d840d4d8d4f14040404040404040bdf2b5f4d1b0046a
  JMSTimestamp:    1132613319300
  JMSCorrelationID:ID:c3e2d840d4d8d4f14040404040404040bdf2b5f4d1b0046a
  JMSDestination:  null
  JMSReplyTo:      queue://MQM1/SYSTEM.CICS.BRIDGE.QUEUE
  JMSRedelivered:  false
  JMSXDeliveryCount:1
  JMS_IBM_MsgType:2
  JMSXAppID:CICS01  XKBP
```

```
  JMS_IBM_Format:MQCICS
  JMS_IBM_Encoding:785
  JMS_IBM_PutApplType:1
  JMS_IBM_Character_Set:500
  JMSXUserID:CICSUSER
  JMS_IBM_PutTime:22483930
  JMS_IBM_PutDate:20051121
Integer encoding: 1, Floating point encoding 768
c3c9c840000000020000000b400000000000000004040404040404040000000000000000000000000
000000000000011100000000000000001ffffffff000000000000000000000000000000000000000000
000000004040404040404040405c5c5c5c5c5c5c5c4040404040404040404040404040404040404040
40404040e7d2c2d740404040404040404040404040404040404040404040404040404040404040404040
404040400000000000000000000000000000000c5c3f0f140404040f2f161f1f161f0f540f1f77a
f4f87af3f9004040
```

Again, and initially, the XKBP will not be present in the JMSXAppID attribute.

Modifying the Default Transaction ID

Up to this point in our testing, we have been using the default transaction in CICS to run our program. The MQCIH header is required if you have different transaction ids that you need to use.

In order to test the case with custom Transaction ID, we first need to create the new Transaction in CICS. Do the following to accomplish that:

1. Log onto CICS and start the CEDA transaction.
2. **EX GR(CSQCKB)**
3. Scroll down until you see Transaction CKBP
4. Next to this item, type **C AS(XKBP)**
5. Press the PF12 key, then the Enter key to reopen this group
6. Scroll down until you see Transaction XKBP
7. Next to this item, type **IN**

You have now installed a new transaction definition for the DPL Bridge.

Now we need to modify our java application to set the TransactionID in the MQCIH to "XKBP". Open the CICSMessageHelper class and find the buildCIH() method. Change the following line of code:

```
cih.setMqcih__transactionid("    ");
```

to:

```
cih.setMqcih__transactionid("XKBP");
```

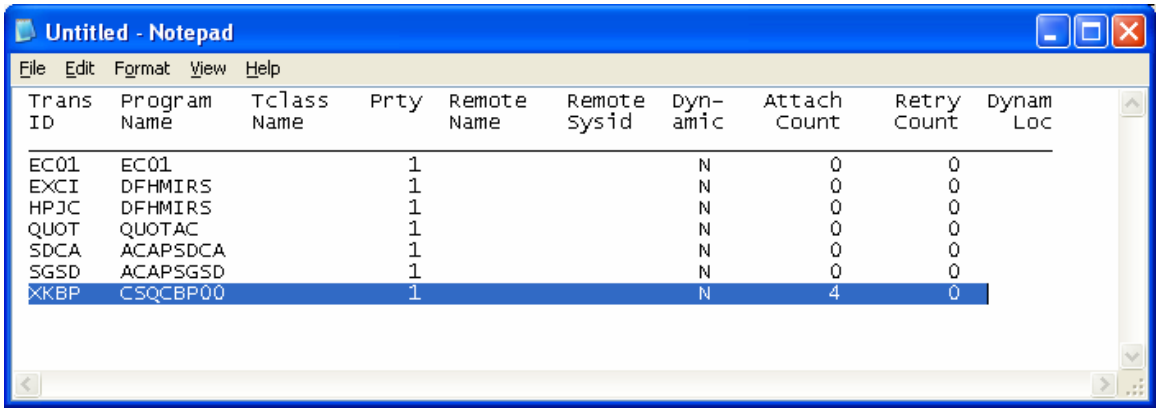
Save the change and re-deploy the application to the server. Now send several messages, using the “PUT_CIH” button.

In order to verify that the program ran under the new TransactionID, the STAT transaction in CICS can be used, as follows:

1. Start the STAT transaction

2. Press PF4 to go to the reports selection screen
3. You want to set all reports to N except "Transactions"
4. Press the Enter key, then PF3 to return to the main STAT screen
5. Generally, all the fields on this screen can be left at their defaults, but you can change the CLASS to a SYSOUT class appropriate for your system
6. Press the PF5 key to print the statistics report

You can now view the report on the SPOOL using SDSF. Scroll down to find your XKBP transaction & look to see if the Attach count (column 6) is equal to the number of messages which you sent. An example of that output is here:



Test Observations

- There seem to be no issues with CCSID or other translations between the platforms
- The button 'PUT_MQAPI' doesn't work unless the code is changed for the specific environment.
- Occasionally during test we received connections errors. We stopped and restarted the server to clean these up, but a better way to handle the error condition would be to enhance the code's exception handling would be to retry with new instances of the QueueConnectionFactory and QueueDestinations).
- If you update the trace spec with com.lot.bridge.*=all=enable (it becomes: *=info:com.lot.bridge.*=all), you will see the trace entries in the server log.

After you shut down the server, look in the profiles directory in the baseV6 server directory of the runtimes under RAD

Appendix A - MQCIH Structure in COBOL

The COBOL code can be found in the MQ target library **SCSQCOBC(CMQCIHL).**

We ftp'd the source to our development machine and modified it slightly, to add the 01 Level and make sure text starts in column 8. We named the file on our workstation **mqcih.cpy** so the JCA Wizard would process it as a code fragment, frequently referred to as a Copy Book.

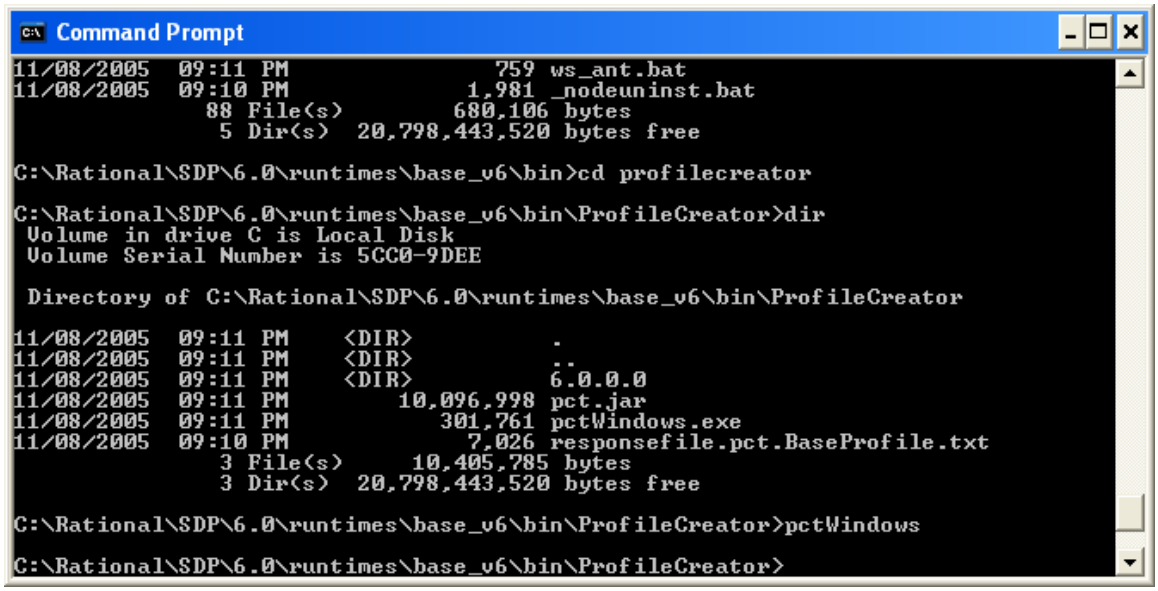
```
01  MQCIH.
    15  MQCIH-STRUCID PIC X(4).
    15  MQCIH-VERSION PIC S9(9) BINARY.
    15  MQCIH-STRUCLENGTH PIC S9(9) BINARY.
    15  MQCIH-ENCODING PIC S9(9) BINARY.
    15  MQCIH-CODEDCHARSETID PIC S9(9) BINARY.
    15  MQCIH-FORMAT PIC X(8).
    15  MQCIH-FLAGS PIC S9(9) BINARY.
    15  MQCIH-RETURNCODE PIC S9(9) BINARY.
    15  MQCIH-COMPCODE PIC S9(9) BINARY.
    15  MQCIH-REASON PIC S9(9) BINARY.
    15  MQCIH-UOWCONTROL PIC S9(9) BINARY.
    15  MQCIH-GETWAITINTERVAL PIC S9(9) BINARY.
    15  MQCIH-LINKTYPE PIC S9(9) BINARY.
    15  MQCIH-OUTPUTDATALENGTH PIC S9(9) BINARY.
    15  MQCIH-FACILITYKEEPTIME PIC S9(9) BINARY.
    15  MQCIH-ADSDESCRIPTOR PIC S9(9) BINARY.
    15  MQCIH-CONVERSATIONALTASK PIC S9(9) BINARY.
    15  MQCIH-TASKENDSTATUS PIC S9(9) BINARY.
    15  MQCIH-FACILITY PIC X(8).
    15  MQCIH-FUNCTION PIC X(4).
    15  MQCIH-ABENDCODE PIC X(4).
    15  MQCIH-AUTHENTICATOR PIC X(8).
    15  MQCIH-RESERVED1 PIC X(8).
    15  MQCIH-REPLYTOFORMAT PIC X(8).
    15  MQCIH-REMOTESYSID PIC X(4).
    15  MQCIH-REMOTETRANSID PIC X(4).
    15  MQCIH-TRANSACTIONID PIC X(4).
    15  MQCIH-FACILITYLIKE PIC X(4).
    15  MQCIH-ATTENTIONID PIC X(4).
    15  MQCIH-STARTCODE PIC X(4).
    15  MQCIH-CANCELCODE PIC X(4).
    15  MQCIH-NEXTTRANSACTIONID PIC X(4).
    15  MQCIH-RESERVED2 PIC X(8).
    15  MQCIH-RESERVED3 PIC X(8).
    15  MQCIH-CURSORPOSITION PIC S9(9) BINARY.
    15  MQCIH-ERROROFFSET PIC S9(9) BINARY.
    15  MQCIH-INPUTITEM PIC S9(9) BINARY.
    15  MQCIH-RESERVED4 PIC S9(9) BINARY.
```

Appendix B: Creating a New Server Profile for RAD

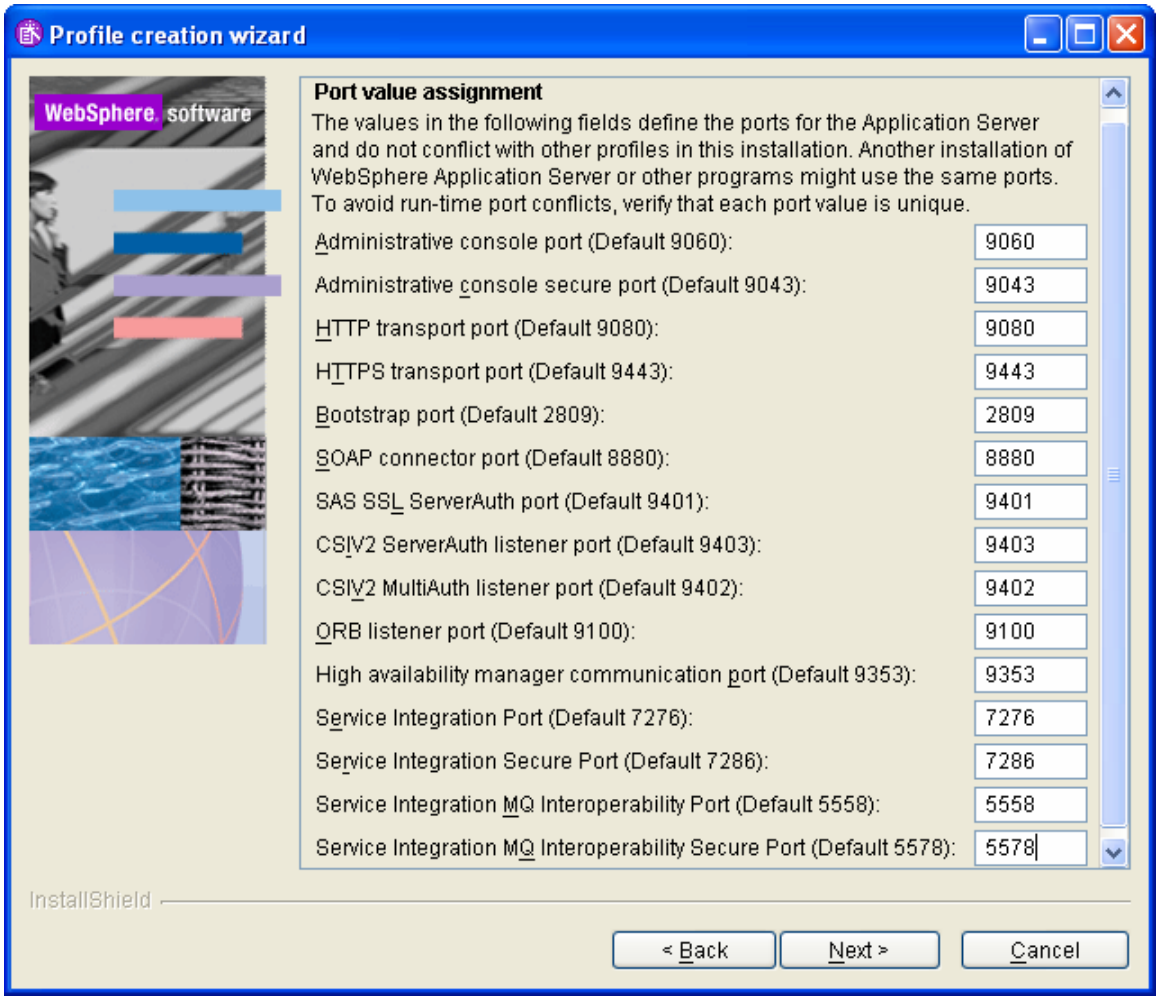
RAD can use the same server configuration across multiple workspaces, and in fact that is the default behavior. If you have a server configured for another project, and don't want to perturb it, you can create a new Server Profile, and use that in RAD when defining a new Test Server.

Create a New Server Profile

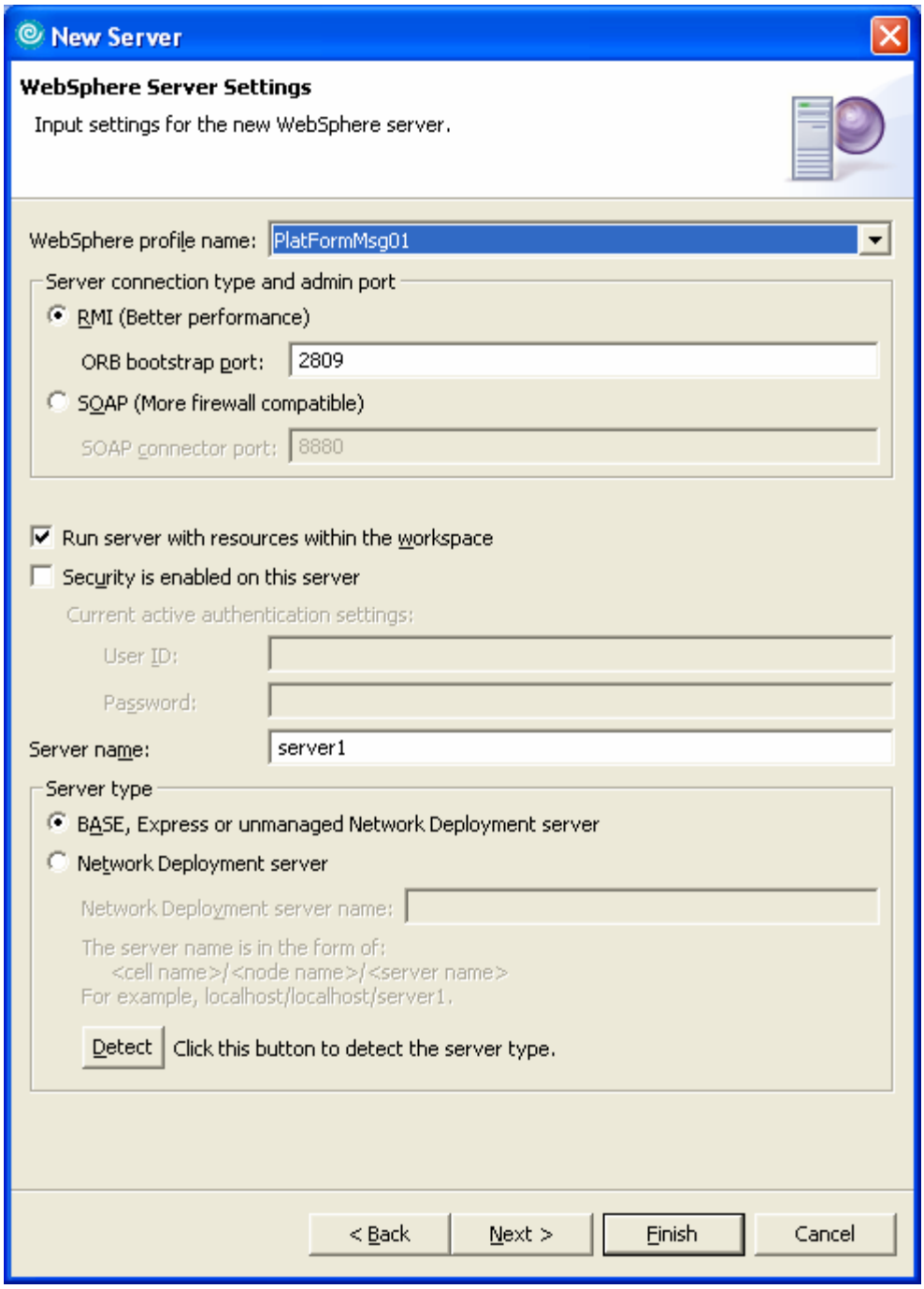
Use the ProfileCreator command in the bin directory of the V6 runtime:



Even though this is new server, we will not have it running concurrently with others we may have defined and configured, a=so for convenience, we want to use all the default ports. You will have to change them back to match those in your other test servers.



Once this profile is defined, go to RAD and create a new server which uses it.


New Server
WebSphere Server Settings
Input settings for the new WebSphere server.
WebSphere profile name:
PlatFormMsg01
Server connection type and admin port
RMI (Better performance)
ORB bootstrap port:
2809
SOAP (More firewall compatible)
SOAP connector port:
8880
Run server with resources within the workspace
Security is enabled on this server
Current active authentication settings:
User ID:
Password:
Server name:
server1
Server type
BASE, Express or unmanaged Network Deployment server
Network Deployment server
Network Deployment server name:
The server name is in the form of:
<cell name>/<node name>/<server name>
For example, localhost/localhost/server1.
Detect
Click this button to detect the server type.
< Back
Next >
Finish
Cancel

Appendix C: References

The WebSphere MQ and CICS/TS books were consulted frequently.

The WMQ InfoCenter can be obtained at the following URL:

http://www-306.ibm.com/software/integration/wmq/library/infocenter/

The CICS Library is available at:

http://www-306.ibm.com/software/htp/cics/library/