WebSphere Application Server V4.01 for zOS and OS/390

Overcoming a Problem Running Simulated Message Driven Beans

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP100301** under the category of "White Papers"

Version Date: August 26, 2002

IBM Washington Systems Center

Don Bagwell
IBM Washington Systems Center
301-240-3016
dbagwell@us.ibm.com

Many thanks go to **Mike Cox** of the IBM Washington Systems Center for being the driving force behind producing this white paper in the first place, and providing the technical guidance during development.

The author wishes to thank **Renuka Chekkala** of IBM for providing the source code for the sample application. The source was modified in a few small ways after receiving it from Renuka, so any comments regarding the application itself should be directed to the author of this white paper.

The author would also like to thank **Vaughn Burton** of the IBM Washington Systems Center for his help setting up MQSeries for the development of this white paper.

Table of Contents

(This page intentionally left blank)

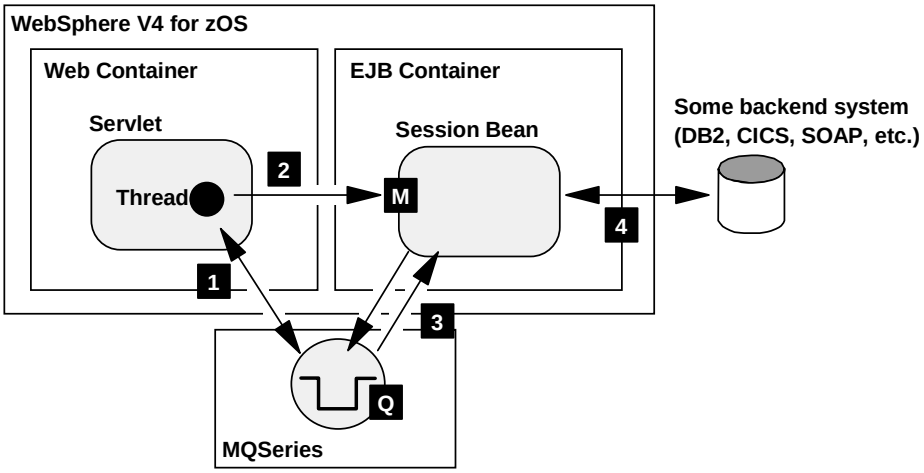
Overview

We have seen a number of cases where customers attempt to construct an test application that looks like a message driven bean (MDB). The full MDB support is not yet in WebSphere for zOS, and many customers are looking to gain experience with the technique by coding up a "pseudo" implementation.

Unfortunately, these customers are running into problems when their servlet clients attempt to drive the session bean. The basic problem is that a servlet cannot create a client thread which accesses managed objects. But there is a workaround, *and it involves moving the servlet client from the WebSphere "web container" environment to the WebSphere "Plugin" environment*, where the invocation of the `getMessage()` method of the session bean will be a remote call.

Background

What these customers are attempting to create is something that looks like this:



"Pseudo" implementation of message driven bean

Here's what this picture is illustrating:

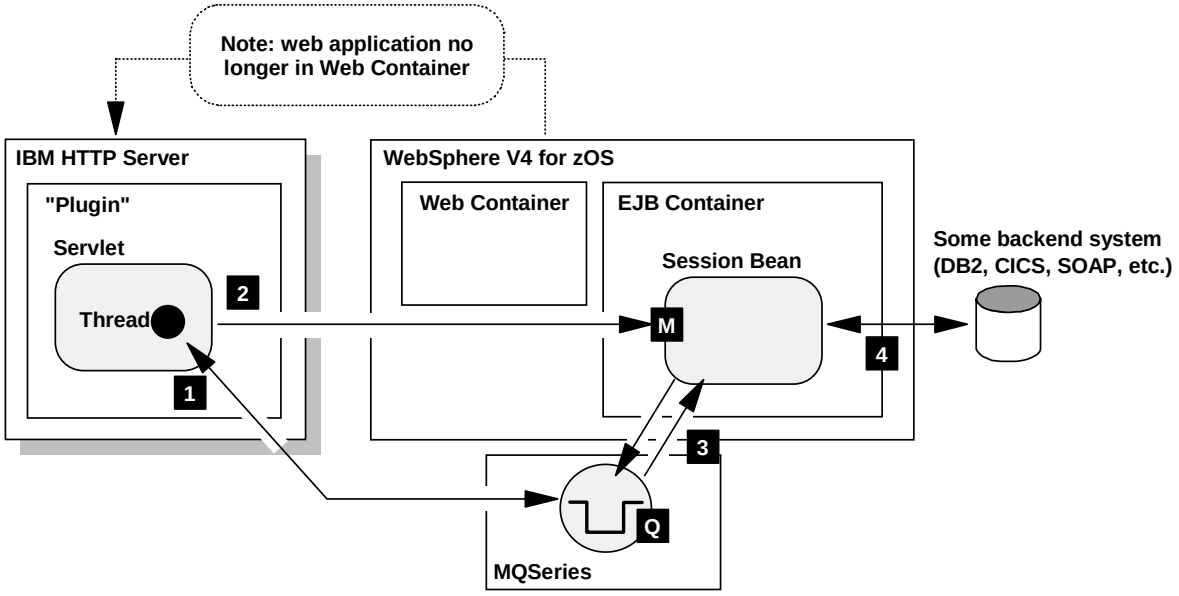
- An auto-started servlet which creates a number of threads, usually one thread for each inbound message queue being monitored. This servlet provides some management functions, such as the create/delete of a queue-monitoring thread on demand and provide some information on usage.
- The thread monitors **[1]** the inbound message queue **[Q]**, and if a message is present it copies the correlation ID and then invokes **[2]** the `getMessage()` method **[M]** of the session bean.
- The session bean then uses the correlation ID passed it and then processes **[3]** the message on the queue. Typically, the session bean then accesses a backend system **[4]**, but what and how it accesses that system is not relevant to the problem being discussed in this paper.
- The bean then returns to the client, and the process starts all over again

This is what the customers are *attempting* to do. Unfortunately, it won't work because the servlet can't create a thread that accesses managed objects.

The Solution

Overview

The solution to this problem involves moving the servlet from the web container environment to the "Plugin" environment, which runs in the IBM HTTP Server and provides a servlet execution environment:

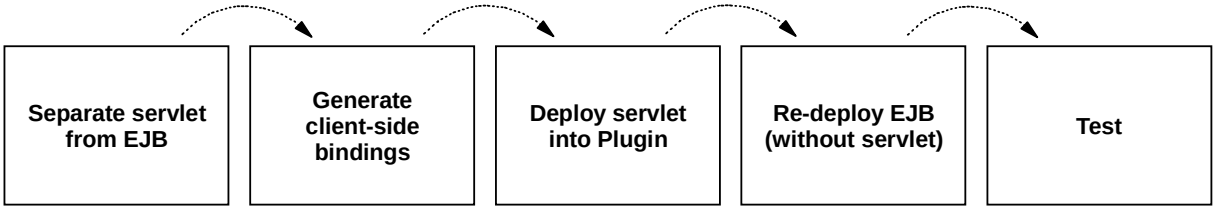


Servlet moved to Plugin environment -- provides remote call, which allows this to work

This works because the servlet thread is now remote from the managed object, and can therefore make the call.

The steps needed to make this work

The basic flow of this process is this:



Flow of steps necessary to allow servlet to call session bean

These steps are discussed in a bit more detail next.

Note: For a far more detailed look at this, see the supplied sample code and the process to deploy it discussed under "Sample Code" starting on page 8.

Separate servlet from EJB

If you're presently deploying the whole application (servlet and EJB) into the WebSphere V4 runtime, the web application (servlet) will be packaged inside the EAR file in the form of a WAR file. That WAR file needs to come out of the EAR so you can deploy the web application into the Plugin environment. To get ahold of *just* the WAR file, you have several options:

- Go back to the development tool (WSAD, for example) and export the WAR file separately

  or

- Go back into AAT, select the web application and export just the WAR file

  or

- Use a tool such as WinZIP to extract the WAR file from the EAR file

All do the same thing: put a WAR file onto the hard drive of your workstation.

Note: Later you will need to *redeploy* the EAR file into the WebSphere V4 runtime environment without the web application as part of the application. Therefore, you may be best served by going into the AAT, exporting the WAR, then deleting the WAR from the EAR.

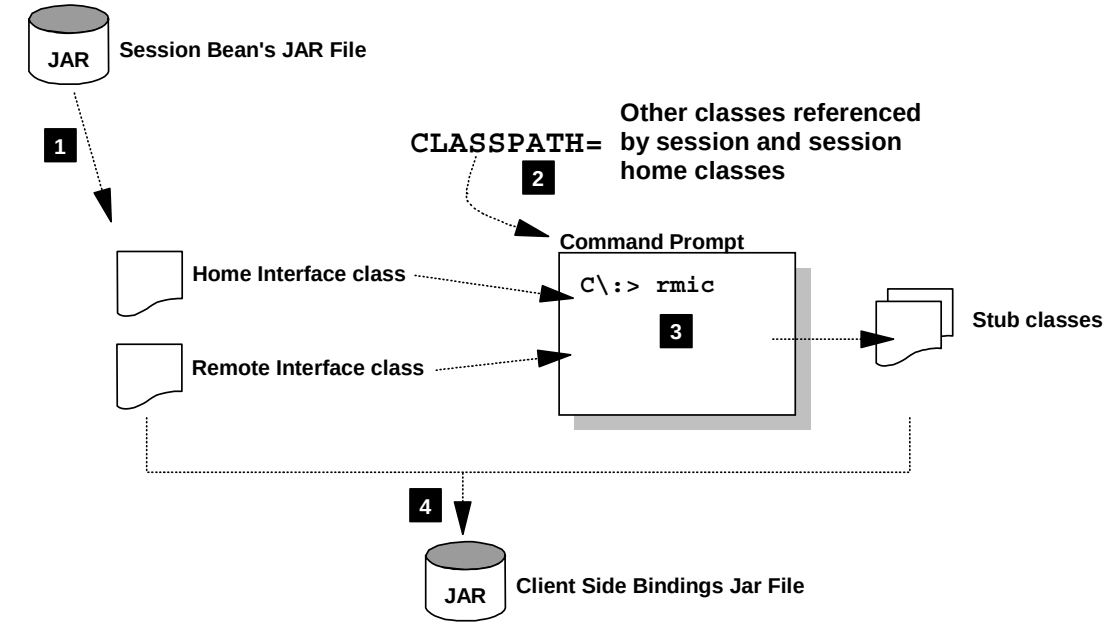
With WAR file in hand, you are *almost ready* deploy the web application into the Plugin.

Generate Client-Side Bindings

In order for a remote client remote to be able to access an EJB in the WebSphere V4 runtime, the client must have access to "client side binding" code. When the client code executes, it will need access to the bindings, and it will look for the binding code in the CLASSPATH.

If your application was developed using VisualAge for Java, that tool provides a handy wizard to easily generate the bindings. All you need to do is select the session bean, right-mouse-click and then select "Export ⇨ Client Jar..." VisualAge will then generate the necessary files and package them into a JAR file.

If your application was developed using WebSphere Studio Application Developer (WSAD), then you have a little more work to do. That tool does not provide a wizard to create the client side bindings. But it can still be done. The high-level process looks like this:



Overview of the manual client-side bindings creation process when using WSAD

Here's what the numbered blocks refer to:

1. The session bean's Remote Interface Class and Home Class are extracted from the bean's JAR file and placed in a directory on your workstation hard drive. You can do that by using the jar command. *For example*:

```
jar -xf PolicySession.jar com\ibm\ws390\samples\ivp\ejb\PolicySession.class
```

and

```
jar -xf PolicySession.jar com\ibm\ws390\samples\ivp\ejb\PolicySessionHome.class
```

Note: This example is illustrating creating the client side bindings for the sample IVP that comes with WebSphere called "PolicyIVP." In the section of this white paper titled "Sample Code" on page 8, you will use different file names.

1. The `CLASSPATH` for a command prompt session is updated to point to any class files referenced by the remote interface class or home interface class. There's a handful of common class files that will *always* be referenced by the "rmic" process:

   ```
   javax.ejb.EJBObject
   javax.ejb.EJBHome
   java.rmi.RemoteException
   javax.ejb.CreateException
   ```

   These are found in the JAR file `j2ee.jar` which is supplied with WSAD. Update your command prompt `CLASSPATH` with the following:

   ```
   (WSAD install path)\plugins\com.ibm.etools.websphere.runtime\lib\j2ee.jar
   ```

   There may be *other* classes referenced by the home and remote interface classes. For example, the "PolicyIVP" application that comes with WebSphere would also require a pointer to the `PolicyUtil.jar` file so it could access the class file:

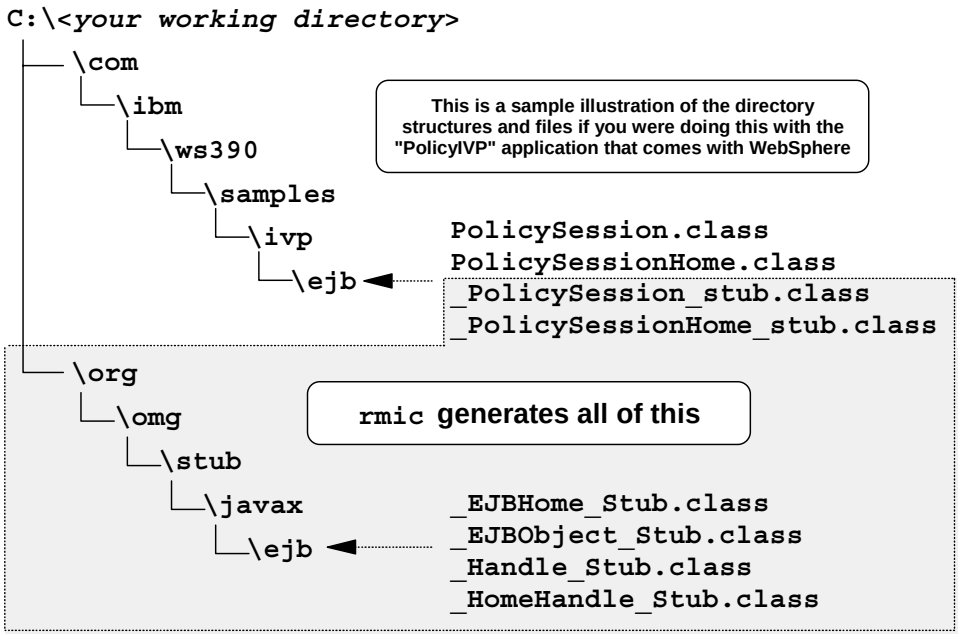
   ```
   com.ibm.ws390.samples.ivp.utilities.IVPVerificationException
   ```

   Your session bean might have different or other class files. The point is, make sure the command prompt's `CLASSPATH` points to all the classes being referenced.

2. Next comes the task of running the **`rmic`** utility to create the stubs. (The `rmic` utility comes with the JDK, much like the `jar` and `javac` commands.). In the working directory into which you extracted the two files in Step 1, issue the following commands:

   ```
   rmic -iiop -always -d . <package name of remote interface class>
   rmic -iiop -always -d . <package name of home interface class>
   ```

   This will result in the creation of six files in addition to the two you extracted in Step 1:



What the "rmic" utility produces when you invoke it against the Home and Remote Interfaces

3. Finally, all eight of the files illustrated in the previous picture need to be bundled up into a JAR file. Using the previous picture as an example, the command to bundle the two package structures up would be:

   ```
   jar -cf PolicySessionClientBindings.jar com org
   ```

   Issue this command from the directory in which the session home and remote interface packages start. In the example above, that would be the "working" directory. The packages ("com.ibm..." and "org.omg...") begin in that directory.

This JAR file must be accessible on the `CLASSPATH` of the client, which in this scenario is the servlet you're moving from the web container to the Plugin environment. Therefore, when you deploy the servlet into the Plugin environment, make sure to copy this "Client Side Binding" JAR file into the `CLASSPATH` as well.

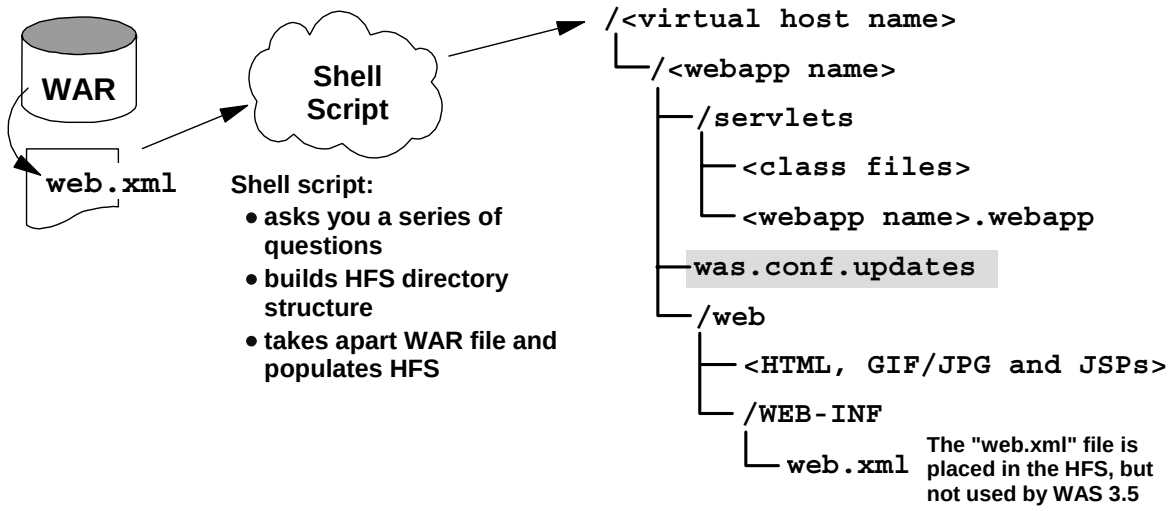
Deploy Servlet in Plugin

Unfortunately, the Plugin doesn't support deploying WAR files directly. Therefore, to deploy a WAR file into the HTTP Server WebSphere plugin involves using the `wartowebapp.sh` utility, which is supplied with the Plugin. This utility does several things:

- It takes the WAR file apart and populates the HFS with the various piece-parts
- It creates a file called `was.conf.updates`, which includes the deployedwebapp statements for this application that you simply cut-and-paste into the `was.conf` file
- It takes the contents of the `web.xml` file (the deployment descriptor inside the WAR file) and converts it into the format of XML file the Plugin understands, which is *close* to the same format as `web.xml`, but not exactly.

Note: It's the last bullet that makes using `wartowebapp.sh` worthwhile. You might be successful taking the WAR file apart by hand, and you would probably get the `deployedwebapp` statements built properly, but converting the `web.xml` file into the proper "dot-webapp" file would be unlikely.

Here's a picture illustrating what the `wartowebapp.sh` utility does:



What the "wartowebapp.sh" utility will do with a WAR file

Note 1: Consult the white paper **WP100238** on `www.ibm.com/support/techdocs` for a more complete explanation of how to configure the Plugin, and how it handles requests and maps them to definitions in the `was.conf`.

Note 2: Don't forget to place the "Client Side Bindings" JAR file in one of the `CLASSPATH` directories. Your webapp will need that to access the session bean.

See "Deploy WAR into Plugin using wartowebapp.sh" on page 16 for an illustration of how to run the `wartowebapp.sh` utility.

Redeploy EJB (without servlet)

Your objective in this step is to remove the webapp from the web container environment, and to do that you must re-deploy the application EAR file (minus the webapp) back into the J2EE application server in the WebSphere V4 runtime.

You can remove the webapp from the EAR by using the AAT tool. Start the AAT, mark the web application, and then delete it. Then validate, deploy and export the EAR. That will result in the a new EAR file being created, minus the webapp. From there, you redeploy the EAR back into the same J2EE application server. WebSphere will overlay the existing application code with the new, and thus remove the webapp from the web container.

??? Is it possible to leave the WAR in the EAR? Technically speaking, yes. But it can get confusing if you have the same webapp deployed in both the Plugin and the web container. If the Plugin sees the same "virtual host + context root" pair in both itself and the web container, it'll send the request to the web container. Therefore, you may end up invoking the webapp in the web container and not realize it. That would put you right back to square one of this problem. It's better to remove the webapp from the web container. It's less confusing. That's done by removing the webapp from the EAR and redeploying the EAR into the server.

If you feel you can't remove the webapp from the EAR, you have two alternatives:

- Prevent the web application from binding to a virtual host in the web container. That's controlled in the `webcontainer.conf` file. Without it being properly bound, it can't be invoked. That'll insure the webapp in the *Plugin* takes precedence. You can prevent a web application from binding to a virtual host by removing whatever `contextroots=` string from the `webcontainer.conf` that this webapp's context root matches to. If you are using the "catch-all" single slash `contextroots=` value, that means you probably can't avoid having this webapp bind (removing the single slash would mean *no* webapps would bind).

- Make sure that the `rooturi=` value in the `was.conf` is different from the `contextroot=` setting for the webapp in the web container. The confusion comes in when the `rooturi=` value of a webapp in the Plugin is the same as the `contextroot=` for a webapp in the web container, and they both have the same virtual host value. If you make those two different, then there's no ambiguity and you can insure the webapp in the Plugin takes precedence.

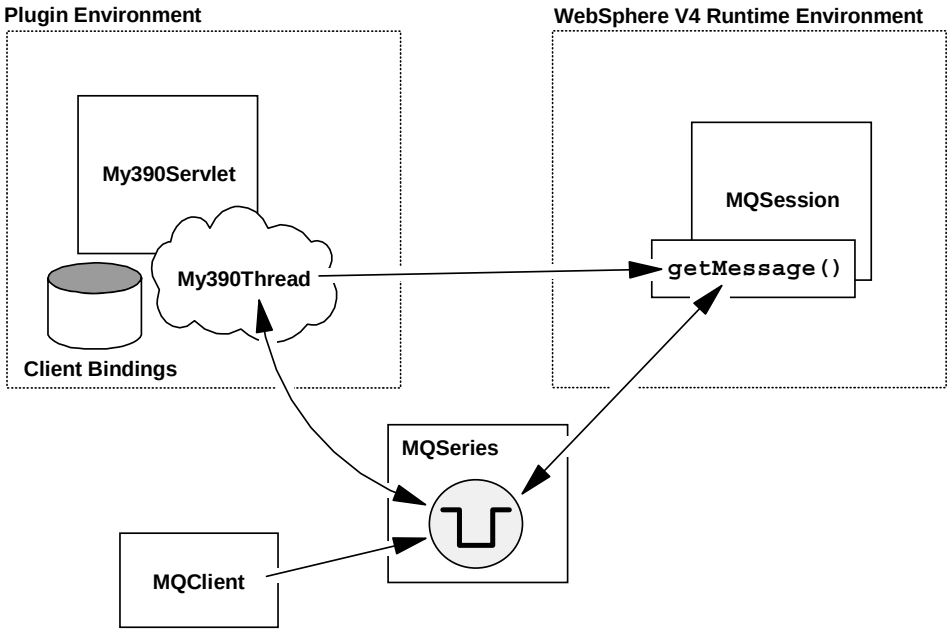
Test

The final step is, of course, to test your newly deployed application.

Sample Code

Note: This application is delivered "as is" and is intended to be simply a sample.

What will be described here is a very simple application that can be used to illustrate the points made earlier in this paper. The sample application looks like this:



The sample application

Note: This application was designed to illustrate the process shown above without being overly complicated. It does not have a lot of error handling function (but enough to get by). Also, the behavior of this application on the browser screen is pretty sparse. So don't think this is going to be an exciting browser experience, because it won't be. But it will do what it's supposed to do.

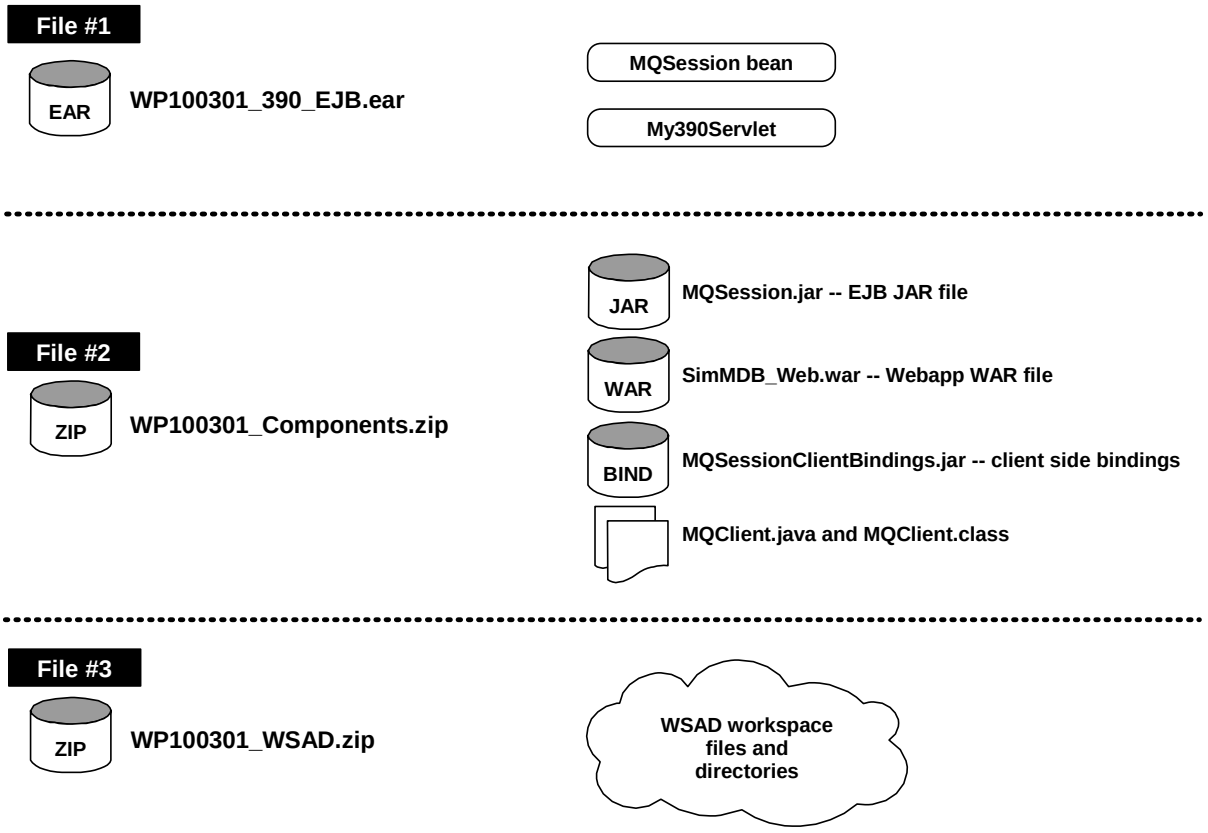
Each piece of the application is described here:

- **My390Servlet** -- a relatively simple servlet that receives the HTTP request and then invokes the "My390Thread" to do the actual processing. My390Servlet will get the values for the MQSeries Queue Manager, the Queue Name and the JNDI name of the session bean from the initialization parameters set for the servlet. Those parameters are set in the `web.xml` file.
- **My390Thread** -- this code will receive from My390Servlet the name of the Queue Manager and Queue Name and then go check MQSeries to see if there are any messages on the queue. If so, it'll take the JNDI name of the session bean that My390Servlet passed in and then do a lookup on the MQSession session bean and once found, drive the `getMessage()` method of the session bean.
- **Client Bindings** -- this is the code needed by the client to invoke the session bean when the client is in a remote environment. Because the servlet is running in the Plugin and not the web container, it needs this to properly drive the session bean.
- **MQSession** -- this stateless session bean will pull messages off the MQ queue when the servlet (or more specifically, when My390Thread) invokes the `getMessage()` method of the session bean. The session bean also has a putMessage() method, but My390Thread doesn't have any code to drive that method.

- **MQClient** -- this is a simple standalone Java program that will fill a specified MQ queue with messages. Use this to "load" the queue prior to running the My390Servlet.
- **MQSeries** -- for this to work, you must have MQSeries installed. You will need to have at the ready the Queue Manager name and the Queue name.

How the sample application is packaged for your use

This application has been packaged in three different ways for your use:



Three different files containing the application in different formats

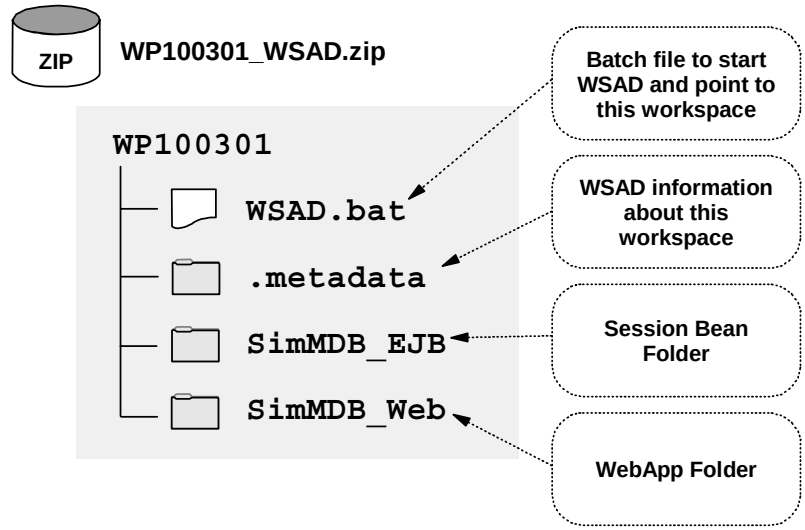
- **WP100301_390_EJB.ear** -- This is an EAR file as produced by the 390 AAT tool. If you wish to deploy this sample application by taking an assembled EAR file apart, use this file.
- **WP100301_Components.zip** -- This is a ZIP file that contains the piece-parts to the application. If you don't want to both taking an EAR file apart, you can simply unzip this file and you'll have the pieces ready to go.
- **WP100301_WSAD.zip** -- This is a zipped-up WSAD workspace. If you unzip this onto your workstation hard drive and then invoke WSAD and point to this directory, you'll have the WSAD environment ready to go.

Note: For the rest of this paper it will be assumed you will use the WSAD workspace. You'll also need access to the `WP100301_Components.zip` file to get the Java source for the MQClient program.

Step by step: from downloaded files to working application

Unzip file onto workstation

The file `WP100301_WSAD.zip` has an internal structure that looks like this:



Internal structure of WSAD workspace zip file

By unzipping this, you'll have a WSAD workspace with nothing but this project in it.

Do the following:

- ☐ Unzip the file `WP100301_WSAD.zip` into the `C:\` directory on your workstation. It will create the directory `C:\WP100301`. Be sure that all sub-directories from the ZIP file are created as well (it should default to this behavior).

Modify WSAD.bat file

??? What's this all about? Using a batch file to start WSAD and point to a separate repository is a nice way of keeping a project cleanly separated from other projects. For anyone who has had to pause while working in WSAD to get things straight again in their mind, this will make sense. For those new to WSAD ... just trust that this will makes thing easier.

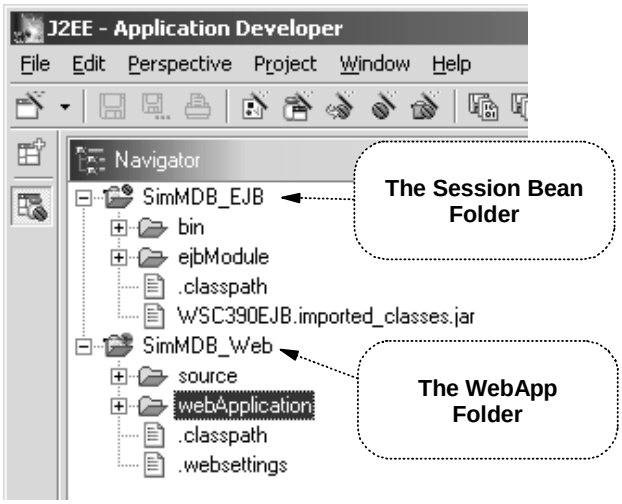
There should be a file called `C:\WP100301\WSAD.bat` on your workstation. It would have come out of the `WP100301_WSAD.zip` file.

- ☐ Edit that file and see if the pointer to the executable `wsappdev.exe` is correct (this is where WSAD would be installed on your workstation). Correct *if necessary*.

```
Rem
Rem start WSAD for this project.
Rem
"C:\Program Files\IBM\Application Developer\wsappdev.exe" -data c:\WP100301
```

When this batch file is run, it will start WSAD (`wsappdev.exe`) and use the separate directory of `C:\WP100301` as the place it'll look for its workspace.

- ☐ Invoke the batch file and start WSAD. You should see a structure that looks something like this:



Initial appearance of SimMDB projects in WSAD

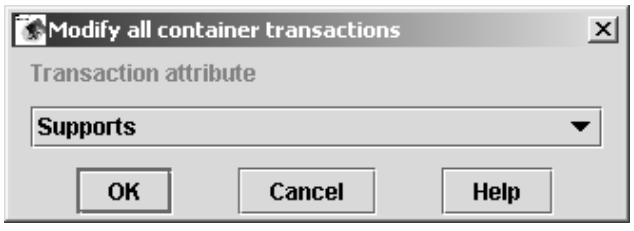
Create EJB JAR file

- ☐ Create a work directory on your PC with a name of `C:\WP100301_work`
- ☐ Select the SimMDB_EJB folder in WSAD, right click and select "Export EJB Jar."
- ☐ Put the JAR out as `C:\WP100301_work\MQSession.jar`

> **Note:** Or you can simply extract the file `MQSession.jar` from the `WP100301_Components.zip` file or the `SimMDB_390_EJB.ear` file supplied with this white paper.

Assemble EJB JAR into EAR file using AAT

- ☐ Start the WebSphere for zOS "Application Assembly Tool" (AAT)
- ☐ Create a new application called SimMDB and import `MQSession.jar` into the "Session Beans" folder
- ☐ Locate the session bean "MQSession," right-click and select "Modify"
- ☐ Click on the "Transactions" tab to show the two container transactions: `getMessage` and `putMessage`.
- ☐ Click on the "Modify All..." button and then set the transaction attribute for both to "Supports":



Set "Transaction Attribute" for both getMessage and putMessage to "Supports"

> **Why?** The bean coming out of WSAD didn't have the transaction attributes set. Unless you set these values, AAT will not allow the bean to be deployed. A value of "Supports" means "The method runs in a transaction if a transaction is already running; otherwise, it runs with no transaction."

- ☐ Save the changes

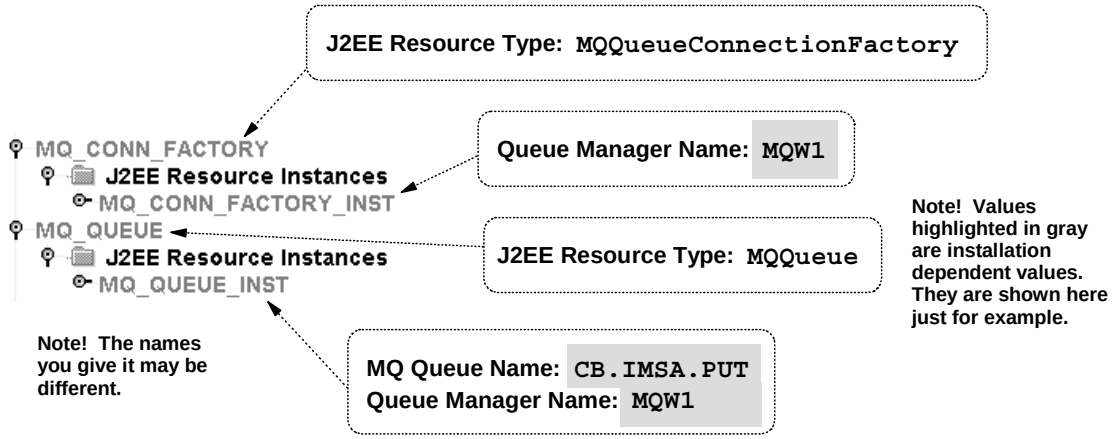
- ☐ Now select the "SimMDB" application, right click and select "Validate"
- ☐ Make sure the "Options" pulldown selection "EJB Deploy" is unchecked (you do not want AAT to regenerate all the classes)
- ☐ Select "SimMDB" again, right click and then "Deploy"
- ☐ Select "SimMDB" one last time, right click and select "Export" and put the application out as an EAR file at location `C:\WP100301_work\WP100301_390_EAR.ear`

Note: Or you can simply use the file `WP100301_390_EAR.ear` supplied with this white paper.

You now have an EAR file ready to be deployed to the WebSphere V4 for zOS runtime. But first you must set up some MQ resources so the application can be properly deployed.

Create J2EE Resources for MQSeries resources

- ☐ Log onto the WebSphere V4 Systems Management administrative console (the "SMSEUI" tool) and create a new conversation.
- ☐ Create two J2EE resources with an resource instance under each. The SMSEUI tool will set most of the values based on the selection you make for "J2EE Resource Type". The following picture illustrates how you should set the values:



Values for J2EE Resources for MQSeries Queue Manager and Queue

Note 1: This assumes you have MQSeries installed on your system, and WebSphere for zOS has been made aware of the new resource. If you don't see "MQQueueConnectionFactory" in the SMS EUI pulldown menu, then MQ hasn't been installed into WebSphere. This paper will not detail how to do that.

Note 2: The values you supply for the "Resource" and "Resource Instance" are arbitrary ... you may name them anything you like. When you go to deploy the EAR file, the names you give them -- whatever those names happen to be -- will appear in the pulldown list when you tie the symbolic resource reference for the bean to the actual resource. For the sake of this illustration, the names are as shown in the picture above.

Note 3: The name of your MQSeries Queue Manager and Queue will no doubt be different from what we're showing in this example. The key point is the Queue Manager must exist, and the Queue must exist. The far more important values on the chart above are the "J2EE Resource Type" values shown. Those have to be set as shown in the chart because those are the values in the bean's deployment descriptors. If you provide different values (and there are some other values available that are close, but not exactly the same as those shown here), then when you go to deploy the bean you won't find any pulldown values available when you tie symbolic reference to actual resource.

- ☐ Save changes, but don't commit the conversation just yet ...

Provide MQSeries library updates to CLASSPATH and LIBPATH for J2EE server

- ☐ For the J2EE server into which you plan to deploy the session bean, modify the server definition, then update the Environment Variable `LIBPATH` and concatenate the MQSeries Java lib directory (separate from previous with a colon):

  ```
  /usr/lpp/mqm/java/lib
  ```

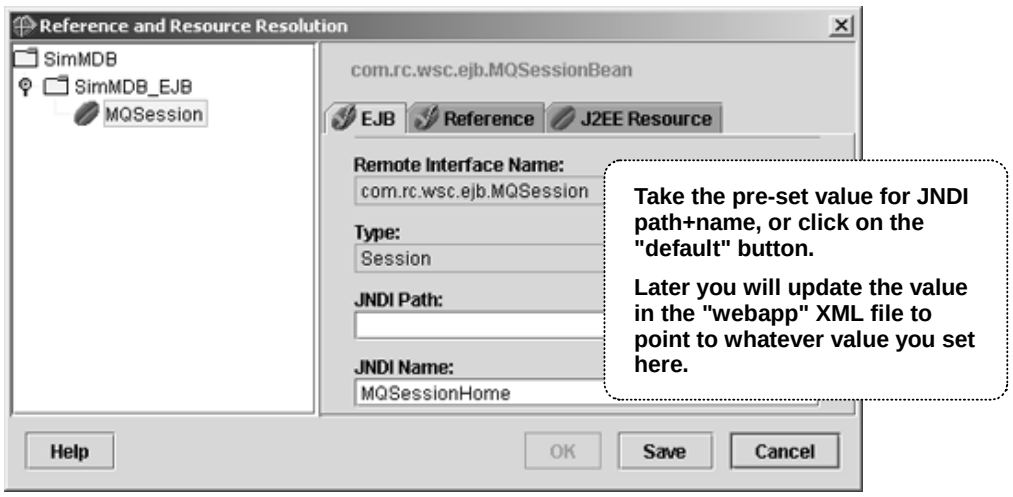
- ☐ Concatenate the `CLASSPATH` variable with the following references (separate each with a colon):

  ```
  /usr/lpp/mqm/java/lib/com.ibm.mqjms.jar
  /usr/lpp/mqm/java/lib/com.ibm.mq.jar
  ```

Deploy MQSession EAR into J2EE server

Note: This white paper assumes the J2EE Server has been created and validated. If you're not certain how to do that, you may wish to consult the white paper `WP100277` at `www.ibm.com/support/techdocs`. In that paper, under the section titled "Phase 3: Create Application Server" the steps needed to create a J2EE application server are outlined.

- ☐ Select the J2EE server into which you will install the application, then right-click and select "Deploy J2EE Application."
- ☐ Select the file `C:\WP100301_work\SimMDB_390_EJB.ear`
- ☐ Set the JNDI path and name:



Setting JNDI path+name for the MQSession bean

- ☐ Set the J2EE Resources by clicking on that tab:

J2EE Resource pulldowns should have your Resource and Resource Instance values

> **Note:** If you don't see a value in the pulldown list, that means the J2EE Resource or Resource Instance you created a few steps back had a different "J2EE Resource Type" value than what's in the bean's deployment descriptor. You may need to go back and redefine your resources.

- ☐ Click on OK to initiate the transfer and installation of the EAR file
- ☐ Save all changes, then validate, commit and activate the conversation
- ☐ Start the J2EE application server (if WebSphere didn't re-start it automatically) and insure that "naming registration" was successful and that the session bean is properly registered in LDAP.

Create client-side bindings file

WSAD does not provide a wizard to produce the client side bindings like VisualAge for Java did. Therefore, the process is manual:

> **Note:** You may follow these steps to create your own Client Side Bindings file, or you can simply extract the file `MQSessionClientBindings.jar` from the `WP100301_Components.zip` file supplied with this white paper.

- ☐ Create the directory `C:\WP100301_rmic` on your workstation
- ☐ Copy the `MQSession.jar` file supplied with the `WP100301_Components.zip` file into the `C:\WP100301_rmic` working directory

> **Note:** If you prefer, you may create your own EJB JAR file from WSAD. Highlight the `SimMDB_EJB` project folder, right click and select "Export EJB Jar." Put the file out as `C:\WP100301_rmic\MQSession.jar` and it's the same thing as copying the file provided in the "components" zip file.

- ☐ Go to a command prompt and change directories to `C:\WP100301_rmic`
- ☐ Extract the MQSession.class and MQSessionHome.class files from the ZIP, maintaining its package structure, with the following commands:

```
jar -xf MQSession.jar com\rc\wsc\ejb\MQSession.class
jar -xf MQSession.jar com\rc\wsc\ejb\MQSessionHome.class
```

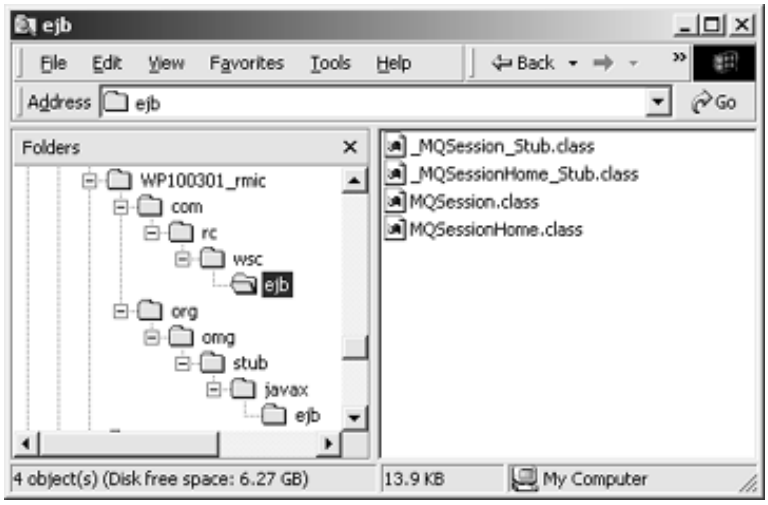
- ☐ Set the `CLASSPATH` variable to point to the directory containing the class files that "rmic" will need (issue this as <u>one</u> command):

```
set CLASSPATH=%CLASSPATH%;C:\Program Files\IBM\Application Developer
   \plugins\com.ibm.etools.websphere.runtime\lib\j2ee.jar
```

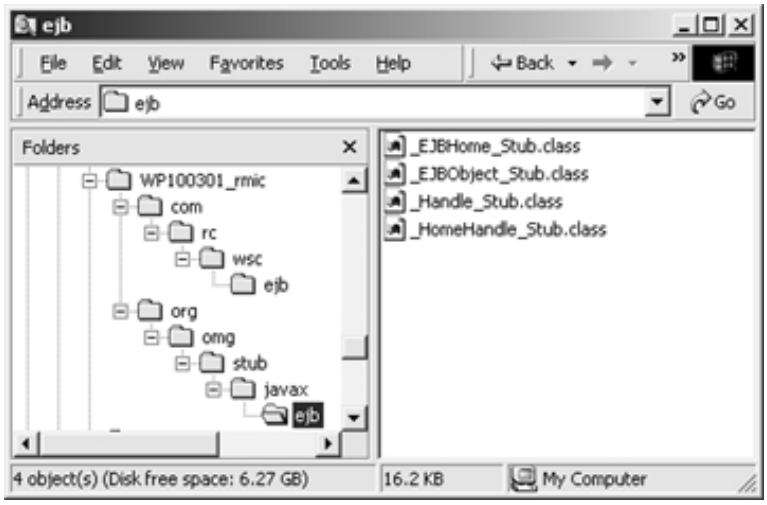
- ☐ From the `C:\WP100301_rmic` directory, Issue the following "rmic" commands to generate the client-side bindings:

```
rmic -iiop -always -d . com.rc.wsc.ejb.MQSession
rmic -iiop -always -d . com.rc.wsc.ejb.MQSessionHome
```

Your directory structure should now look like this:



Four files in the com.rc.wsc.ejb folder



Four files in the org.omg.stub.javax.ejb folder

- ☐ The final task is to "jar" both of these directory structures up into a single client-side bindings file. In the command prompt go to the `C:\WP100301_rmic` working directory and issue the following command:

  ```
  jar -cf MQSessionClientBindings.jar com org
  ```

  A bit later you see instructions on how to FTP the file up to the 390 box and put it in the directory named on the `deployedwebapp.classpath` directive in the `was.conf`.

Create WAR file

Note: You may follow these steps to create your own WAR file, or you can simply extract the file `SimMDB_Web.war` from the `WP100301_Components.zip` file or the `SimMDB_390_EJB.ear` file supplied with this white paper.

- ☐ Go back to WSAD
- ☐ Select the SimMDB_Web folder, right click and select "Export War."
- ☐ Put the JAR out as `C:\WP100301_work\SimMDB_Web.war`

Deploy WAR into Plugin using wartowebapp.sh

- ☐ FTP the `SimMDB_Web.war` file to the S/390 server in *binary* mode. Assume you put that into the directory `/u/user1`.
- ☐ Log onto TSO and make sure you have a sufficiently large logon "SIZE" value. This utility consumes a bunch of memory. For example, `2096128` works.
- ☐ Create a "temp" directory somewhere; let's say `/u/user1/tmp`, and give it permissions `777`
- ☐ Create a directory into which the WAR file contents will be place: `/u/user1/classes` and give it permissions `755`
- ☐ Go into OMVS and set the `WAS_HOME` environment variable to:

  ```
  export WAS_HOME=/usr/lpp/WebSphere401/WebServerPlugIn
  ```

- ☐ Change to the `/usr/lpp/WebSphere401/WebServerPlugIn`**`/bin`** directory and issue the following command:

```
./wartowebapp.sh WAR_FILENAME=/u/user1/SimMDB_Web.war TEMP_DIRECTORY=/u/user1/tmp
```

- ☐ The utility will now ask you a series of questions:

  `VIRTUAL_HOST_NAME` `<null to accept: default_host>`

  This is asking what virtual host defined in the `was.conf` you wish to use for this web application. If you use the default `default_host`, which by default maps to the value `localhost`, then just hit enter. Otherwise, type in the virtual host name you want this webapp to map to.

  `WEBAPP_NAME` `<null to accept: SimMDB_Web>`

  This will be used to determine the value for `<name>` in the `deployedwebapp` statement format of:

  ```
  deployedwebapp.<name>.classpath=
  ```

  It can be anything you like (no spaces or periods allowed). By default it takes the name of the WAR file.

  `WEBAPP_DESTINATION` `<null to accept: $was_install_root$/Web...`

  This is key. It is asking where you want the contents of the WAR file to be placed. By default the contents are going to get buried deep down the installation directory of the WebSphere Plugin. Instead, point it back to your `/u/user1/classes` (or where ever you've put your files) directory.

  `WEBAPP_AUTO_RELOAD_INTERVAL` `<null to accept: 0>`

  Provide a value, or accept default value of 0.

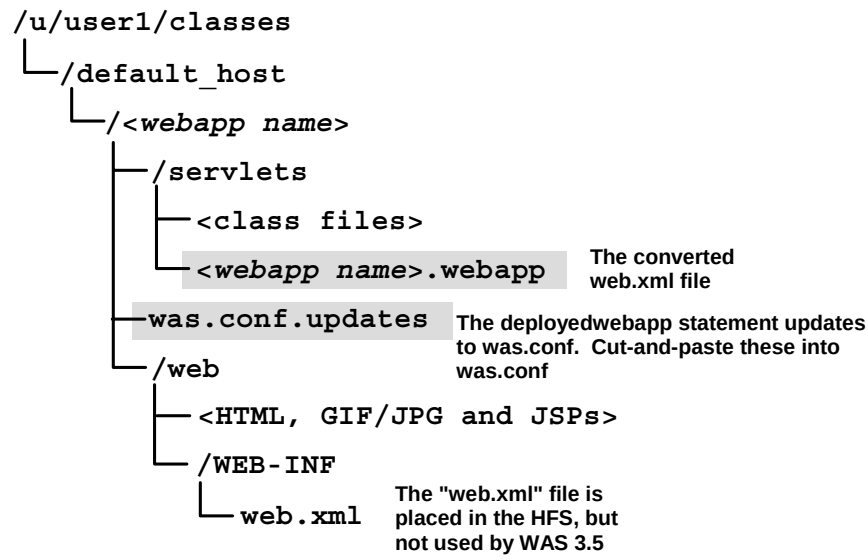
  `WEBAPP_PATH` `<null to accept: /webapp/SimMDB_Web>`

  This will be the `rooturi=` value. It will also be what you use on your `Service` statement. Set this to `/SimMDB_Web` (in other words, remove the extraneous `/webapp` from the default).

  `LOCAL_FILE_ENCODING` `<null to accept: en_US.IBM-1047>`

  Hit enter to accept the default, or supply a value if you know what you're doing.

If all works, you should see a message saying `BUILD SUCCESSFUL`, and you will have a structure that looks *something* like this in your HFS:



Update httpd.conf with new Service statement

- ☐ You now need to create a `Service` statement in the httpd.conf that will serve to map a URL from your browser over to the Plugin. That `Service` statement should be made equal to whatever value you provided to the `WEBAPP_PATH` question from the `wartowebapp.sh` utility. In this example, we set that to `SimMDB_Web`, so your `Service` statement should be:

  ```
  Service  /SimMDB_Web/*  /usr/lpp/WebSphere...
  ```

- ☐ Save the file

(Optional) Update WebSphere V3.5 to support connection to WebSphere V4

Note: This is only necessary if the Plugin that you're using is the older WebSphere V3.5. If you're using the Plugin that comes with WebSphere V4, you don't need to do this. Skip to the next heading.

- ☐ Update the appserver.classpath property of `was.conf` and provide the following:

  ```
  /usr/lpp/WebSphere/lib/ws390crt.jar
  ```

- ☐ Add a new property to the `was.conf` file (place all on one line in the file):

  ```
  appserver.java.extraparm=-Djava.naming.factory.initial=
     com.ibm.ws.naming.ldap.WsnLdapInitialContextFactory
  ```

- ☐ Update the `httpd.envvars` file and provide the following variables:

  ```
  RESOLVE_IPNAME=(fully qualified IP hostname of WebSphere)
  RESOLVE_PORT=900
  ```

All this is necessary to provide WebSphere V3.5 Standard Edition the knowledge of how to reach the V4 runtime where the EJB resides. The new V4 Plugin has all this built in.

Cut-and-paste values from was.conf.updates files

- ☐ Browse the file `was.conf.updates`, created by the `wartowebapp.sh` utility
- ☐ Copy the contents out of that file
- ☐ Paste into `was.conf`, making sure that no characters far to the right are truncated

FTP client side bindings into classpath

Back in "Create client-side bindings file" on page 14 you created a file called `MQSessionClientBindings.jar` (or you copied the same file supplied in the `WP100301_Components.zip` file). Now is the time to make that file available to your servlet.

- ☐ Take a look at the `deployedwapp.<app_name>.classpath=` property that you pasted into the `was.conf` file. Note the directory named on that property.
- ☐ FTP the file `MQSessionClientBindings.jar` in binary mode into the classpath directory

Note: That file can go into any `CLASSPATH` directory accessible by the servlet. It's probably best to put the file into the directory named on the `deployedwebapp.<name>.classpath=` statement generated by the `wartowebapp.sh` utility.

Provide appserver.classpath and appserver.libpath pointers to MQSeries in was.conf

- ☐ Update `appserver.libpath` in the `was.conf` with a pointer to the MQSeries Java "lib" directory:

  ```
  /usr/lpp/mqm/java/lib/
  ```

- ☐ Update `appserver.classpath` in the `was.conf` with the a pointer to the following JAR files and directories:

  ```
  /usr/lpp/mqm/java/lib:
     /usr/lpp/mqm/java/lib/com.ibm.mq.jar:
     /usr/lpp/mqm/java/lib/com.ibm.mqjms.jar:
     /usr/lpp/mqm/java/lib/com.ibm.mqjms.jar:
     /usr/lpp/mqm/java/lib/providerutil.jar:
     /usr/lpp/ldap/lib/jndi.jar:
     /usr/lpp/ldap/lib/ibmjndi.jar
  ```

  Note: The `appserver.classpath` property is coded on one line, with entries separated by a colon. *Do not* code the references broken across lines as shown here.

- ☐ Save the file

Update application webapp file with pointers to QMGR, Queue and Home of MQSession

The `<application_name>.webapp` file is found in the directory named on the `deployedwebapp.<name>.classpath=` property created by `wartowebapp.sh`. The contents of this file are the converted contents of the webapp's `web.xml` file.

Note 1: You could make these changes in WSAD prior to generating the WAR file. Changing the XML file on the 390 box works just as easily.

Note 2: The values you provide for `QMGR_NAME` and `QUEUE_NAME` *must* match what you provided for the J2EE Resources back under "Create J2EE Resources for MQSeries resources" on page 12. In the example provided back on page 12, the Queue Manager name was `MQW1`, and the Queue Name was `CB.IMSA.PUT`.

Note 3: The value for `HOME_NAME` must match whatever value you set for the JNDI name of the MQSession bean when you deployed the EAR file into the J2EE Server. If you took the preset value of `MQSessionHome`, then code that. If you clicked on the "default JNDI" button, then code that whole value on one line here in the XML file. The point is what's coded here must match what's registered in LDAP. Not sure? Use an LDAP browser to see, or go into the SMS EUI tool and check the JNDI name for the session bean.

- ☐ Locate the file `SimMDB_Web.webapp` under the `/SimMDB_Web/servlets` directory
- ☐ Edit the file, and change the following text:

```
<servlet>
    <name>My390Servlet</name>
    <code>com.rc.wsc.web390.My390Servlet</code>
    <init-parameter>
        <name>QMGR_NAME</name>
        <value>Your Queue Manager Name Here</value>
    </init-parameter>
    <init-parameter>
        <name>QUEUE_NAME</name>
        <value>Your Queue Name Here</value>
    </init-parameter>
    <init-parameter>
        <name>HOME_NAME</name>
        <value>JNDI Home Interface Name of Session Bean Here</value>
   </init-parameter>
    <servlet-path>/My390Servlet</servlet-path>
</servlet>
```

Note: If you clicked on the "Default JNDI Name" button of the SMS EUI tool, the value you would code for the JNDI name would be *something* like this:

`/WSLPLEX/WASASR2/WP100301_390_EJB/MQSession/MQSession/com...`

- ☐ Save the file

(Optional) Compile MQClient code

The MQClient program will load 100 messages onto your MQ queue. This step is "optional" because the compiled class file is supplied in the `WP100301_Components.zip` file. But just in case you wanted to compile it yourself, here's how you would do that.

- ☐ FTP in ascii mode the file `MQClient.java` to your 390 system. Place in a working directory such as `/u/user1`.
- ☐ Go into an OMVS session
- ☐ Update your OMVS session's `CLASSPATH` to point to the following directories:

```
/usr/lpp/mqm/java/lib/com.ibm.mq.jar
/usr/lpp/mqm/java/lib/com.ibm.mqjms.jar
```

- ☐ Compile the code with `javac MQClient.java`

Note: Or you could simply FTP in binary mode the file `MQClient.class` found in the `WP100301_Component.zip` file

Load MQ queue using MQClient

- ☐ Go into an OMVS session
- ☐ Update your OMVS session's `CLASSPATH` to point to the following directories:

```
/usr/lpp/mqm/java/lib/com.ibm.mq.jar
/usr/lpp/mqm/java/lib/com.ibm.mqjms.jar
/usr/lpp/mqm/java/lib/providerutil.jar
/usr/lpp/mqm/java/lib
/usr/lpp/ldap/lib/ibmjndi.jar
```

- ☐ Invoke the MQClient program using the following command:

`MQClient <Queue Manager> <Queue Name>`

For example:

```
MQClient MQW1 CB.IMSA.PUT
```

If the program is able to connect to the Queue Manager and locate the Queue Name, it'll load the queue with 100 messages. Your OMVS screen will fill up with *100 iterations* of something like this:

```
  :
Putting message : <Mon Aug 19 09:47:59 EDT 2002> of length 28
Message Id as Bytes : ÝB@4c3d33dc
Message Id Length 24
Message Id as String : CSQ MQW1        ½--$¢
Correlation Id : ÝB@4c33f3dc
  :
```

You're now ready to drive the servlet and have the session bean pull the messages off the queue!

Start J2EE server and HTTP server and drive servlet

- ☐ Start the J2EE application server into which the MQSession bean was deployed. If this is the first time the server has been started since deploying the bean, make sure that naming registration succeeds for the newly deployed application.
- ☐ Start the HTTP Server
- ☐ On your browser, enter the URL with the proper rooturi and servletmapping string:

  `http://<your host>[:port]/SimMDB_Web/My390Servlet`

  Note: If you get an HTTP Server Error 404, it means you didn't properly code the `Service` statement. Go back to "Update httpd.conf with new Service statement" on page 17 and make sure the `Service` is coded properly to match this URL. Restart the HTTP Server and try again.

  You will get a very unfriendly message back even when things are successful.

The less-than-helpful message of success provided by the servlet

  If you get this message, things may be okay. The only way to check is to look in the "ncf" trace.

- ☐ Go to the "ncf" trace of the Plugin and scroll to the bottom. The sign of success will be a string of messages that looks something like this:

```
My390Thread -- Successful in LookUp
My390Thread -- Successful in obtaining the home
My390Servlet -- Leaving servlet init
My390Thread -- In the run method of My390Thread
My390Thread -- Success! In My390Thread Message off queue was :Mon Aug 19 09:47:50
My390Thread -- Success! In My390Thread Message off queue was :Mon Aug 19 09:47:50
My390Thread -- Success! In My390Thread Message off queue was :Mon Aug 19 09:47:51
  :
My390Thread -- Success! In My390Thread Message off queue was :Mon Aug 19 09:47:53
My390Thread -- No messages on queue, so I will take a 3000 millisecond nap
My390Thread -- No messages on queue, so I will take a 3000 millisecond nap
```

  You'll see as many "Success!" messages as you had messages on the queue. Once all the messages have been stripped off the queue, the thread will go into a 3000

millisecond loop and then check again. Unless you put more messages on the queue, the "No messages on queue" notice will repeat forever.

Document Change History

Check the date in the footer of the document for the version of the document.

August 26, 2002	Original document.

End of Document