WebSphere Application Server V4.0 and V4.0.1 for zOS and OS/390

Configuring Web Applications

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number WP100238 under the category of "White Papers"

Date: *Thursday, April 18, 2002*

IBM Washington Systems Center

Donald C. Bagwell
301-240-3016
dbagwell@us.ibm.com

This document would not have been possible without the assistance of others in the Washington Systems Center who offered guidance, supplied subject material and reviewed the final draft. In particular much credit must flow to Mike Cox, who unlocked the mystery of WAS 4.0 webapps so the rest of us could follow. In addition, John Hutchinson, Carl Wohlers and Tom Hackett provided valuable content updates and review. Finally, Mike Kearney provided input on issues related to security.

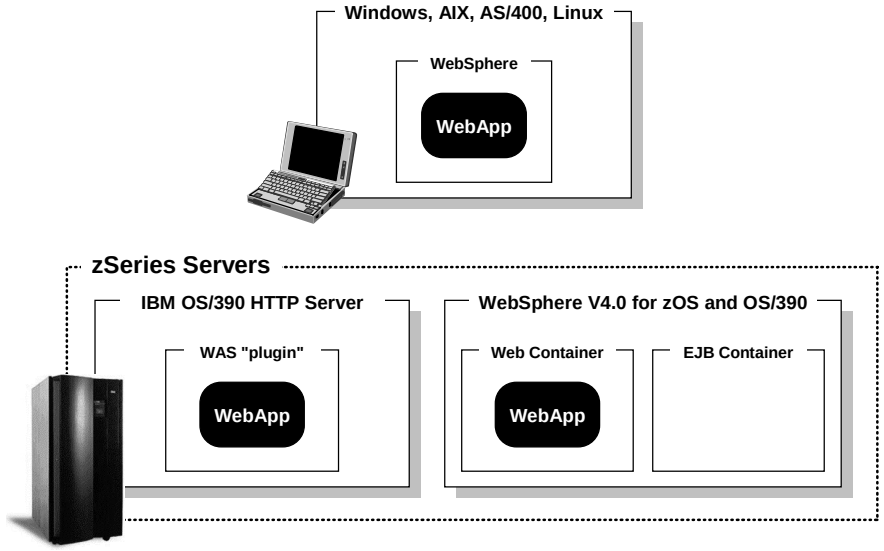
Table of Contents

Change History

December 7, 2001	Original document
February 11, 2002	Added section on HTTP Authentication based on security constraints defined in the webapp's deployment descriptor rather than `Protect` statements in the `httpd.conf` file.
April 18, 2002	Cleaned up the topic of binding a given application to multiple virtual hosts.

Overview of Web Applications

A web application is a servlet (or some number of servlets) working in conjunction with other web files such as HTML pages, JPG/GIF image files and Java Server Page (JSP) files. A servlet is a Java program, and the servlet requires a servlet execution environment in which to run. IBM provides many different places in which you may run a servlet:



IBM servlet execution environments

Note: The message here is that anywhere IBM provides support for WebSphere Application Server, servlets will run there. This chart doesn't show HP-UX and Solaris, but servlets can run there as well because WebSphere Application Server is supported on those platforms.

Background: the basics of serving out a web applications

Regardless of the environment in which the webapp is running, *three fundamental things* must be in place:

1. Some piece of programming code must act as the listening for HTTP requests coming in from the network. The browsers out in the world will send their requests in using the HTTP protocol, and this "HTTP Listener" must be active to catch the request.

   For the zSeries platform, that HTTP listener is the IBM HTTP Server or the new "Transport Handler," which is an HTTP listener integrated into the WAS 4 runtime environment.

2. The HTTP listener's configuration must be set so that requests for servlets are recognized as such. This allows the HTTP listener to pass the request over to the servlet execution environment. Requests coming in from browsers may be for many different things: HTML pages, image files, or requests for programs (such as servlets) to be run. This ability to determine the nature of the request and properly act on it is critical.

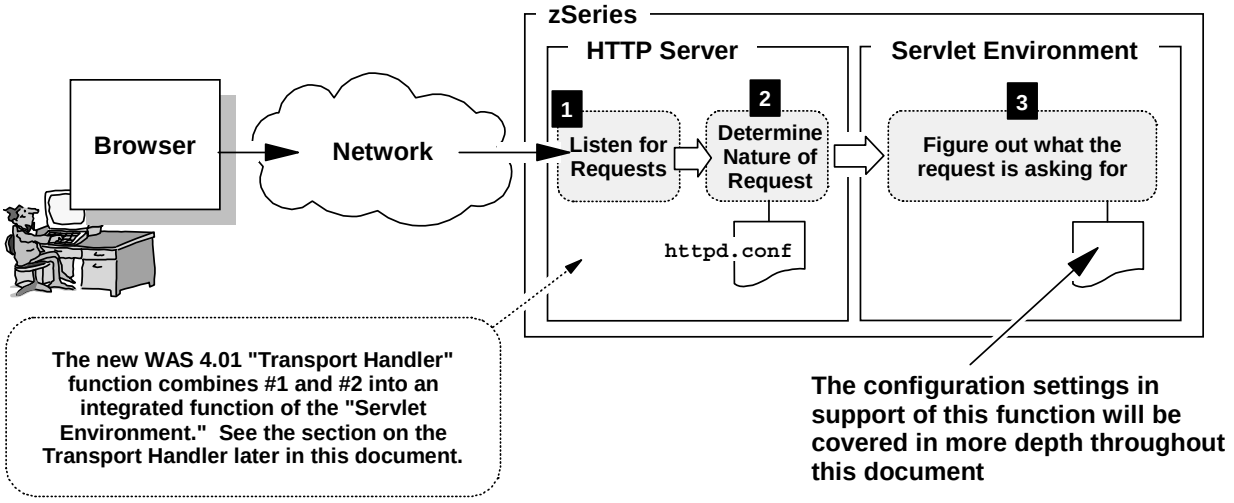
   For the IBM HTTP Server, the file in which this ability to determine the nature of the requests is the `httpd.conf` file. The configuration statement in that file that directs requests for servlets over to the servlet environment is the `Service` statement. The Transport Handler is designed to be a

streamlined pipe to the servlet environment and attempts to route all requests to the servlet execution environment.

3. The servlet execution environment's configuration must be set to allow that environment to figure out *what* is being requested by the sender of the request. The servlet environment may have dozens or hundreds of servlets it knows about. Which one is being asked for? Further, web applications consist of not just servlets, but static files such as HTML and image files. The servlet environment is capable of simply sending them out as well. Is that what is being requested?

   The configuration file settings for this function differs depending on whether the web application is running in the "WAS plugin" of the HTTP Server, or in the "Web Container" of the WAS 4.0 runtime. This subject will be explored in depth in this document.

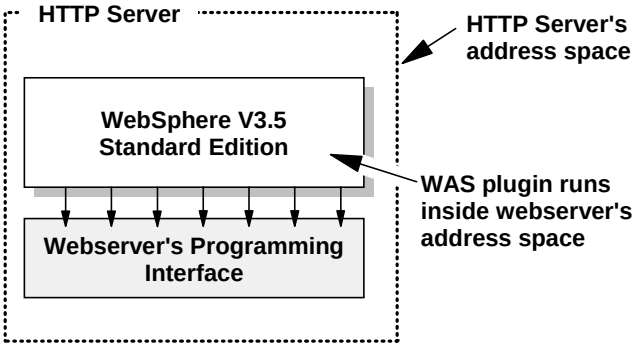
Understanding these three basic functions of serving out web applications helps when trying to understand all the configuration files and statements that will be discussed in this document. These three functions are represented in the following picture:



Three basic functions involved with serving out a web application

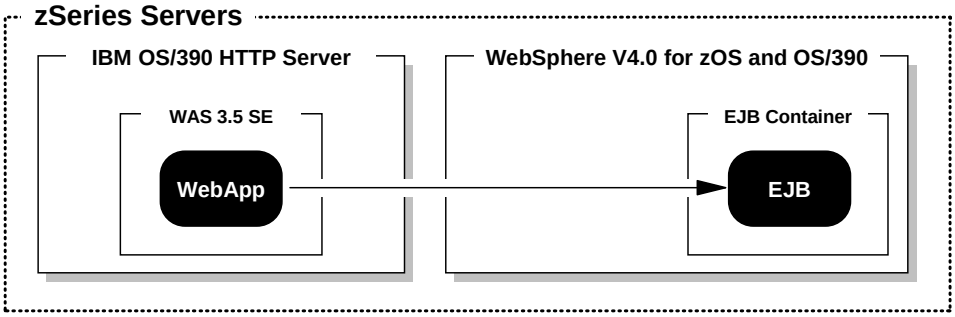
Background: WAS 3.5 Standard Edition plugin

Prior to the introduction of WebSphere V4.0 for zOS, servlets on the zSeries platform were run in the WAS 3.5 Standard Edition product, which is a "plugin" to the HTTP Server. It is called a "plugin" because the code runs *inside* the webserver's address space and makes use of a programming interface provided by the webserver. Therefore, it is said to "plug in" to the webserver:



The WAS "plugin" to the HTTP Server

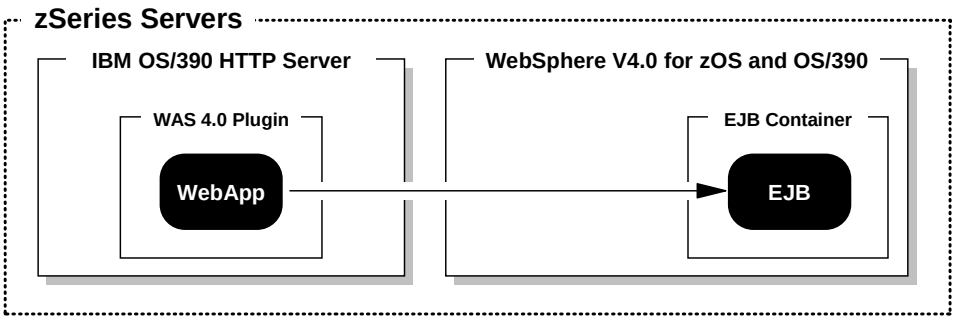
The WAS 3.5 Standard Edition environment still exists. It is possible for servlets running in this environment can talk to EJBs running in the WAS 4.0 EJB environment:



Webapp in WAS 3.5 plugin talking to EJB in WAS 4.0 runtime

Background: WAS 4.0's webserver plugin

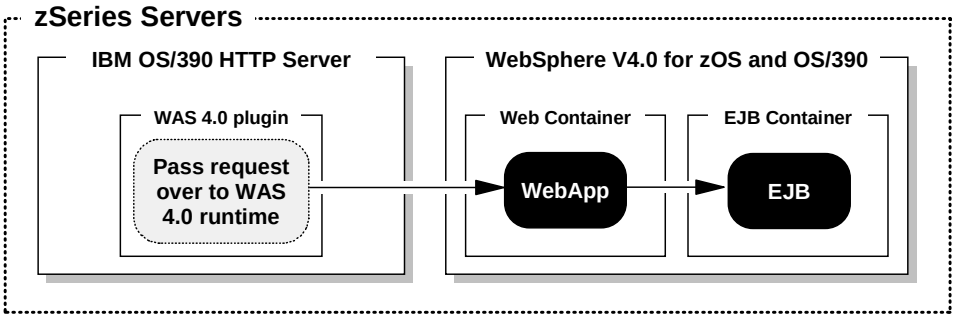
The WAS 4.0 product ships with a module that on the surface looks and feels just like the WAS 3.5 SE plugin. It too can run inside the webserver's address space, and servlets can run inside of it:



Webapp in new WAS 4.0 plugin talking to EJB in WAS 4.0 runtime

The servlet execution environment provided by the new WAS 4.0 plugin is pretty much equal to that provided by WAS 3.5 SE. In fact, the WAS 3.5 SE code is wrapped up inside the new WAS 4.0 plugin.

But the new WAS 4.0 plugin has a feature the older WAS 3.5 SE plugin doesn't have: the ability to understand what webapps are deployed in the WAS 4.0 runtime's "web container" and to route requests over to web container:

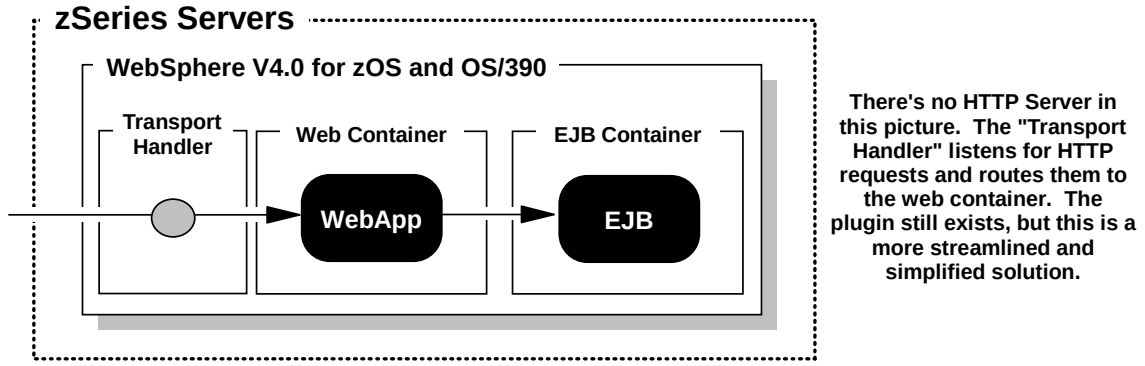


WAS 4.0 plugin acting as request router to webapp running in WAS 4.0 runtime

The WAS 4.0 plugin in this scenario acts as a traffic cop of sorts, routing the requests over to the environment where the servlet will run. This scenario is the primary focus of this document. The configuration steps necessary to do this is provided starting with "Webapps Running in WAS 4.0 Runtime and Driving EJB" on page 15.

Background: WAS 4.0.1 Transport Handler

The Transport Handler is an HTTP listener integrated into the WAS 4.0.1 runtime environment. It became available with the release of WAS 4.0.1 in October of 2001:



WAS 4.0.1 Transport Handler listens for HTTP and routes request to web container

The Transport Handler is a streamlined HTTP listener and is designed to quickly route requests to the web container environment. The Transport Handler itself does not have the ability to execute a servlet.

The Transport Handler is for the most part a replacement of the HTTP Server and Plugin. Typically you would use one or the other (though you can use both at the same time if you want). This topic is covered in more detai under "The WAS 4.0.1 Transport Handler" on page 73.

Background: WAS 4.0's web container

As described earlier, the new WAS 4.0 plugin code has the ability to route requests for webapp execution over to the WAS 4.0 runtime. The picture showed something called a "web container," and it was in that box that the webapp was represented.

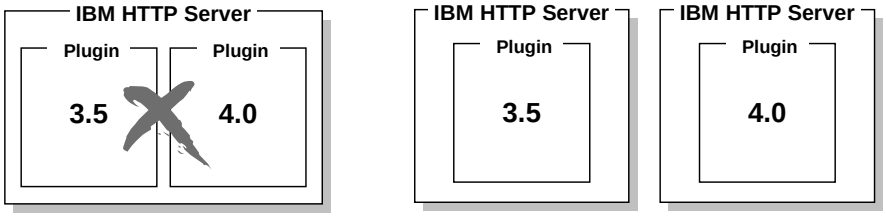
A web container is a logical software structure within the coding of the WAS 4.0 product. The WAS 4.0 runtime has another type of container as well: an "EJB Container." Their mission in life is to provide a place in which the two types of applications -- webapps or EJBs -- will run.

This primary configuration file for the web container is called `webcontainer.conf`. That file and its contents are covered in "Webapps Running in WAS 4.0 Runtime and Driving EJB" on page 15.

For a webapp to run *in the web container* of the WAS 4.0 runtime, *you must have the WAS 4.0 plugin configured in the HTTP Server*.

Question: can both plugins be configured in the same webserver?

No. The plugins will conflict with one another if they are configured within the same webserver. If you have a need for both plugins to be active, you must provide separate webservers: one for each plugin:



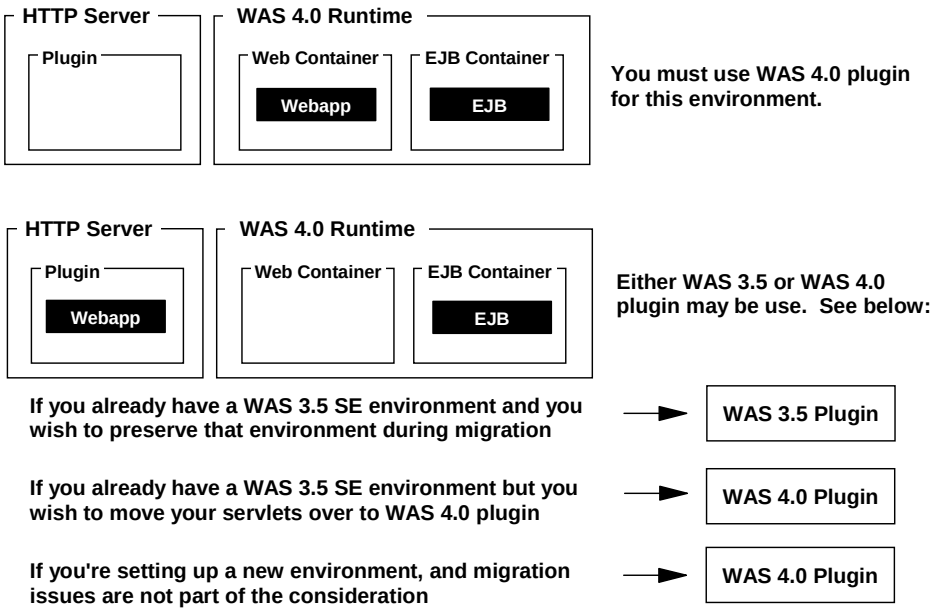
WAS 3.5 and WAS 4.0 plugin cannot coexist in the same webserver

Question: will the webserver support the WAS plugin and another product's plugin?

Most likely. There's nothing about the webserver's API architecture that prohibits multiple plugins from coexisting in the same address space. The restriction applies only to attempting to have the WAS 3.5 and the WAS 4.0 plugin running concurrently.

Question: which plugin should be used?

This depends on what you wish to do. The following diagram summarizes the options:



Summary of which plugin you should use

If you wish to deploy webapps into the WAS 4.0 web container, the decision becomes simple: you must use the new WAS 4.0 plugin. If your desire is to run the webapp in the plugin environment, the general rule of thumb is to use the new WAS 4.0 plugin unless you wish to maintain your existing WAS 3.5 plugin for migration purposes.

You'll see in "Activity: preparing a WAS 3.5 was.conf for use with WAS 4.0 plugin" on page 69 that the new WAS 4.0 plugin will happily use a WAS 3.5 plugin configuration file, so migrating to the new plugin is fairly easy.

Question: when should I use the Transport Handler vs. the plugin?

The Transport Handler is designed to listen for HTTP requests and route them quickly to the web container. The Transport Handler does *not* have an internal servlet execution environment; its sole function is to grab requests off the network and pass them to the web container. Therefore, the Transport Handler is designed to be used when *all the webapps are deployed into the web container environment*. If you still wish to run servlets in the WAS 3.5 Standard Edition runtime, you'll have to have a plugin somewhere in the picture.

At the time of this writing the Transport Handler had some notable limitations (see "Question: does the new feature have all the capabilities of the HTTP Server?" on page 73). Understand the nature of those restrictions before committing to the new function.

Please refer to "The WAS 4.0.1 Transport Handler" on page 73 for more information on the Transport Handler.

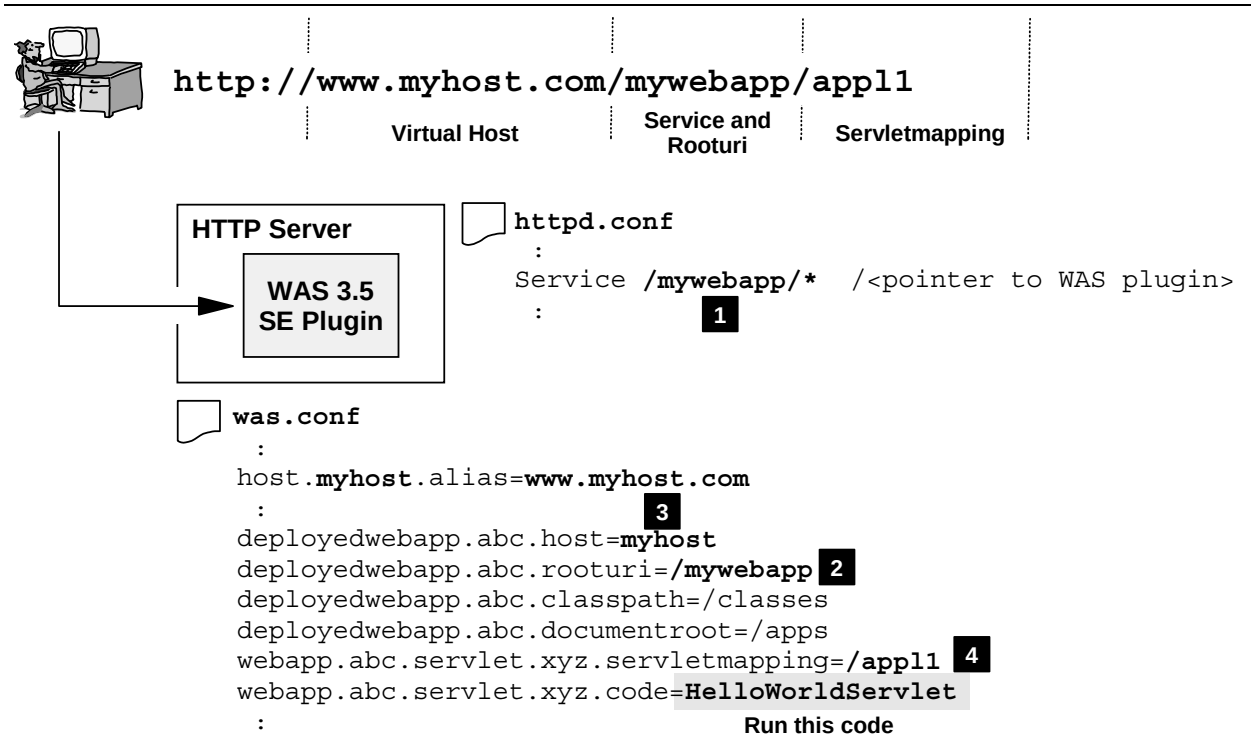
Question: may I use the Transport Handler and the plugin in the same configuration?

The two may be configured and operating in the same configuration. However, a given request from a browser client will use one or the other of the listeners, but not both. So the answer is a qualified "yes." This topic is covered in more detail in "Question: how do I configure both the plugin and the Transport Handler for a given server?" on page 77.

A Big Picture of How it Works

This section provides several high-level pictures of the flow, and shows how a URL from a browser works its way to the execution of a servlet.

How it worked in WAS 3.5 (and when you run servlet in WAS 4 plugin today)



The flow to servlet execution in WAS 3.5 SE environment

Assume a user sends a URL as shown in the chart above. The URL breaks down into three sections, and those sections come into play by mapping to different configuration settings:

1. The `Service` statement in the `httpd.conf` file (the webserver's configuration file) is what maps a request over to the plugin environment. If the URL matches the template on the `Service` statement, the webserver will "throw the request over the wall" to the plugin. Where it passes the request is defined on the `Service` statement after the `/mywebapp/*` template. That's where the plugin's executable module is defined, as well as the "entry point" to be invoked in that module. In the case of the `Service` statement, `:service_exit` is the entry point.

   If you fail to code a `Service` statement for a URL, the request will never get mapped to the plugin, and the servlet will never be executed.

2. Once in the WAS 3.5 environment, the plugin will scale the `was.conf` file (the plugin's configuration file) and go looking for a `rooturi=` value that matches the URL received. WAS will match -- character for character -- with the received URL in an attempt to determine which `rooturi=` definition in the `was.conf` applies to this request. Once WAS gets a hit, it then "bundles up" all the other definitions related to this webapp using the `<webapp-name>` portion of the defintion (in this example, `abc`).

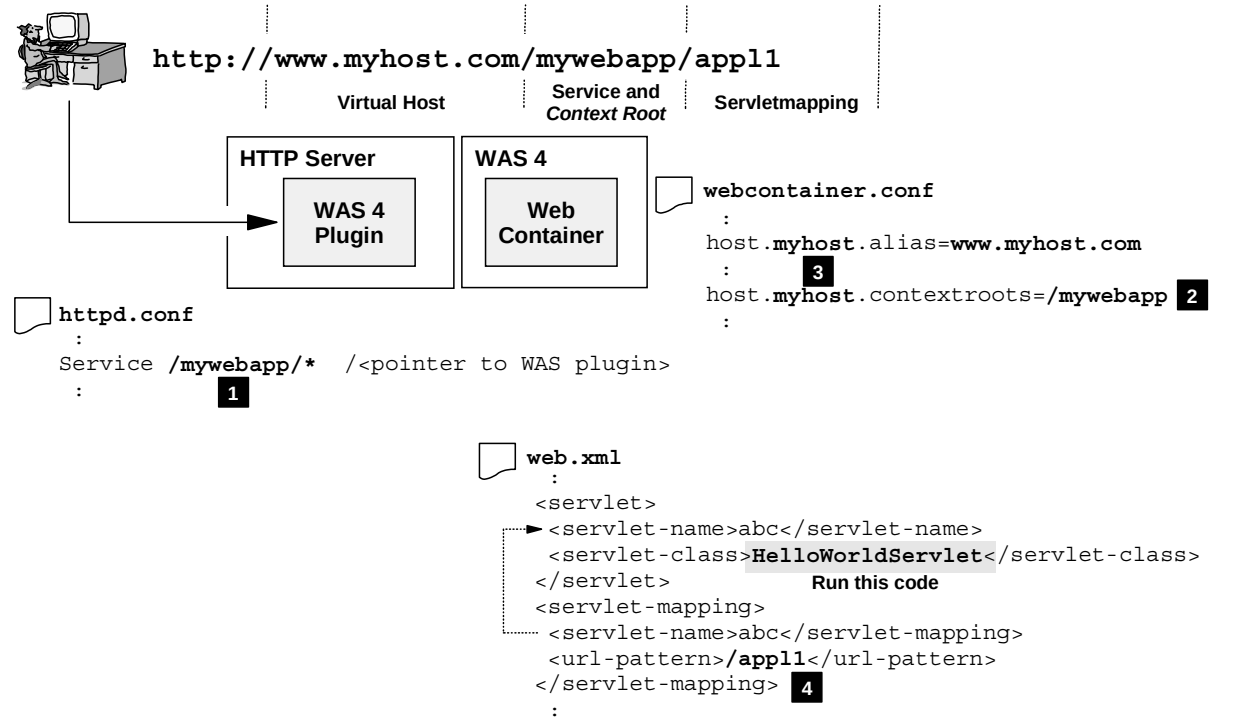
3. WAS next inspects the `host=` definition from the block of definitions it "bundled" in the previous step and relates the value found to a `host.<name>.alias=` statement from earlier in the `was.conf` file. The value on the `alias=` statement is the *virtual host*, and for this request to run the servlet, the host name on the URL must match the virtual host.

Failure to match will mean WAS will discard the bundle of definitions and go looking for another `rooturi=` to match.

4. Assuming you have a match on the `rooturi=` and the virtual host, WAS then looks for a `servletmapping=` definition. It takes the value it finds there, concatenates it to the *rooturi* value, and if it matches the URL it received, it knows the request is to run a servlet. The only question remaining is what servlet, and that's answered with the `code=` definition. The value defined there is the class file to be executed.

   Note: What is illustrated here is an example of "specific servletmapping," where the Java class file is named in the `was.conf` rather than implied on the URL itself. In addition to `servetmapping`, you may also define `jspmapping` and `filemapping` definitions. The point is there's more complexity to this than shown here. But this gets the concept across.

How it works in WAS 4.0.x when using the plugin



The flow to servlet execution in the WAS 4.0.x web container environment using the plugin

In this scenario the servlet is executed in the WAS 4.0.x web container rather than the plugin. The plugin is still in the picture because it is acting as HTTP listener and will "route" the request over to the WAS 4 runtime.

1. The `Service` statement in the `httpd.conf` file (the webserver's configuration file) is what maps a request over to the plugin environment. If the URL matches the template on the `Service` statement, the webserver will "throw the request over the wall" to the plugin. Where it passes the request is defined on the `Service` statement after the `/mywebapp/*` template. That's where the plugin's executable module is defined, as well as the "entry point" to be invoked in that module. In the case of the `Service` statement, `:service_exit` is the entry point.

   The servlet *will not* be run in the plugin. The plugin is simply acting as an intermediate router of the request.

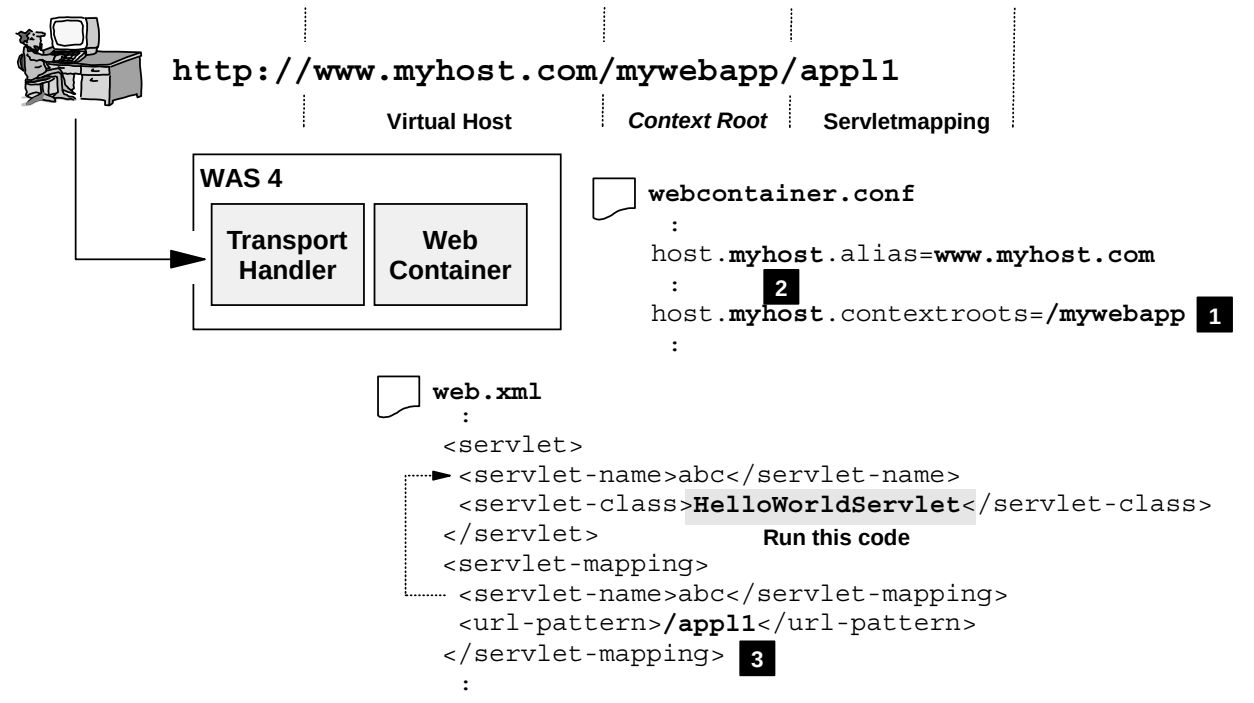
2. Once the request is mapped over to the WAS 4 web container, the web container will search through all the virtual-host/context-root pairs it has knowledge of to see if the request matches. This comes out of the `webcontainer.conf` file, which uses `contextroots=` definition to "bind" applications to virtual hosts. The context root is nearly identical in concept to the rooturi of WAS 3.5, as is the virtual host. If the URL received matches a virtual-host/context-root pair, then the web container knows to proceed. If no match is found, then WAS will reject the URL.

   Note: The concept of "binding" an application to a virtual host is covered later in this document, starting at "Background: binding applications to virtual hosts" on page 19.

3. This step illustrates the connection between the `contextroots=` statement in `webcontainer.conf` and the `host.<name>.alias=` statement, which defines the virtual host.

4. WAS next goes looking for an application whose defined *servletmapping* value matches that implied on the received URL. In this example the servletmapping string on the URL is `appl1`, and it knows this because the servletmapping value is whatever comes after the context root value on the URL. WAS looks through the `web.xml` files contained in each web application's WAR file looking for a `<url-pattern>` tag that matches the servletmapping string on the URL. When it finds one, it takes the `<servlet-name>` value and goes to the point in the `web.xml` file where `<servlet-class>` is defined. That's the class file that is executed.

How it works in WAS 4.x when using the Transport Handler



The flow to servlet execution in the WAS 4.0.x web container environment using the Transport Handler

This scenario is nearly identical to the previous, except that the plugin is not part of the picture. Rather, the new "Transport Handler" acts as the HTTP listener.

1. The request goes directly to the web container through the Transport Handler, and the same matching on the virtual-host/context-root pair is done, just like what was illustrated in the previous scenario.
2. The connection between `contextroots=` and `alias=` is exactly the same as illustrated in the previous scenario.
3. The `<url-pattern>` and `<servlet-class>` processing is exactly the same as illustrated in the previous scenario.

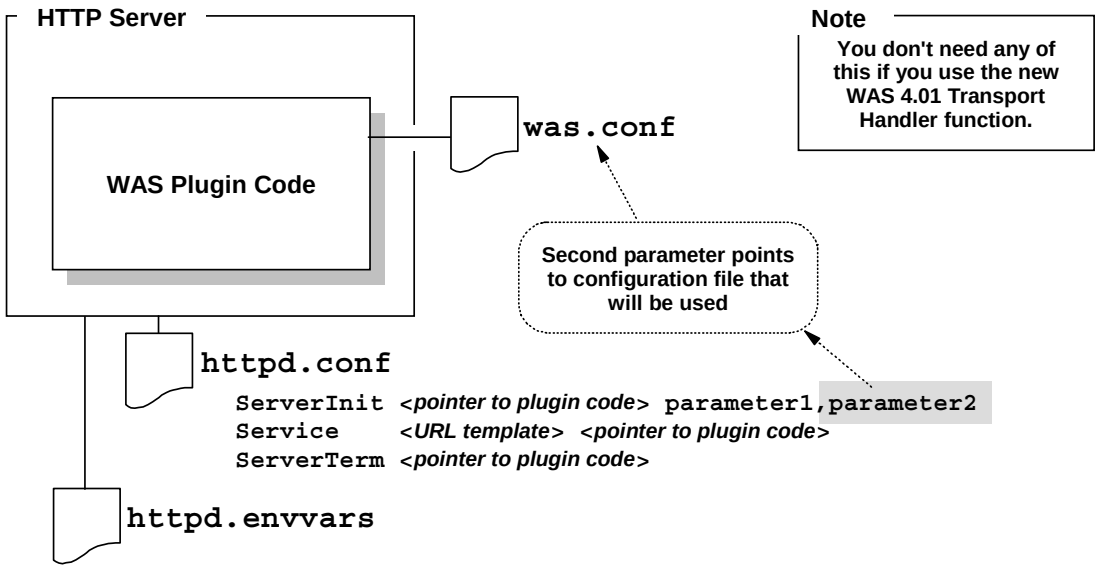
The only difference between this scenario and the previous is there's no plugin involved, so no `Service` statement is used to map the request to the plugin. The request flows directly to the Transport Handler, which is listening for HTTP requests. Beyond that, everything else is the same.

Initial Configuration of the Webserver Plugin Code

Overview

As stated earlier, the WebSphere Application Server plugin code executes within the address space of the HTTP Server. That is true whether the plugin is the WAS 3.5 SE plugin, or the newer WAS 4.0 plugin.

The plugin's configuration file is called `was.conf`. The webserver knows to load the plugin and use a particular copy of `was.conf` based on the second parameter found on the `ServerInit` statement coded in the webserver's `httpd.conf` file



Relationship between HTTP Server's configuration files and plugin's configuration file

Here's what each plugin statement does:

`ServerInit`	This statement is used to tell the webserver to initialize the plugin code when the webserver is in the act of coming up. The `ServerInit` statement points to the plugin's executable module and the code exit to invoke to intialize the plugin. This statement has two parameters: the first names the HFS install root of the WAS product, and the second parameter points to the configuration file to be used by the plugin. You will have only one `ServerInit` statement in your `httpd.conf` file.
`Service`	This statement is used to map URLs received by the webserver over to the plugin for execution. If the URL received matches the "URL template" named on the `Service` statement, the webserver will take the URL and pass it to the plugin executable module and code exit specified on the statement. If you want a URL to execute a servlet, you must have a `Service` statement that'll catch the URL and "throw it over the wall" into the plugin. You will have between one and many `Service` statements in your `httpd.conf` file, depending on how many different URLs you wish to define that'll map over to the WAS plugin environment.
`ServerTerm`	This statement is used to bring the plugin down in an orderly fashion when the webserver is stopped. This statement simply points to the plugin executable module and code exit used to shut down the plugin gracefully. You will have only one `ServerTerm` statement in your `httpd.conf` file.

The exact syntax of each statement is provided in "Activity: configuring the WAS 4.0 plugin code" on page 12.

Activity: configuring the WAS 4.0 plugin code

Very Important Note: Any given webserver may have *either* the WAS 4.0 plugin configured, *or* the WAS 3.5 plugin configured, *but not both at the same time*! See "Question: can both plugins be configured in the same webserver?" on page 5.

Do the following:

- ☐ Copy the supplied sample `was.conf` file from the following directory:

  `/`*WAS 4.0 Install Root*`/WebServerPlugIn/properties`

  to the directory in which the `httpd.conf` file resides.

  Note: If you're wondering if you can use your existing WAS 3.5 SE `was.conf` with the new WAS 4.0 plugin, the answer is "yes." How this is done, and other migration related topics, is provided in "Migration Scenarios" on page 67.

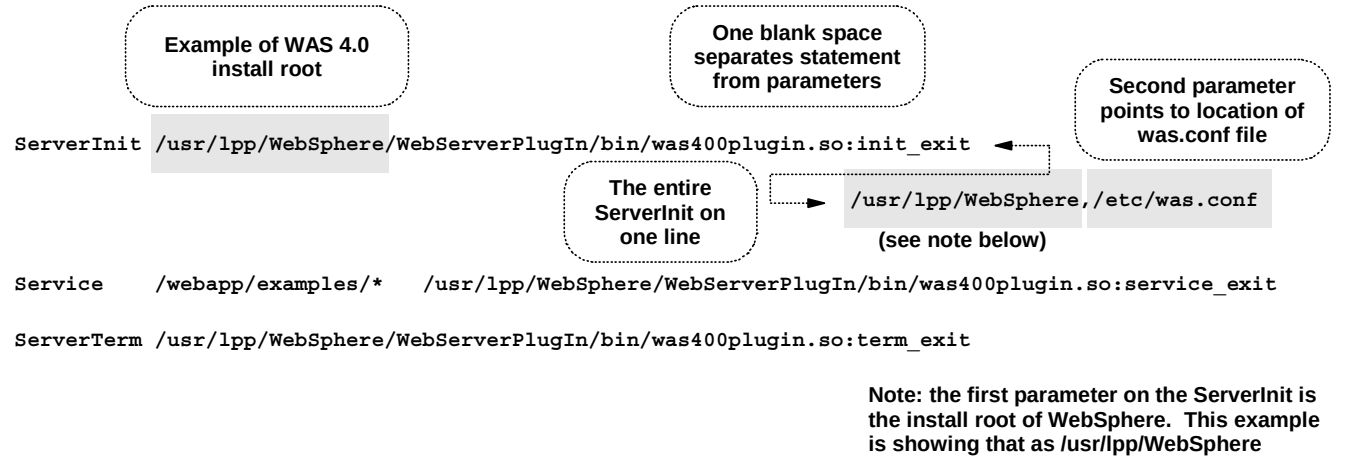
- ☐ Edit the file `httpd.conf` and locate the following string starting in column 1:

```
Service /servlet/* /usr/lpp/WebSphere/AppServer/lib/libadpter.so:AdapterService
```

  Comment out this line. It is leftover from the days of WAS 1.2 and will cause confusion if any URL comes in that maps to the `/servlet/*` mask on the `Service` statement.

- ☐ Immediately following the line you just commented out, add the following:

  Note: The directory "WebServerPlugIn" has a capital "I" for "In" at the very end. It is very easy to not see that and type that with a lowercase. Making a tiny mistake like that does matter. Be very careful with your typing.



Statements added to httpd.conf to support WAS 4.0 plugin initialization

Note: The `Service` statement provides the "basic function" #2 from "Background: the basics of serving out a web applications" on page 1.

- ☐ Edit your `httpd.envvars` file and add the following:

  `JAVA_HOME=/usr/lpp/java2/J1.3`

  or wherever the Java 1.3 Developer Kit for Java is installed on your system.

- ☐ Stay in your `httpd.envvars` and add the following to the `NLSPATH` variable:

  `/usr/lpp/WebSphere/WebServerPlugIn/msg/%L/%N`

  or whatever your WAS 4.0 install root happens to be.

- ☐ Add the following two variables to `httpd.envvars`:

  `RESOLVE_IPNAME=`*<fully qualified IP host name of server on which WAS 4.0 SMS exists>*
  `RESOLVE_PORT=900` *(or port on which WAS 4.0 SMS server is listening if not default)*

  Note: If your HTTP Server (and therefore the plugin as well) is on the same system as your WAS 4.0 runtime, and you configured the SMS server to use the default port value of 900, you don't need these two values. But coding them is relatively easy, and it avoids confusion. So go ahead and code these even though strictly speaking they're not always necessary.

- ☐ Now go to "Activity: validation and basic debugging of plugin" on page 13 for how to validate the plugin initializes properly.

Background: configuring and running servlets in the WAS 4.0 plugin

The WAS 4.0 plugin has imbedded within it the WAS 3.5 Standard Edition execution environment. That means the WAS 4.0 plugin is capable of running web applications just like WAS 3.5 Standard Edition was. The process of configuring the `deployedwebapp` and `webapp` statements inside the `was.conf` file is exactly the same as with WAS 3.5 SE.

The subject of configuring web applications for execution in the plugin is covered in the IBM document GC34-4835, "WebSphere Application Server Standard Edition, Planning, Installing and Using."

If the servlet you are deploying will drive an EJB in the WAS 4.0 runtime, the servlet will locate the EJB with the aid of the `RESOLVE_IPNAME` and `RESOLVE_PORT` settings in the `httpd.envvars` file. The WAS 4.0 plugin is able to locate the WAS naming service "initial context factory" because its been coded to know where that's located. The WAS 3.5 plugin, by contrast, must be told where that resides.

Activity: validation and basic debugging of plugin

Note: These instruction apply to both the WAS 3.5 plugin as well as the WAS 4.0 plugin

- ☐ Start the webserver.
- ☐ Now browse the `SYSOUT` of the `BBOWEB` started task and find the following string:

  `:-)`

  Yes, that's a "smiley face", and that's an indication that the plugin initialized okay. The plugin sometimes takes a few moments to initialize, and may not be up even though the webserver is operational. Give it a few moments and try again if you don't see it initially.

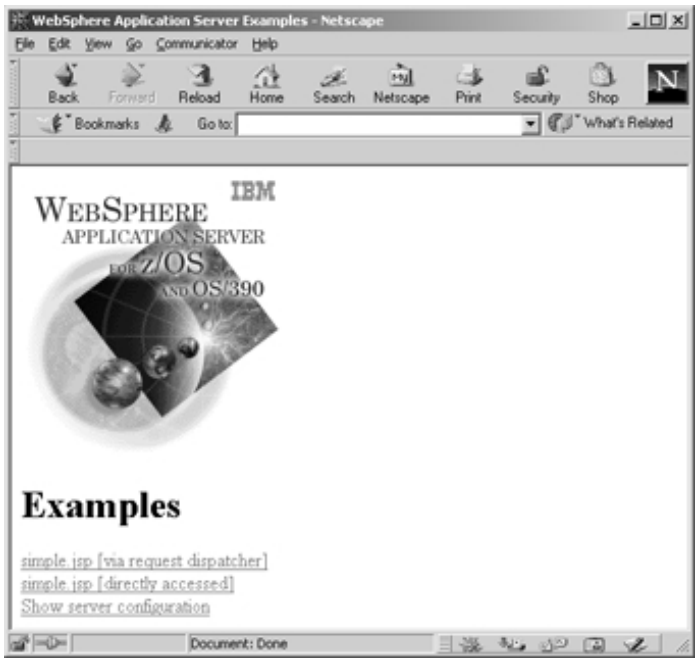
  If you still can't find the "smiley" face, search on the "frowny face" **`:-(`**. The webserver will throw that message if something prevented the plugin from initializing. Common causes for plugin initialization failure:

  - `JAVA_HOME` variable in `httpd.envvars` not set correctly.
  - Mistyped directory or file name on `ServerInit` statement in `httpd.conf`. Check for case problems.
  - Second parameter on `ServerInit` statement points to `was.conf` file and/or directory that does not exist.

- ☐ Once you've verified the smiley face, issue the following URL from your browser:

  `http://<host>/webapp/examples/index.html`

  You should see a screen that looks something like this:



Initial screen for "webapp/examples" that validates basic operation of plugin code

If you receive this, it is an indication that your URL was successfully mapped over to the WAS 4.0 plugin using the `Service` statements in `httpd.conf`. You have successfully invoked the plugin's function to serve out the static page you see above.

Next Steps

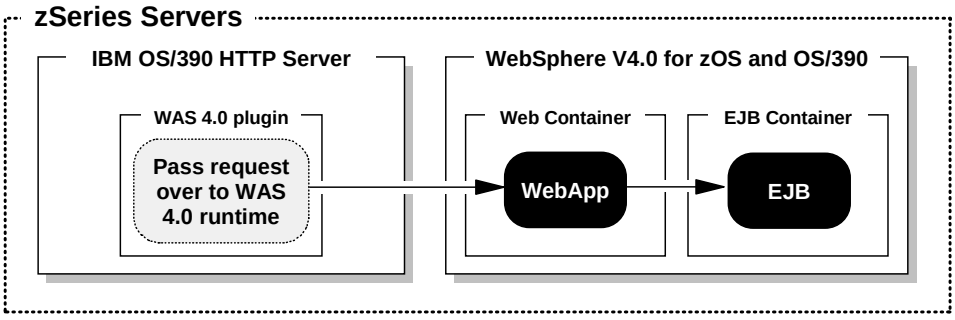
By achieving the smiley face you have taken the first step towards establishing your webapp environment. But you have not invoked any webapps and no EJBs are yet in the picture.

Proceed to "Webapps Running in WAS 4.0 Runtime and Driving EJB" on page 15.

Webapps Running in WAS 4.0 Runtime and Driving EJB

Overview

This is the scenario that is the focal point of this document. The web application is deployed into the web container of the WAS 4.0 runtime. The WAS 4.0 plugin is configured into the HTTP Server, and requests received by the webserver are routed over to the web container for webapp execution:



WAS 4.0 plugin acting as request router to webapp running in WAS 4.0 runtime

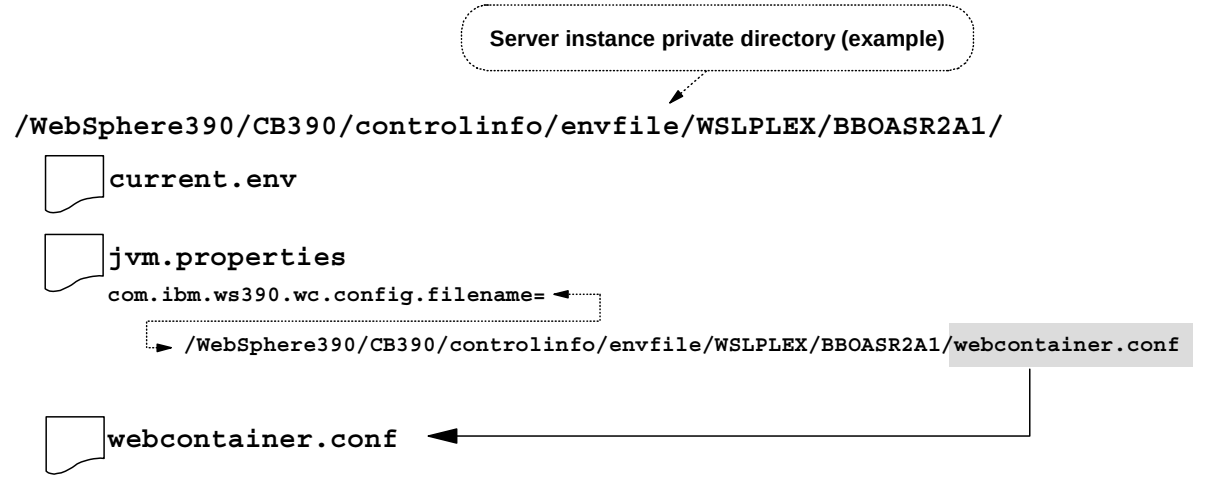
To illustrate this, the "PolicyIVP" sample application shipped with the WAS 4.0 product will serve as the EJB application. That sample application has an EJB environment (consisting of one session bean and two entity beans) as well as a webapp.

Note: With WAS 4.0.1 a new method getting the request passed over to the WAS 4 runtime was made available: the "Transport Handler." It is an HTTP listner that is integrated into the WAS 4 runtime environment. The Transport Handler eliminates the requirement for the plugin (though you may still have the plugin in the picture concurrent with the Transport Handler if you wish). For more information on the Transport Handler, see "The WAS 4.0.1 Transport Handler" on page 73.

Configuration

Activity: creating the webcontainer.conf file

The web container utilizes a file called `webcontainer.conf` to hold configuration information for the container. The server instance knows what file to use for this purpose by reading the contents of the `jvm.properties` file and looking for a pointer to the container configuration file:



How the server instance knows what web container configuration file to use

Do the following:

- ☐ Determine the private directory of your server instance. This is where the `current.env` and `jvm.properties` files resides. This is also where you will copy the sample `webcontainer.conf` file. Write down the location of that private directory:

  _____

- ☐ Copy the sample `webcontainer.conf` file from the following location to the private directory of your server instance:

  From: /WAS 4.0 install root/`bin/webcontainer.conf`

  To: /Server Instance Private Directory/`webcontainer.conf`

- ☐ Edit the `jvm.properties` file in your server instance's private directory and add the following line (all on one line), which points to the new `webcontainer.conf` file in your server instance's private directory:

  `com.ibm.ws390.wc.config.filename=`/private directory/`webcontainer.conf`

- ☐ Make certain that `webcontainer.conf` file has permissions of at least `644` and is owned by the same userid which owns the `current.env` file (should be `CBSYMSR1`, or whatever the ID for your Systems Management Server -- SMS -- is).

  Note: Some have asked if changing the ownership of this file to something other than the Systems Management Server ID is permitted. That'll work, but you have to be very careful. If you ever use the SMS GUI tool to delete the server, the SMS ID will be the one that performs the file deletions, and it *must* have authority to do so. You can find yourself in a pretty messy state if the SMS can't delete files at the time of server deletion. Therefore, the recommendation is to assign ownership of this file to the SMS ID.

There are many configuration statements in the `webcontainer.conf` file, but only two that you must modify to make it work:

```
host.default_host.alias=
host.default_host.contextroots=
```

Both relate to the concept of "virtual hosts," which is discussed next.

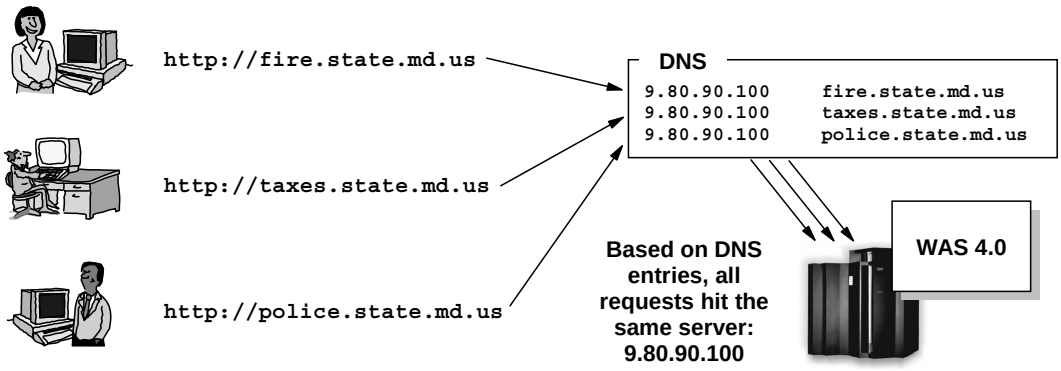
Question: do I need to configure a webcontainer.conf if I use the Transport Handler?

Absolutely. The webcontainer.conf provides the configuration for the web container, which exists regardless of the HTTP listener you employ out front. So it is required, as is the definition of the virtual hosts and the binding of applications to the virtual hosts (all described next).

Background: the concept of virtual hosts

The HTTP Server has for some time now had the ability to handle URLs with different host names. This allows you to host multiple host domains on the same webserver. This functional concept has been included in the WAS 4.0 runtime's web container environment as well.

Imagine a scenario where you are asked to put up a web application environment for three different divisions of a company, each with its own host name. You are told to do it all on one server:



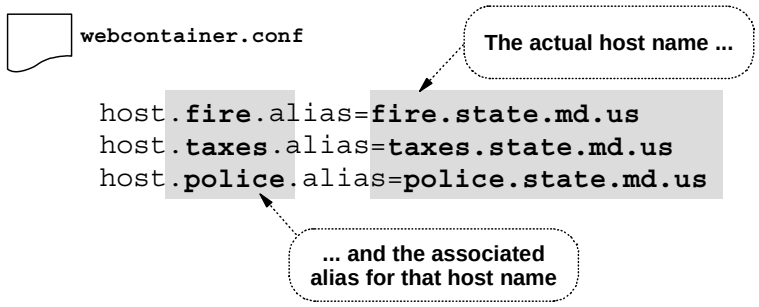
One webserver hosting three different host domains

Further, you are told to make certain that anyone coming in with host name `fire.state.md.us` has access only to those applications for the Fire Department, and the same for the Police Department and the Department of Taxation.

The web container support of WAS 4.0 for zOS and OS/390 is quite capable of handling this. Doing so involves first defining virtual hosts in your `webcontainer.conf` file, and then indicating which applications are associated (or "bound") to which virtual host.

Background: defining virtual hosts in the webcontainer.conf file

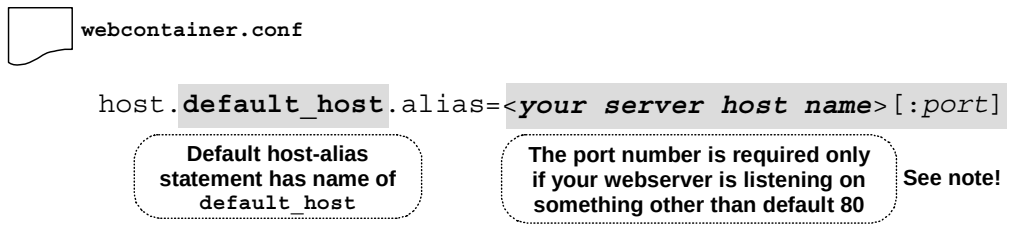
A virtual host is defined in the `webcontainer.conf` file by providing that host name on a `host.<name>.alias=` statement:



Configuring virtual host aliases in webcontainer.conf

After the virtual hosts have been defined, another definition in the `webcontainer.conf` file will be used to "bind" applications to a given virtual host. With an application bound to a virtual host, only URLs coming in with the IP host name connected to the virtual host will be allowed to execute the application. More on that subject in "Background: binding applications to virtual hosts" on page 19.

The first question that comes to mind for most people is, "What if I am hosting *only one* host name on my server?" You will still need to code a host alias. You can take advantage of the "default host alias definition that's in the sample `webcontainer.conf` file:



One virtual host alias defined using default host alias in sample webcontainer.conf

Note: The webapp support of WAS 4.0 had a problem initially when the default port 80 was used. PTF UQ57590 addresses that problem (APAR PQ50839). Your copy of WAS 4.0 may not have that fix. So if you're in test mode and things don't work with the default HTTP listen port of 80, change the `Port` directive in your `httpd.conf` file to something like `8080` and then code the `8080` in the `webcontainer.conf` file's virtual host alias statement and try again.

Activity: defining a virtual host in the webcontainer.conf file

Note: You must have at least one `host.<name>.alias=` statement defined in your `webcontainer.conf`

Do the following:

- ☐ Edit your copy of `webcontainer.conf`, locate the `host.default_host.alias=` statement and provide the host name that will be used by browsers to reach your webserver. Include a port designation if the webserver is listening on something other than port 80.

Example: `host.default host.alias=wsc4.washington.ibm.com`

Using a `<name>` value of `default_host` and a single virtual host with IP host name of `wsc4.washington.ibm.com`. Webserver is listening on default port 80, so no port designation provided.

Example: `host.default_host.alias=wsc4.washington.ibm.com:8080`

A single virtual host using the sample `default_host` name. The webserver is listening on port 8080, so the port designation `:8080` is provided on the virtual host definition.

- ☐ If you have more than one virtual host to define, you can do that in one of two ways:

  1. Code multiple `host.<name>.alias=` properties in your `webcontainer.conf` file, and provide a different virtual host value on each:

Example:
```
host.taxes.alias=taxes.state.md.us
host.police.alias=police.state.md.us
host.fire.alias=fire.state.md.us
```

Three virtual hosts defined, each with a unique <name> value. The webserver is listening on default port 80, so no port designation is provided.

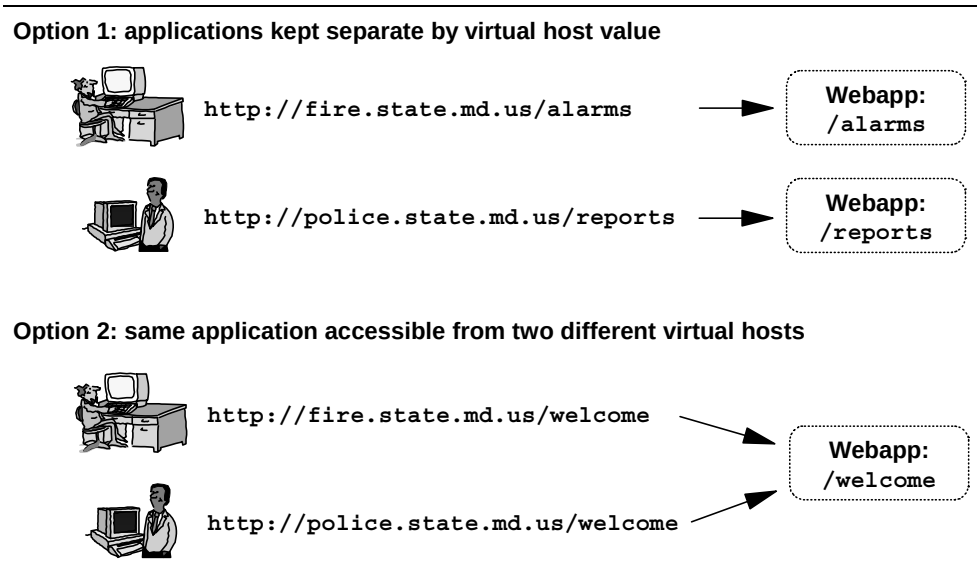
---

  2. Code *multiple virtual host values* on the same `host.<name>.alias=` property:

Example: `host.taxes.alias=taxes.state.md.us,police.state.md.us`

Note the comma that separates the two virtual hosts.

What's the difference? Option #1 is used when you wish to keep your applications separated by virtual host and no application is accessible by more than one virtual host. Option #2 is used when you wish to allow an application to be accessible from multiple virtual hosts:



Two ways of coding multiple virtual host aliases in the webcontainer.conf file

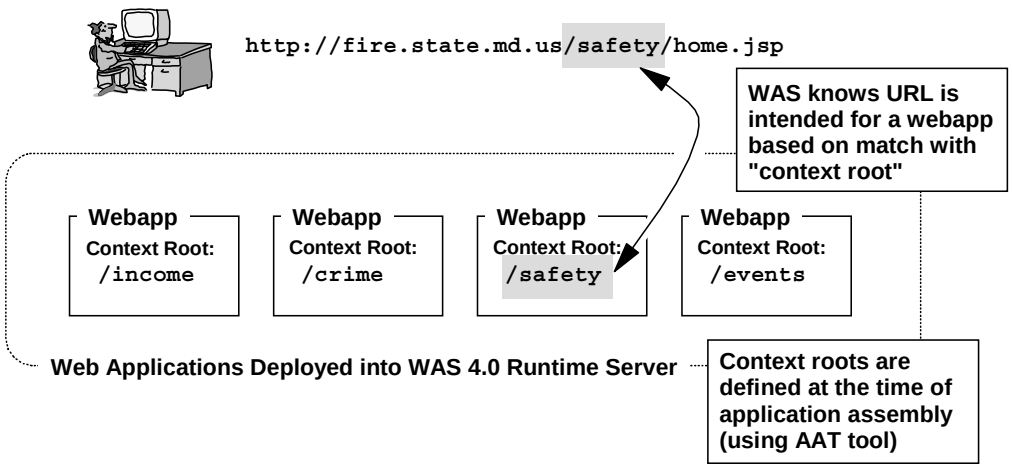
This topic makes more sense when you read and understand "Background: binding applications to virtual hosts" on page 19.

Note: You should code your virtual host values in the `webcontainer.conf` file in lowercase. It appears the URL received from the browser is folded into lower-case and then compared against the value found in `webcontainer.conf`. If your virtual host IP name is coded in uppercase in `webcontainer.conf`, your request may not be honored..

Note: The webapp support of WAS 4.0 had a problem initially when the default port 80 was used. PTF UQ57590 addresses that problem (APAR PQ50839). Your copy of WAS 4.0 may not have that fix. So if you're in test mode and things don't work with the default HTTP listen port of 80, change the `Port` directive in your `httpd.conf` file to something like `8080` and then code the `8080` in the `webcontainer.conf` file's virtual host alias statement and try again.

Background: binding applications to virtual hosts

When a web application is deployed into a WAS 4.0 web container, one of the properties set for the webapp is a "context root." This is a string of text that will be matched up against the received URL to determine which web application is being requested. For those familiar with the WAS 3.5 Standard Edition environment, a "context root" is analogous to the "rooturi" value:

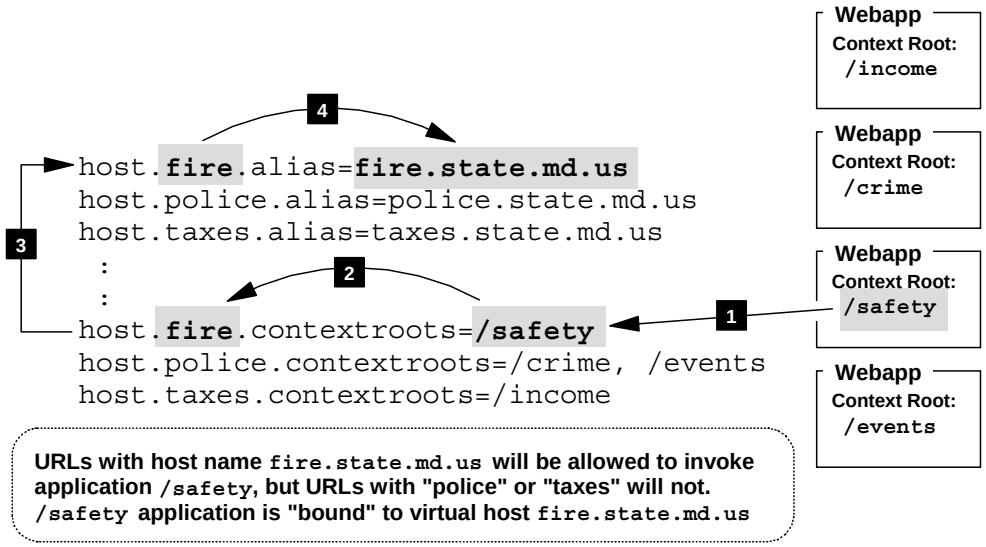


Context roots are used to associate received URL with an application

The picture above illustrates how a URL is associated with a webapp, but it does not show how the application is bound to a virtual host. That is done by making a match between the "context root" value set for the webapp and the values found on the following property in the `webcontainer.conf` file:

`host.<virtual_host_alias_name>.contextroots=`

If the "context root" as set in the deployment descriptor of the application matches the value found on the `contextroots=` statement in the `webcontainer.conf` file, then that application is bound to the virtual host named on that statement:

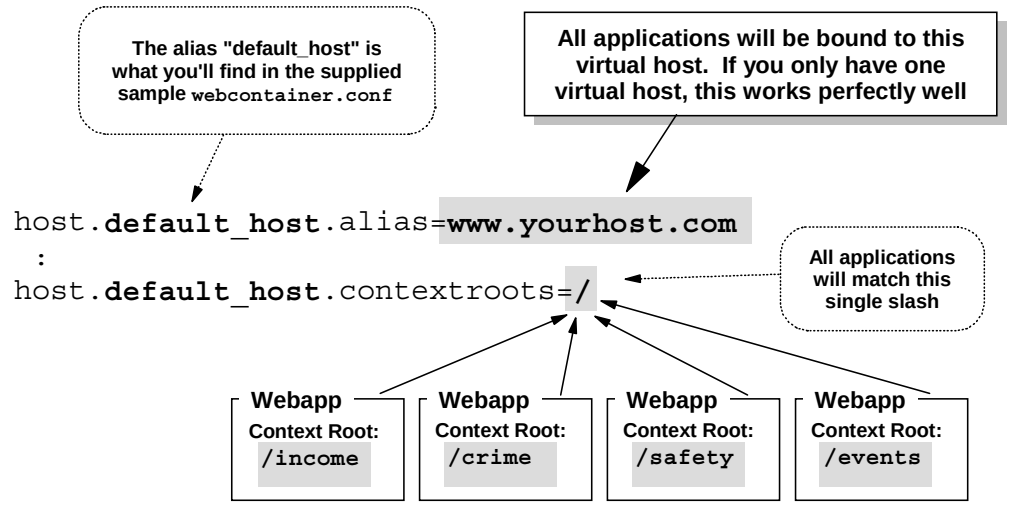


Context root match and the binding of application to virtual host

This picture shows a few things you should note:

- Multiple `host.<name>.alias=` properties are allowed in the `webcontainer.conf` file
- Multiple `host.<name>.contextroots=` properties are allowed in the `webcontainer.conf`
- Defining more than one string per `host.<name>.contextroots=` property is allowed. This allows you to bind multiple applications to the same virtual host (see "police" example above).

What's not being shown in that picture is something that'll simplify this whole "virtual host" and "context root" issue quite a bit: *a single slash on the* `host.<name>.contextroots=` *property will allow* ***all*** *web applications to bind to that virtual host:*



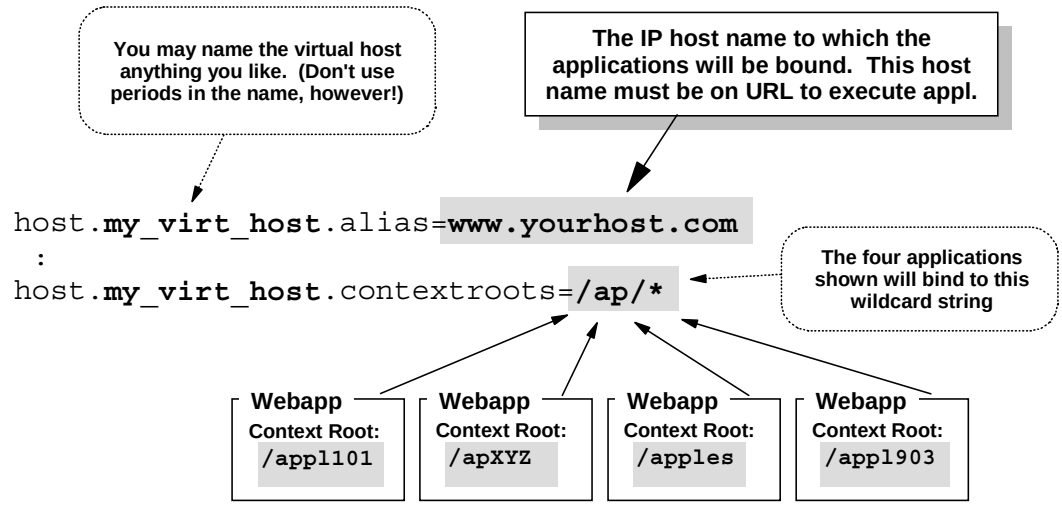
The "catch-all" contextroots value allows all webapps to bind to it

Using the "catch-all" single slash is the best way to start. If your plans do not call for having multiple IP hosts serviced by your one server, this works well.

Background: using wildcards in the contextroots= values

You have seen how an explicit coding of a `contextroots=` value will allow an application to "bind," and you have seen how the "catch all" value of a single slash (`/`) will allow all applications to bind. Now let's explore how you can incorporate the use of wildcards into your `contextroots=` strings. Wildcards allow you to bind multiple applications with similar names to a virtual host while avoiding the use of the "catch all" single slash.

The wildcard character is the asterisk (`*`). However, there's a twist to this that you must understand: *you must precede the wildcard character with a forward "slash" (/) so the web container can know a wildcard comes next*. This is best illustrated with an example:



Using wildcards in the contextroots= string

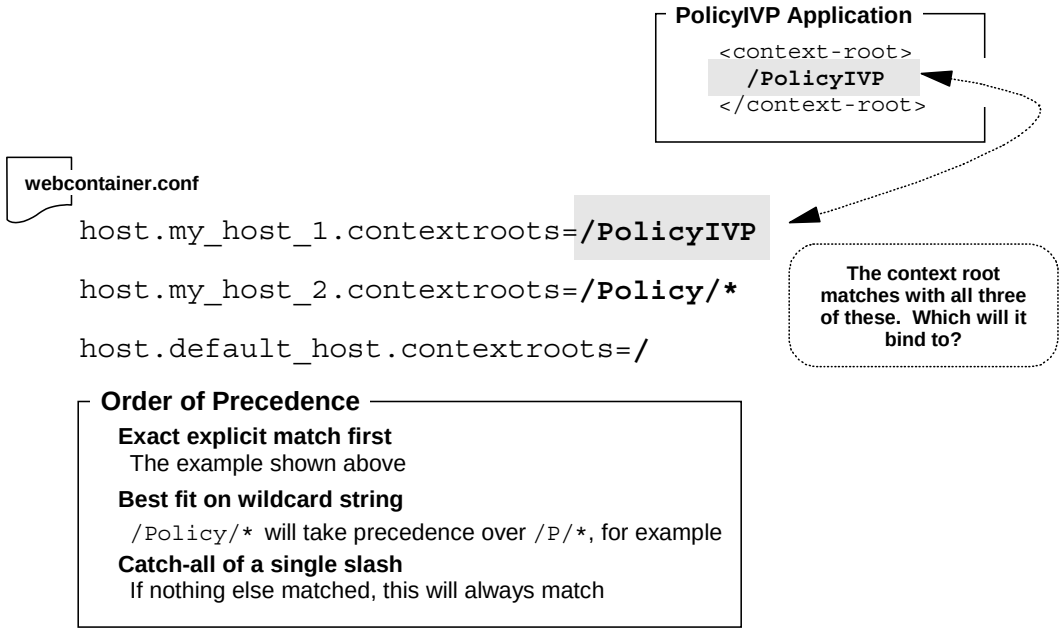
In this example, the `contextroots=` value of `/ap/*` serves as a wildcard template that will "catch" any application `<context-root>` string that starts with the letters "ap".

(I can not explain *why* the preceding slash is required; it just is.)

Whenever the issue of wildcards is discussed, someone invariably asks the following two questions:

Question	*Answer*
May I use multiple wildcards?	No. Don't code a `contextroots=` string with multiple wildcards. The wildcard function isn't that sophisticated.
Is there a single-character placeholder wildcard?	No. Other programs that use wildcards permit the question mark (`?`) to serve as a wildcard for a single character. This is not permitted on the `contextroots=` statement.

The next natural thing to ask is this: *what's the order of precedence for the three types of contextroots= value coding (explicit vs. wildcard vs. catch-all)?* What happens if a `<context-root>` value for an application maps to more than one `contextroots=` value? The answer is this:



The order of precedence for matching an application to a context root value

Warning: avoid ambiguity in your contextroots= coding

When a WebSphere application server starts up, it looks at all the applications deployed within it and reads in all the context root settings. It then starts the process of comparing the application's context root against the values on the `host.<name>.contextroots=` properties in the `webcontainer.conf` file.

What happens if a web application's context root setting maps to two different `contextroot=` definitions? For example, imagine you have the following in your `webcontainer.conf` file:

```
host.default_host.alias=www.yourhost.com
host.my_host.alias=www.myhost.com
 :
host.default_host.contextroots=/PolicyIVP
host.my_host.contextroots=/PolicyIVP
```

And your application has a `<context-root>` value of `/PolicyIVP`. To which of the two virtual hosts would your application bind? Both?

The answer is this: it'll bind to only one, but when coded like this, *not both*. If your objective is to bind `/PolicyIVP` to two different virtual hosts, that's done by using a single `contextroots=` definition, and coding two different virtual host values on the associated `alias=` definition. See "Background: binding an application to multiple virtual hosts" on page 23.

Follow this simple rule: *do not code the same value on two different `contextroots=` definitions*. That introduces ambiguity into your `webcontainer.conf` file. If you want to bind an application to multiple virtual hosts, do that by coding multiple virtual hosts on a single `alias=` definition, as described next.

Background: binding an application to multiple virtual hosts

This is done by defining multiple hosts on a single `alias=` definition:

```
host.taxes.alias=taxes.state.md.us,www.taxes.are.us
```

The different host values are separated by a comma. With this done, an application that matches a `contextroots=` value will bind to both hosts defined on the statement.

Background: use of localhost value for virtual host

The value you code for the virtual host is typically something related to the host IP name of your system (for example, `wsc1.washington.ibm.com:8080`). There is another value that you could code: `localhost`. This is a form of "universal" virtual host; one that in theory allows any host name on the URL to use the application. The documentation advises *against* using this value.

Rule: Do not use the value `localhost` for any virtual host values in the `webcontainer.conf` file!

There are two reasons for this:

- The documentation advises against it, and
- If an application binds to the virtual host defined with `localhost`, there is no way you can get a request to flow from the plugin over to the web container. The plugin will flow the requests over to the web container only if there's an exact character-for-character match on the virtual host and context-root values. If there's a way to get this to work, I haven't found it. It is better to simply avoid the ambiguity and confusion and provide a hard-coded virtual host value.

Example: "PolicyIVP" application and its "context-root" setting

The WAS 4.0 product comes with a sample application that can be used to verify the environment. This application is commonly known as the "PolicyIVP" application, and it is supplied as a fully-assembled "EAR" file in the following location:

```
/usr/lpp/WebSphere/samples/PolicyIVP/ejb/PolicyIVP.ear
```

If you were to download that EAR file to your workstation and use a product like WinZIP® to open the file, you'd see, among other things, a file called `application.xml`. That is the "deployment descriptor" for the assembled application, and was created by the assembly tool (AAT). View the contents of `application.xml` and you'll find the following:

```
<web>
 <web-uri>PolicyWebApp.war</web-uri>
 <context-root>/PolicyIVP</context-root>
</web>
```

When the WAS 4.0 server starts it'll read in the deployment descriptors of all the applications deployed into the server, and it'll see that `<context-root>` value for PolicyIVP application.

That sets the "context root" for the application. All that's left is to provide a `host.<name>.contextroots=` property in the `webcontainer.conf` file to allow this application to bind to a virtual host. The easiest, as just described, would be:

```
host.default_host.contextroots=/
```

But you could also use:

```
host.default_host.contextroots=/PolicyIVP
```

However, an explicit coding like this would allow only the PolicyIVP application to be bound to the "default_host" virtual host alias. If that's the *only* application in the server, then it is acceptable. However, if another application came along -- let's assume with a context root of `/Sample1` -- you would need to add that context root to the property:

```
host.default_host.contextroots=/PolicyIVP, /Sample1
```

Multiple values are allowed on the statement. With only one virtual host defined, however, it makes things quite a bit easier to code just the single slash and allow all applications to map to it.

Activity: defining context roots in webcontainer.conf

Back in "Activity: defining a virtual host in the webcontainer.conf file" on page 18 you defined the virtual host in the `webcontainer.conf` file. Now do the following:

- ☐ Edit the `webcontainer.conf` file. Locate the property `host.default_host.contextroots=` near the bottom of the file.
- ☐ Add a single slash to the statement:

  ```
  host.default_host.contextroots=/
  ```

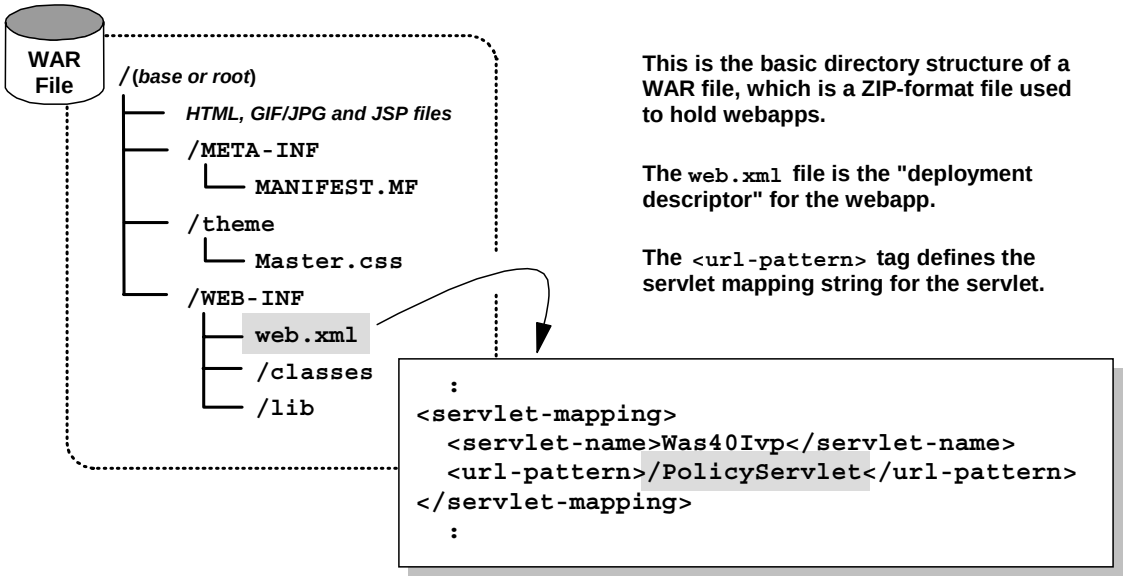
  Note: The name "default_host" in this example is what's provided in the sample `webcontainer.conf` file. You may continue to use that value, or provide one of your own. The key is having the value you choose be equal on both the `contextroots=` property and the `alias=` statement to which you wish the webapp to bind. Case matters, so type carefully.

- ☐ Save the file
- ☐ Stop and restart your application server.

Background: the "servlet mapping" value of an application

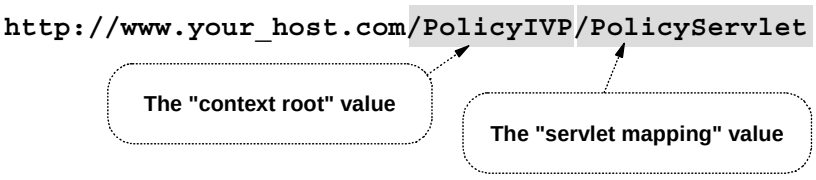
With an application successfully bound to a virtual host with the "context root" setting in both the `application.xml` file and the `webcontainer.conf` file, the final piece of the puzzle is the setting that connects the URL to a *specific servlet* to execute. A given "context root" such as `/PolicyIVP` only gets you to the proper web container; something called the *servlet mapping* is what relates the URL to the specific servlet.

The servlet mapping value is something set at the time the web application is developed. It is *not* something you configure into your `webcontainer.conf` file or any other configuration file related to the WAS 4.0 server or the plugin. The servlet mapping string for a servlet is contained in the deployment descriptor for the webapp, which is a file called `web.xml` and is part of the WAR file:



Servlet mapping string inside webapp's deployment descriptor

In this example the string `/PolicyServlet` is the servlet mapping string. What that means is that the URL to invoke this application is the following:



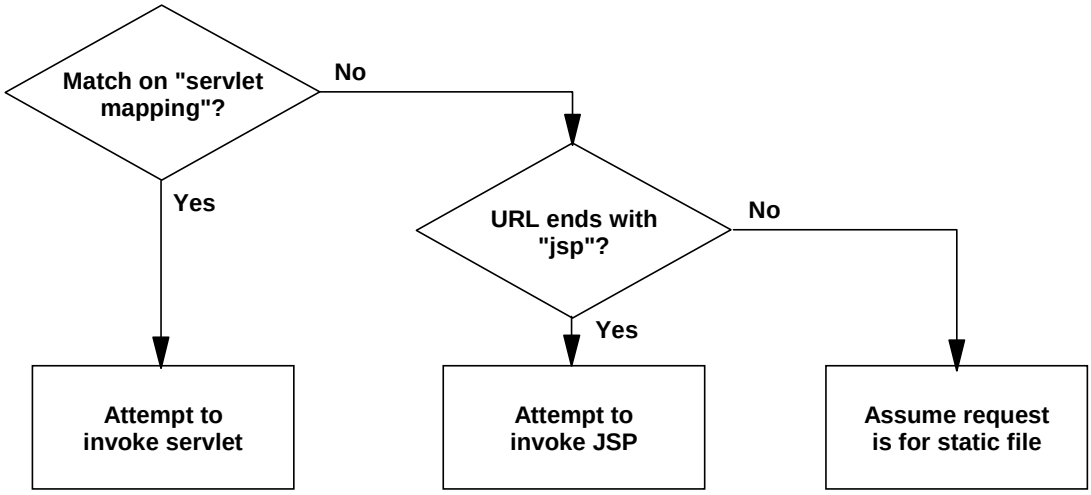
URL to invoke PolicyIVP's servlet

The context root (`/PolicyIVP`) gets you to the proper web container; the servlet mapping (`/PolicyServlet`) gets you to the proper class file for the servlet code to run. If you go back to "Background: the basics of serving out a web applications" on page 1, this provides the "basic function" tagged with #3 in the diagram.

The servlet mapping is part of the webapp development and *not* something you code into the `webcontainer.conf` file or specify at the time of application assembly with the AAT tool.

Background: WAS 4.0 serving of static files and JSPs

A web application may consist of files other than servlets, such as JPG/GIF image files, HTML files and JSP pages. WAS will happily serve those files out as long as it can figure out that the URL request is for that kind of file. Here's how WAS figures that out:



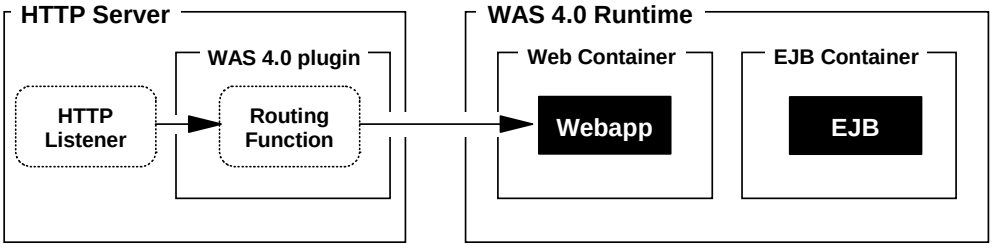
Process by which WAS determines if request is for servlet vs. JSP vs. file

Notice what happens: if the URL does *not* match on a "servlet mapping" known to the web container, WAS will work down through its logic and default to considering the request to be for a static file. Requsts for static files are served by a built in function of WAS known as the "SimpleFileServlet."

This is important because you may very well encounter a problem were the servlet mapping string in the `web.xml` file isn't what you enter on the URL. When that happens, you see problem as described in "Servlet mapping string doesn't match" on page 63.

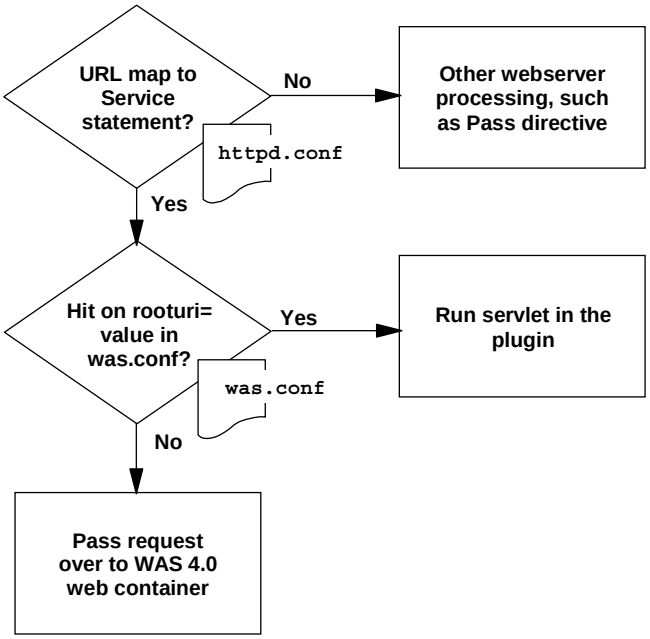
Background: the role of the WAS 4.0 webserver plugin code

The WAS 4.0 plugin code is a requirement to running a webapp in the web container environment of the WAS 4.0 runtime. This is because the WAS 4.0 runtime does not at the present time have an HTTP listener, so the webserver has to serve that role. The WAS 4.0 plugin provides an environment to which the webserver can pass a request for webapp execution. The WAS 4.0 plugin then routes the request over to the WAS 4.0 runtime web container for application execution:



WAS 4.0 plugin routing request over to the WAS 4.0 runtime web container

But what tells the WAS 4.0 plugin whether to run the webapp locally or route the request over to the WAS 4.0 runtime? It depends on the definitions in the local `was.conf` file used by the plugin. If the plugin finds a `deployedwebapp.<name>.rooturi=` statement whose value matches the received URL, it'll try running the webapp locally. If it can't find a definition for the requested application in the `was.conf` file, it'll route the request over to the WAS 4.0 runtime. The following flowchart illustrates the logic of this:



Logic employed to determine where to run the web application requested

Note: Truth is the WAS 4.0 plugin is a bit smarter than this. It maintains communication with the WAS 4.0 runtime to know what applications are bound to what virtual hosts. If the requested application isn't defined in the `was.conf` and isn't bound to a virtual host over in the web container environment, the plugin will issue out an error message on behalf of the WAS 4.0 runtime. It won't route requests over to the WAS 4.0 runtime it knows isn't defined over there. See "Activity: check plugin Application Dispatching Information" on page 29 to read how to determine what applications the plugin knows about. See also "Advanced Webapp Topics" on page 91 for information on changing the behavior of this "communication" conducted by the plugin.

Question: do I still need a webcontainer.conf with the new Transport Handler?

As mentioned earlier, yes. Everything that's been discussed in the last several pages regarding `webcontainer.conf` files, virtual hosts, etc., still applies regardless of the HTTP listener employed -- plugin or Transport Handler.

Activity: restart the servers

Do the following to refresh your environment:

- ☐ Stop the application server instance. This is necessary to pick up the changes to the `jvm.properties` file and its pointer to the new `webcontainer.conf` file.
- ☐ Stop the webserver. This is not strictly required if you have restarted it after making the changes to the `httpd.conf` and `httpd.envvars` detailed in "Initial Configuration of the Webserver Plugin Code" on page 11, but it doesn't hurt to do it again here "just to be sure."
- ☐ Restart the webserver and restart the application server instance.

Validation and Basic Debugging

The most basic form of validation is to point your browser at the webserver and issue the URL used to invoke the application. But there are a few things you can do before that to insure success, and there's a methodology used afterwards when things don't work.

Background: preliminary validation

The following two "activities" help you determine if things are in proper working order prior to issuing a URL against the system. There's no point in testing it if a fundamental piece of the puzzle is broken.

Activity: check server region SYSPRINT

The `SYSPRINT` of the application server region (not the control region) has two pieces of key information that'll tell you if things are working okay:

- An indication of what `webcontainer.conf` file is in use

  Why you should care: If the server can't locate the `webcontainer.conf` you pointed to in the `jvm.properties` file, it'll take the default `webcontainer.conf` file located in the `/usr/lpp/WebSphere/bin` directory. If that happens, your virtual host won't be defined and things won't work. *The server can fail to find your* `webcontainer.conf` *file with something as small as a typo in the long directory pointing to the private directory of the server*.

- An indication of what applications are bound to what virtual hosts

  Why you should care: If for whatever reason the application you deployed didn't get bound to a virtual host, your request to run the application will fail. Failure to bind to a virtual host can occur if the string you provide on the `host.<name>.contextroots=` property doesn't match the `<context-root>` setting in the deployment descriptor for the application. If you code the single slash, this problem isn't likely to occur. But an explicit coding of the `contextroots=` property might lead to a mismatch.

Note: Checking the SYSPRINT should be done whether you've configured the plugin or the new Transport Handler. The issue of binding applications to virtual hosts is entirely separate from the issue of which HTTP listener is in the mix.

Do the following:

☐ Check the `SYSPRINT` of the server region. Near the top you'll find the following statement:

`Web Container:Configuration File Name:` <directory and file name of config file>

The key is to make certain the directory and file name is what you specified in the `jvm.properties` file. A minor typo will result in the server not finding your file. That'll result in the server taking the default in the `/usr/lpp/WebSphere/bin` directory.

☐ If the "Web Container Configuration File Name" doesn't point to your `webcontainer.conf`, then go back and check your `jvm.properties` file and make certain your pointer is exactly correct.

☐ Now look further down in `SYSPRINT` and locate the following string:

```
VirtualHost Web Application Context Root Bindings:
   /
```

Note: You'll find a separate block of information for each "context root binding" in the `webcontainer.conf` file. If you coded just the single forward slash, then you'll see only one such block of information. More "context root bindings" will result in more blocks of information presented in the `SYSPRINT`.

☐ Now scroll down just a bit within the block of information for the "context root binding" (the single slash in this example) and look for the following:

```
VirtualHost Bound Web Applications:
    Web Application Context Root: /PolicyIVP
```

This indicates which applications matched the "context root binding" value. In this example, the application with a context root of `/PolicyIVP` has matched the single slash.

So far, so good. But now you must make sure the application is bound to the virtual host you intended.

☐ Scroll down a bit further and look for the following:

```
VirtualHost Alias List:
     wsc4.washington.ibm.com:8080
```

This indicates the virtual host, as defined on the `host.<name>alias=` property in the `webcontainer.conf` file, to which the application has been bound. Make certain this is the virtual host you intended.

Activity: check plugin Application Dispatching Information

The WAS 4.0 plugin provides a program that will tell you what applications it sees bound to virtual hosts over in the WAS 4.0 runtime. This is a very handy way to verify that your plugin sees things the way you intended them to be.

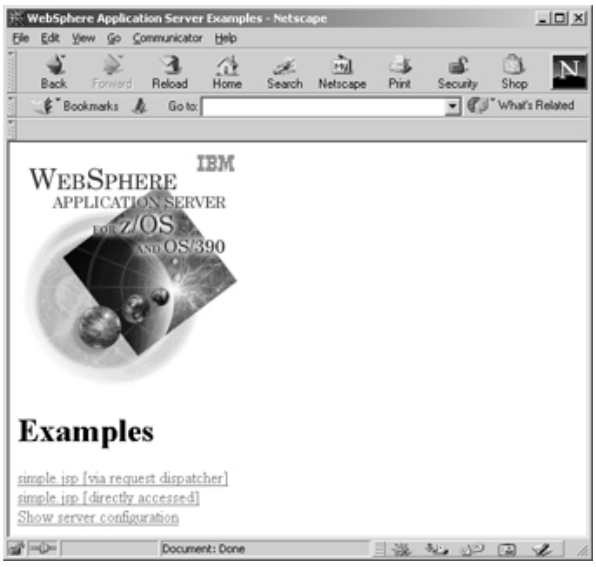
Note: The "Application Dispatching Information" is a function of the WAS 4.0 plugin code, and not related to the new Transport Handler.

Do the following:

☐ With the webserver and application server up and running, point your browser to:

```
http://<your host>/webapp/examples/index.html
```

That'll bring up a screen that looks like this:



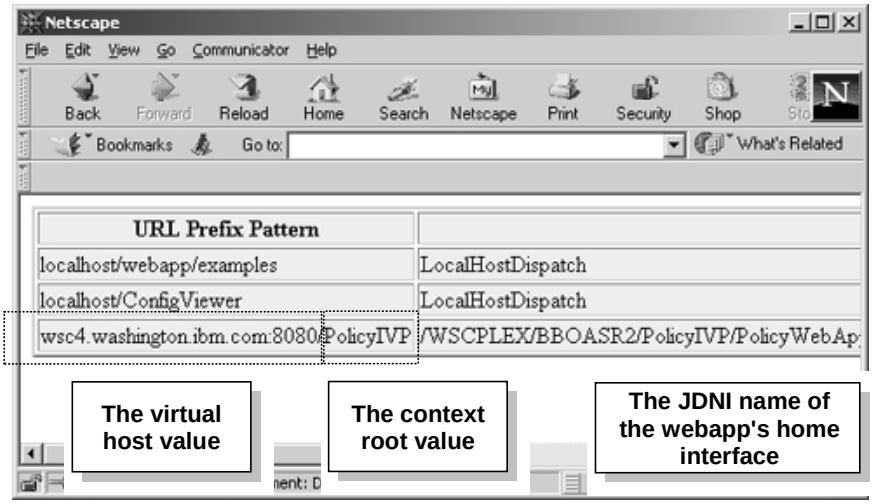
The "webapp examples" primary screen

Note 1: Receiving the screen shown above is a good way to validate that the plugin is working, but it does not *in itself* mean the web container configuration information is correct.

Note 2: See "Background: WebSphereSampleApp.ear shipped with WAS" on page 91 for

information on another application that comes with WAS that uses the same URL and produces a very similar string, but is run from the web container.

- ☐ Click on the "Show server configuration" link and when the next screen comes up, scroll down and find the heading "Application Dispatching Information." Click on that link. That'll bring up a screen that looks like this:



"Application Dispatching Information" screen

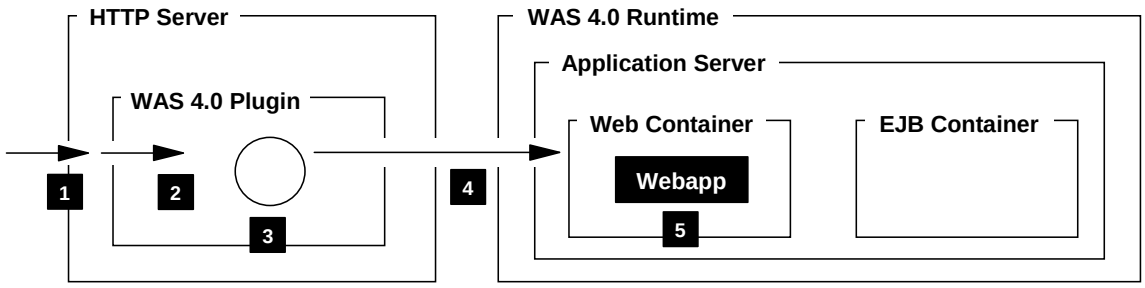
The appearance of your virtual host and your context root indicates the plugin has knowledge of your deployed application over in the WAS 4.0 runtime environment. If you see only the two "localhost" values, things aren't working right.

Note: The linkage between the plugin and the WAS 4.0 runtime is not instantaneous. If you just started your server region it may take a few moments (15 to 45 seconds sometimes) for the information about what webapps are in the web container to work its way over to the plugin. If you don't see your webapp immediately, give it a minute or so and hit the "refresh" on the screen.

Background: Basic Debugging

When something goes wrong (and something *always* goes wrong), there's a way you can methodically walk through the system and determine what's failing (or at least determine what's *not* failing). The following picture illustrates this methodology:

Note: This section was written to offer basic debugging of a configuration where the plugin is the HTTP listener. If you're using the new Transport Handler, the debugging becomes much easier because the Transport Handler is coupled to the web container much more tightly. So if you're using the Transport Handler, go to the section "The WAS 4.0.1 Transport Handler" on page 73 for information on that environment.



1. **Did your request reach the webserver?**
2. **Did your request get mapped to the WAS 4.0 plugin?**
3. **Did the plugin try to execute your request locally, or route it to WAS 4.0?**
4. **Did your request get routed over to the web container?**
5. **Are you able to drive any portion of your webapp?**

Basic debugging methodology

Activity: validate that your request reached the webserver

Typing a URL at the browser and hitting "enter" doesn't guarantee the request will hit the server you think it will. There best way to validate your request is getting to your server is to start the "vv" trace and see if your request is recognized by the webserver.

For example, a URL of:

```
http://<your_host>[:port]/webapp/examples/index.html
```

will show up in the "vv" trace as:

```
Client sez.. GET /webapp/examples/index.html HTTP/1.0
```

That validates that your request hit your webserver and is being acted upon.

Activity: validate that your request was mapped to the plugin

A request received by your webserver doesn't guarantee that request will be mapped to the WAS 4.0 plugin. That requires a properly coded `Service` statement. Here again, the "vv" trace validates the request being mapped over to the plugin.

Note: Of course, a properly coded `Service` statement is of little use if the plugin itself isn't initialized. You can make certain the plugin is up and going by reviewing the information found at "Activity: validation and basic debugging of plugin" on page 13.

Again, a URL of:

```
http://<your_host>[:port]/webapp/examples/index.html
```

with a `Service` statement of:

```
Service /webapp/examples/*  /usr/lpp/WebSphere/WebServerPlugIn/...
```

will show the following in the "vv" trace when the request is mapped to the plugin:

```
Service..... /webapp/examples/* matched "/webapp/examples/index.html" ->
  :
Pattern..... match SUCCEEDED.
  :
APIClassExec Calling function "service_exit"
```

That validates that your request was mapped over to the plugin. If you see your request mapping to a `Pass` statement, then the *plugin* is not coming into play.

Activity: validate that the plugin isn't trying to run the webapp locally

The WAS 4.0 plugin will attempt to run a request *locally* whenever it gets a match on a "rooturi" statement in the `was.conf` statement. This relationship is illustrated in the figure "Logic employed to determine where to run the web application requested" on page 27. (The truth is it checks the "rooturi" as well as the virtual host. How all that works is beyond the scope of this document since the objective here is to run webapps over in the WAS 4.0 web container, not the plugin.)

The best way to check is to review your `was.conf` and make sure no `deployedwebapp rooturi` statement is defined that will map to your inbound request. If it maps, the plugin will try to run the request locally. Remember: it is the *absence of definitions* in the `was.conf` that means it'll try to map the request over to the WAS 4.0 runtime.

The other way to validate this is by seeing if the request is mapped to the WAS 4.0 runtime. That is explained next.

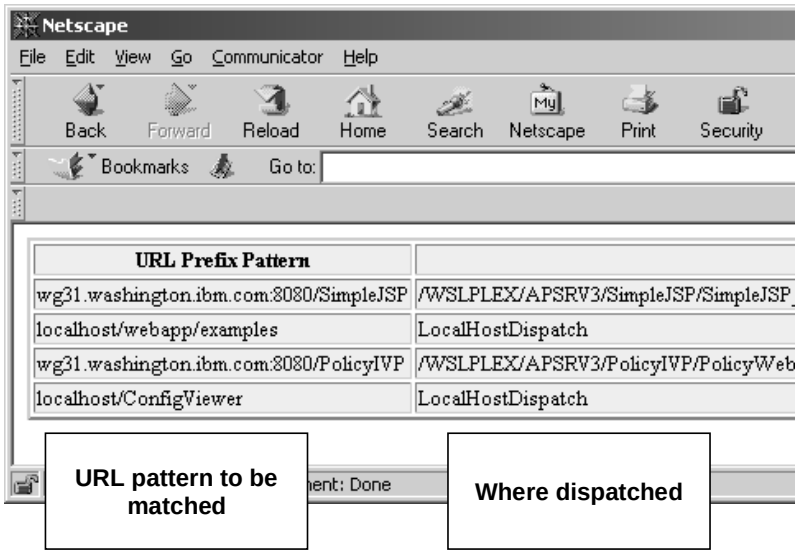
Background: how the plugin determines if a request is to be sent to WAS 4.0 runtime

A request received by the plugin will get mapped over to the WAS 4.0 runtime only when the plugin matches the received URL to a known webapp over in the web container. This matching is done in the WAS 4.0 plugin's code and uses something called the "String Matcher Table." This "string matcher table" is kept in memory, and is constructed using information the plugin sees in the local `was.conf` file as well as information it receives from the web container about webapps deployed there.

Here's an example of the table, taken from the "ncf" log of the plugin when `appserver.loglevel=WARNING` at a minimum is set in the `was.conf` file.

```
String Matcher Table:
=====================================
/wg31.washington.ibm.com:8080/SimpleJSP/* --> (remote web container JNDI home)
/localhost/webapp/examples/* --> LocalHostDispatch
/wg31.washington.ibm.com:8080/PolicyIVP/* --> (remote web container JNDI home)
/localhost/ConfigViewer/* --> LocalHostDispatch
```

This string matcher table will contain the *same information* you'll find in the "Application Dispatching Information" panel (see "Activity: check plugin Application Dispatching Information" on page 29):



Application Dispatching Information is the same as what's in "ncf" trace "string matcher table"

This is important because you don't need to crawl through the "ncf" trace to validate what webapps the plugin knows about: you can use the "appliction dispatching" information as well. You do have to look in the ncf trace to see for certain that a request was properly mapped. That is discussed next.

Key: For the sake of simplicity the process is often described as, "the plugin will look for a local definition in `was.conf`, and if not there then send it over to the web container." The truth is the plugin won't send the request unless it matches a known list of webapps in the web container. This list is the "string matcher table." With that understanding, you may now proceed to validating that your request is getting mapped to the WAS 4.0 runtime's web container.

Activity: validate that request has been mapped to WAS 4.0 runtime

There are two ways you can validate that a request was mapped to the WAS 4.0 runtime:

- Perform a visual comparison of URL against information in "Application Dispatching Information" (that proves nothing, but it'll weed out obvious errors of typing and such)
- Enable tracing of the plugin and interrogate the trace file for evidence of the request getting mapped.

It is the latter that will be covered here. Do the following:

☐ Edit the `was.conf` file and set the following properties:

```
appserver.tracelevel=com.ibm.*=all=enabled
appserver.loglevel=WARNING
appserver.logdirectory=(directory to which logging will occur)
```

Note: Setting the `appserver.tracelevel` to `all=enabled` like this produces a *tremendous* amount of output. You would never have this running on a production system as it would drain away too much system resource. If you set this property on your test system, remember that is set and turn it off (comment out the line and restart the webserver) after you have done your debugging. Otherwise, you will likely quickly fill the HFS in which the logging is done.

☐ Stop and restart the webserver.

☐ Verify that the plugin initialized (see "Activity: validation and basic debugging of plugin" on page 13).

☐ Clear your browser cache

☐ Issue the URL that produces the failure indication

☐ Browse the "ncf" log, which should be quite large with tracing enabled

☐ Issue a "find" command on the URI you issued. For example, if your entire URL was:

```
wg31.washington.ibm.com:8080/SimpleJSP/simple.jsp
```

then do a find on only the `/SimpleJSP/simple.jsp` portion of it.

☐ When you find the first occurrence of the string, scroll down just a bit. You should see the "string matcher table" as well as the "Basic Rules" and "Exact Rules" tables. If the plugin sees a match on the URI vs. the "string matcher table," you should see the following immediately afterwards in the trace:

```
WS390Redirect < localDispatch
WS390Redirect D Matched JNDI Name : "/WSLPLEX/APSRV3/SimpleJSP/...
InProcNativeS D ConnectionStub.getRequestURI: instance = 2
WS390Redirect D Remotely Dispatching Request URI "/SimpleJSP/simple.jsp"
```

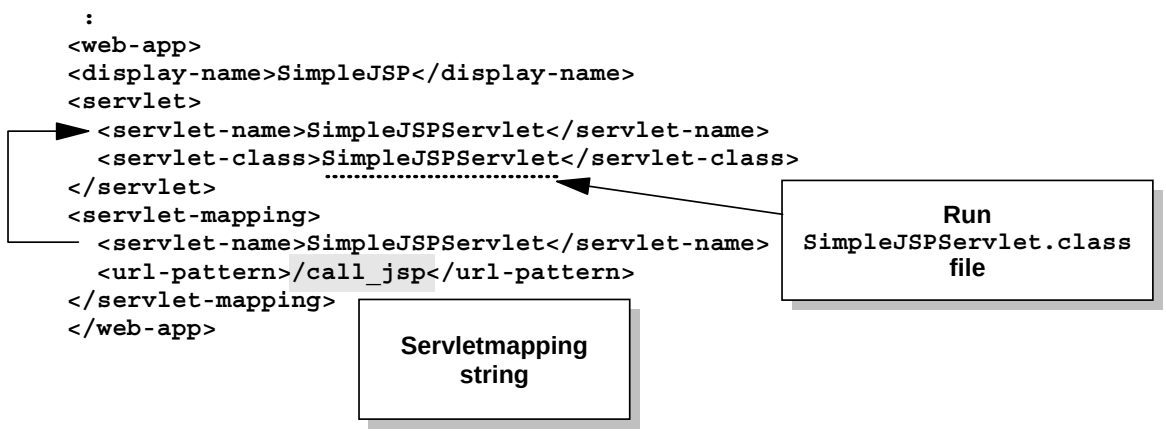
What this means is that the request (`/SimpleJSP/simple.jsp` in this example) matched an entry in the string matcher table, the plugin was able to determine the JNDI name from the table, and that the plugin is dispatching the request remotely.

After this there is a great deal of trace activity showing the plugin flowing requests to invoke the home interface of the remote web container. All that will augment the validation provided by the "Remotely Dispatching Request URI" message. Your request has been sent to the web container.

Background: the servlet-mapping string and execution of webapp class files

The activity just discussed simply validates that your request got to the web container. But that does not guarantee that it'll execute a web application. That's because the specific webapp to be executed is implied with the *servletmapping string*, which is part of the URL to be sure, but not something the plugin worries about. So a request you send could have the correct context-root, which will allow the plugin to map the request to the runtime, but have an incorrect servletmapping, which will result in an error.

The servletmapping string is defined in the deployment descriptor for the webapp (the `web.xml` file in the WAR file), and has an XML tag of `<url-pattern>`:



Servletmapping string in the webapp deployment descriptor

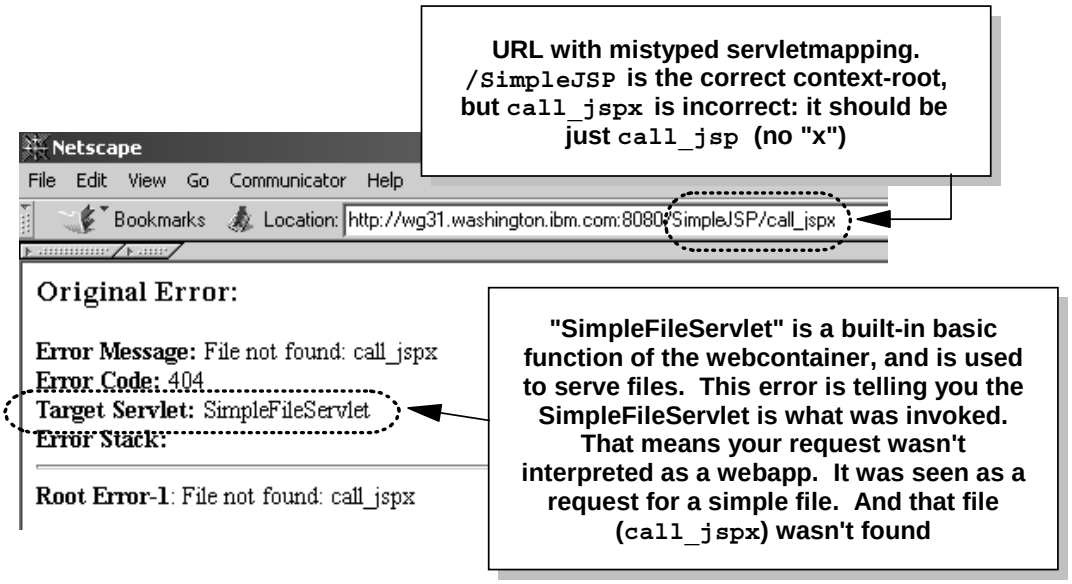
Note: If you're interested in where this "SimpleJSP" example is coming from, see "Example: SimpleJSPServlet from WAS 3.5 Standard Edition" on page 43.

The interesting thing is this: if the web container doesn't see a match with a defined `<url-pattern>` servletmapping string, it'll then determine if the request is for a JSP. It does this by checking the end of the URI for a string of `.jsp`. If the URI doesn't end in `.jsp`, the web container will assume the request is for a simple file and go looking for a file to serve out.

Key: If you accidentally mistype the servletmapping value on the URL, but the context-root value is proper, the request will flow over to the web container. But the web container won't find any webapps with a servletmapping equal to your garbled URL value, so it'll eventually consider it a request for a simple file. The error you get on the browser screen will tell you which file type the web container was trying to service. Those are described next.

Background: key error indicators found on the browser screen

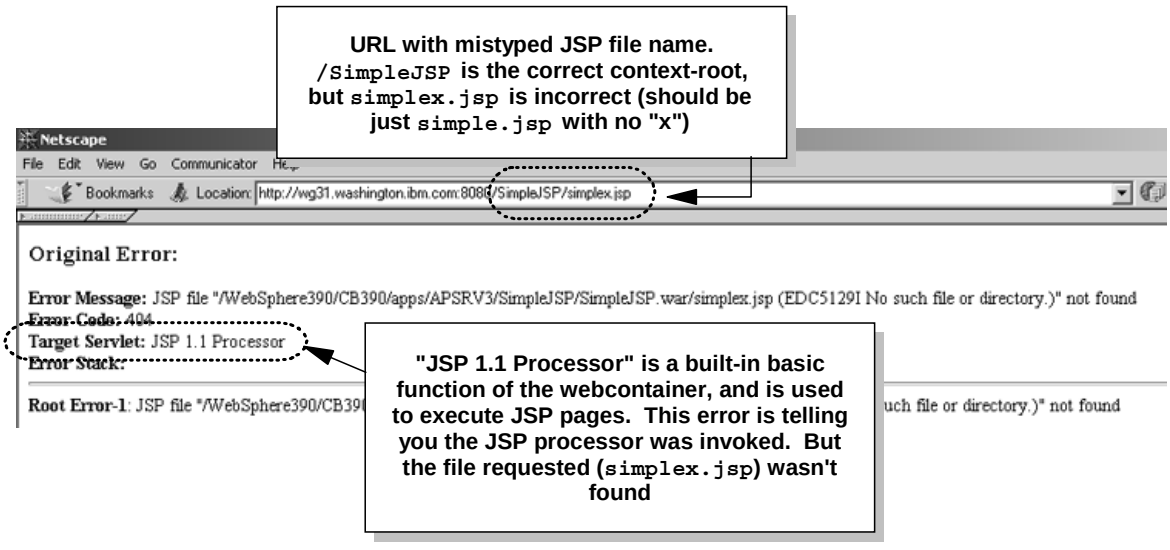
If your URL has the proper context root value and the request makes it over to the web containter, there are plenty of opportunities for errors. The first step is to look at the error message on the browser to narrow the possibilities. Consider the following examples:



Example A: URL with mistyped servletmapping string

In Example A, the person issuing the URL fat-fingered the servletmapping value and provided an extraneous "x" at the end. The plugin properly interpreted the correctly typed context-root value of SimpleJSP and dutifully passed the request over to the web container. But the web container looked through its set of known servletmapping values and didn't find a `call_jspx` servletmapping string, so it assumed the request was for a simple file. Having failed to find `call_jspx` in the root of the WAR file in the HFS, it gave up and issued the "File not found" error message.

Now consider an example where a JSP is requested directly, but the person issuing the URL mistypes the JSP name. They get the `.jsp` extension right, but fail to properly type the first part of the name correctly:



Example B: URL with mistyped JSP file name

So the bottom line is this: if what you're trying to do is invoke a servlet and you see either the SimpleFileServlet error message or JSP 1.1 Processor error message, you know you've probably got something wrong with your servletmapping string.

Activity: determine if you can serve any portion of your webapp

Webapps consist of not just servlet class files, but static files like HTML and JPG/GIF files, and JSP pages. You should be able to request and be served those files directly. To insure your path to the webapp in the web container is open, you could request for example an image file with the following URL:

`http://<your host>/SimpleJSP/banner.gif`

Note: The GIF file `banner.gif` is part of the "SimpleJSP" example illustrated in "Example: SimpleJSPServlet from WAS 3.5 Standard Edition" on page 43.

If you can get the GIF, and you're certain that GIF is not being served by the HTTP Server directly (validate this in "vv" trace), then you know the following things have worked: `Service` in `httpd.conf`; "string matcher table" function in WAS 4.0 plugin; receipt of request by web container in WAS 4.0; recognition of request as simple file; and the locating and serving of the file itself.

Activity: validate request results in execution of desired webapp class file

So how can you validate that your webapp was in fact run in the web container? By routing the `TRACEALL=1` output to `SYSPRINT` and looking at the results. Do the following:

- ☐ Edit the `current.env` file of your application server instance and insure the the `TRACEALL` property is set to at least 1.
- ☐ Set `TRACEBUFFLOC=SYSPRINT`
- ☐ Stop and restart your application server
- ☐ Clear your browser cache
- ☐ Issue the URL that you wish to test
- ☐ Review the contents of the application server instance's `SYSPRINT`. What you will find is something that looks like this (example shown is for `SimpleJSPServlet` as illustrated in "Example: SimpleJSPServlet from WAS 3.5 Standard Edition" on page 43).

```
Trace: 2001/09/12 18:09:47.991 01 t=8E15C0 c=1.27 key=P8 (13007002)
  FunctionName: com.ibm.servlet.engine.webapp.ServletInstance
  SourceId: com.ibm.servlet.engine.webapp.ServletInstance
  Category: AUDIT
  ExtendedMessage: Loading.servlet:."SimpleJSPServlet"
Trace: 2001/09/12 18:09:48.066 01 t=8E15C0 c=1.27 key=P8 (13007002)
  FunctionName: com.ibm.servlet.engine.srt.WebGroup
  SourceId: com.ibm.servlet.engine.srt.WebGroup
  Category: AUDIT
  ExtendedMessage: [Servlet.LOG]:."SimpleJSPServlet: init"
Trace: 2001/09/12 18:09:48.090 01 t=8E15C0 c=1.27 key=P8 (13007002)
  FunctionName: com.ibm.servlet.engine.webapp.ServletInstance
  SourceId: com.ibm.servlet.engine.webapp.ServletInstance
  Category: AUDIT
  ExtendedMessage: Servlet.available.for.service:."SimpleJSPServlet"
```

Contents of SYSPRINT showing loading and making ready of a servlet in the webcontainer

With this indication appearing in the application server instance's SYSPRINT, you know the request has been received by the web container, recognized and the servlet class file is being loaded.

Example: PolicyWebApp in the PolicyIVP Application

Note: This section applies to both the plugin environment as well as the new Transport Handler environment, except for any references to configuring the web server or the plugin. But all the application stuff and the web container configuration information applies to both environments.

Overview of the application

The WAS 4.0 product ships with a sample verification program called "PolicyIVP." Most people, when first testing their WAS 4.0 environment, use the "fat client" to drive the application. The fat client is a Java program run from the OMVS enviornment that exercises both the CMP and BMP bean found in the PolicyIVP application:

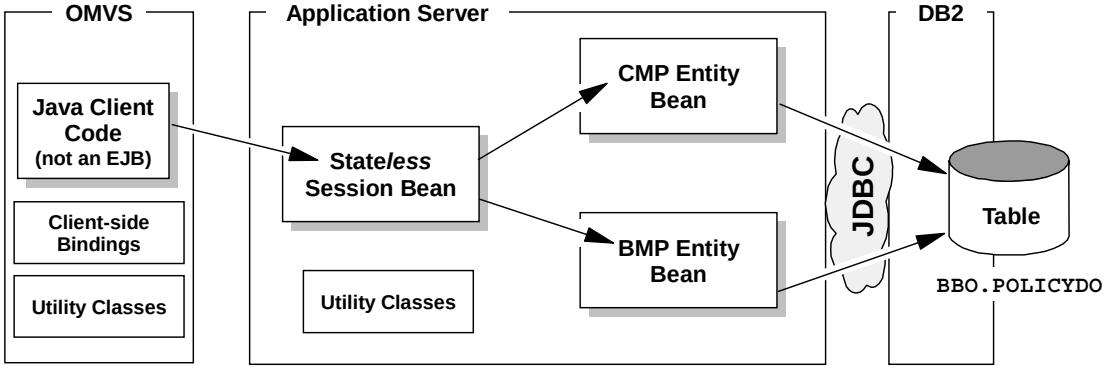
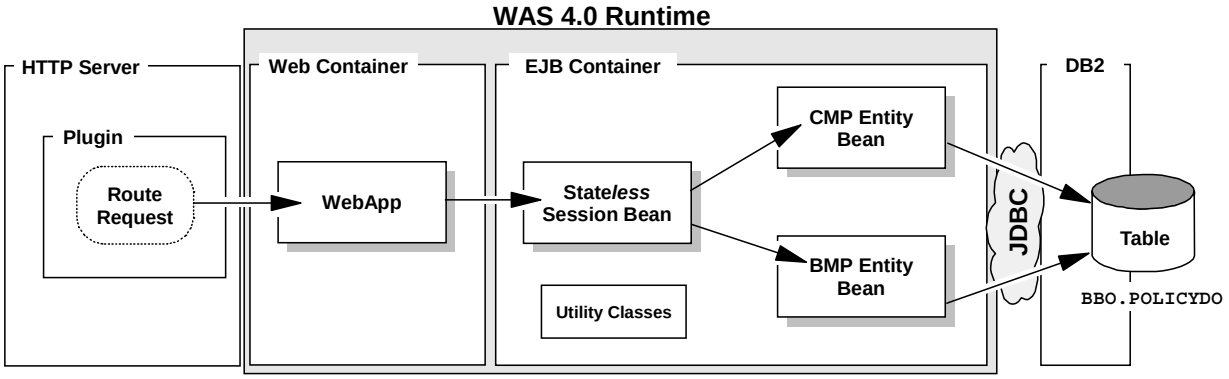


Diagram of PolicyIVP sample application when driven by "fat client"

There is no webserver or web container involved when the PolicyIVP application is driven by the fat client. However, the PolicyIVP application also comes with a *servlet* client, and when you configure the web container and plugin to run the servlet, the picture of PolicyIVP then becomes:



PolicyIVP when driven by the PolicyWebApp servlet client

The behavior of the beans is the same; the client code used to drive that behavior is different.

Background: deployment descriptor for PolicyIVP application

The `PolicyIVP.ear` file has within it a "deployment descriptor" (an XML file) that provides information about the beans and the webapp contained within the EAR file. This XML file is generated by the AAT tool at the time of application assembly. Of particular interest to this topic is the `<context-root>` tag in the XML file. The value named there is what WAS uses when it tries to bind the application to the virtual host:

```
<application>
  <display-name>PolicyIVP</display-name>
  <module>
    <ejb>policysession_deployed.jar</ejb>
  </module>
  <module>
    <ejb>policycmp_deployed.jar</ejb>
  </module>
  <module>
    <ejb>policybmp_deployed.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>PolicyWebApp.war</web-uri>
      <context-root>/PolicyIVP</context-root>
    </web>
  </module>
  <module>
    <ejb>PolicyWebApp_WebApp.jar</ejb>
  </module>
</application>
```

XML generated by the AAT tool

The EJB JAR files referenced

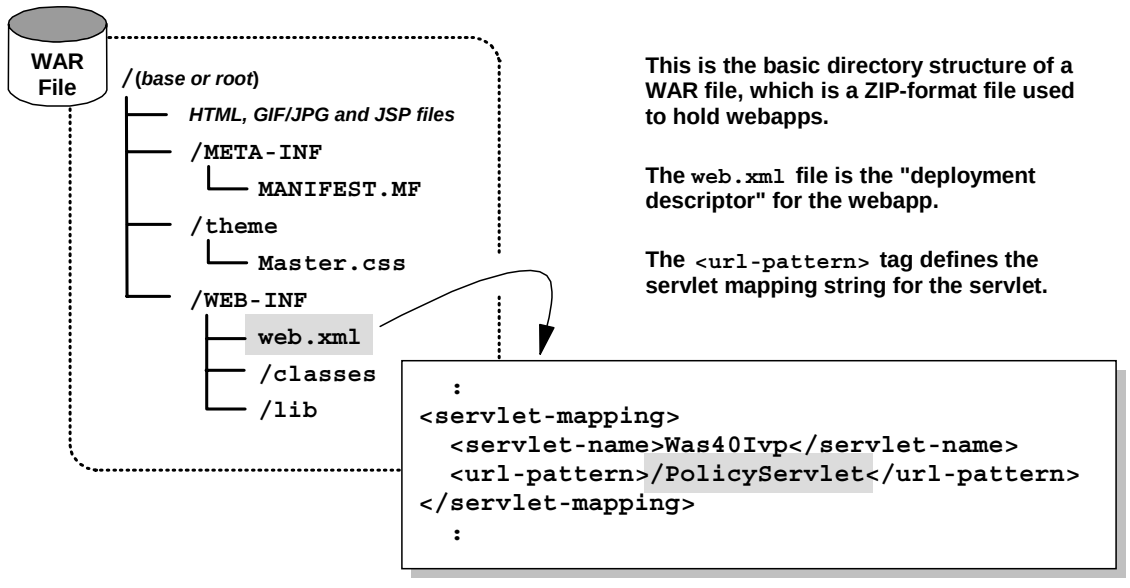
The context root for the webapp

The "appliciation.xml" deployment descriptor in PolicyIVP.ear file

The `PolicyIVP.ear` file is a ZIP-format file, and a tool like WinZIP can be used to look inside of -- and extract files from -- the EAR file. If you wish to see the `application.xml` file's content for yourself, use WinZIP and pull the file out.

Background: deployment descriptor for PolicyWebApp webapp

The `PolicyIVP.ear` file has within it a file called `PolicyWebApp.war`, which is another ZIP-format file that holds the webapp. The WAR file has within it a deployment descriptor that describes the webapp. Of interest is the servlet mapping value, which when appended to the context root value, provides the web container knowledge of which specific webapp to run. The WAR file can be accessed using WinZIP as well, and if you looked inside the `PolicyWebApp.war` file, you'd see something like the following:



The layout of the PolicyWebApp.war file and the servlet mapping inside the deployment descriptor.

So now you have the two key strings used to locate and run the webapp:
`<context-root>` in the `application.xml` file sets the context root value.
`<url-pattern>` in the `web.xml` file sets the servlet mapping value.

Configuration

Note: This example will show the WAS 4.0 plugin being used. The application is deployed into the BBOASR2 application server. The webapp will be deployed into the WAS 4.0 web container. The port on which the HTTP Server is listening is 8080.

Example: httpd.conf configuration

The updates to `httpd.conf` required to support the PolicyIVP application are:

```
ServerInit /usr/lpp/WebSphere/WebServerPlugIn/bin/was400plugin.so:init_exit
                                                    /usr/lpp/WebSphere,/etc/was.conf

Service    /webapp/examples/*   /usr/lpp/WebSphere/WebServerPlugIn/bin/was400plugin.so:service_exit
Service    /PolicyIVP/*         /usr/lpp/WebSphere/WebServerPlugIn/bin/was400plugin.so:service_exit

ServerTerm /usr/lpp/WebSphere/WebServerPlugIn/bin/was400plugin.so:term_exit
```

Statements added to httpd.conf to support WAS 4.0 plugin and PolicyIVP application

The `Service` statement with `/PolicyIVP/*` provides the webserver the ability to recognize any URL received with `/PolicyIVP` as its starting string and pass it over to the plugin environment.

Example: httpd.envvars configuration

Note: This will show only the *additions* to `httpd.envvars` and not the whole file.

```
 :
NLSPATH= ... /usr/lpp/WebSphere/WebServerPlugIn/msg/%L/%N
JAVA_HOME=/usr/lpp/java2/J1.3
RESOLVE_IPNAME=wsc.washington.ibm.com
RESOLVE_PORT=900
 :
```

Example: was.conf configuration

Beyond copying the supplied sample `was.conf` to the directory pointed to on the `ServerInit` statement in the `httpd.conf` file (`/etc` for this example), *no updates* to `was.conf` are necessary. In fact, if you added any references to PolicyIVP to the `was.conf` file, the plugin might think the webapp is to be run locally. Remember: the absence of a webapp "rooturi" specification in the `was.conf` allows the WAS 4.0 plugin to consider routing the request over to the web container environment.

Example: jvm.properties configuration

The `jvm.properties` file must have a pointer to the `webcontainer.conf` file to be used. For this example, that looks like this:

Pointer to webcontainer.conf from inside the jvm.properties file

This example is illustrating putting the `webcontainer.conf` file in the same directory as the `jvm.properties` file and the `current.env` file for the server instance.

Example: webcontainer.conf configuration

Two statements are updated in the `webcontainer.conf` file:

```
  :
host.default_host.alias=wsc.washington.ibm.com:8080
  :
host.default_host.contextroots=/
  :
```

As specified earlier, the HTTP Server is listening on port 8080. Therefore any URLs passed over to the web container will container not just the host name (`wsc.washington.ibm.com`) but the port as well (`8080`).

The single slash allows any application -- including the PolicyIVP application with a context root setting of `/PolicyIVP` -- to bind to the virtual host.

Starting the servers

It doesn't really matter what order you start the server region and the webserver. The two will act in concert with one another, and when the second one comes up the two will shake hands and shart exchanging information about deployed webapps.

Example: SYSOUT of webserver

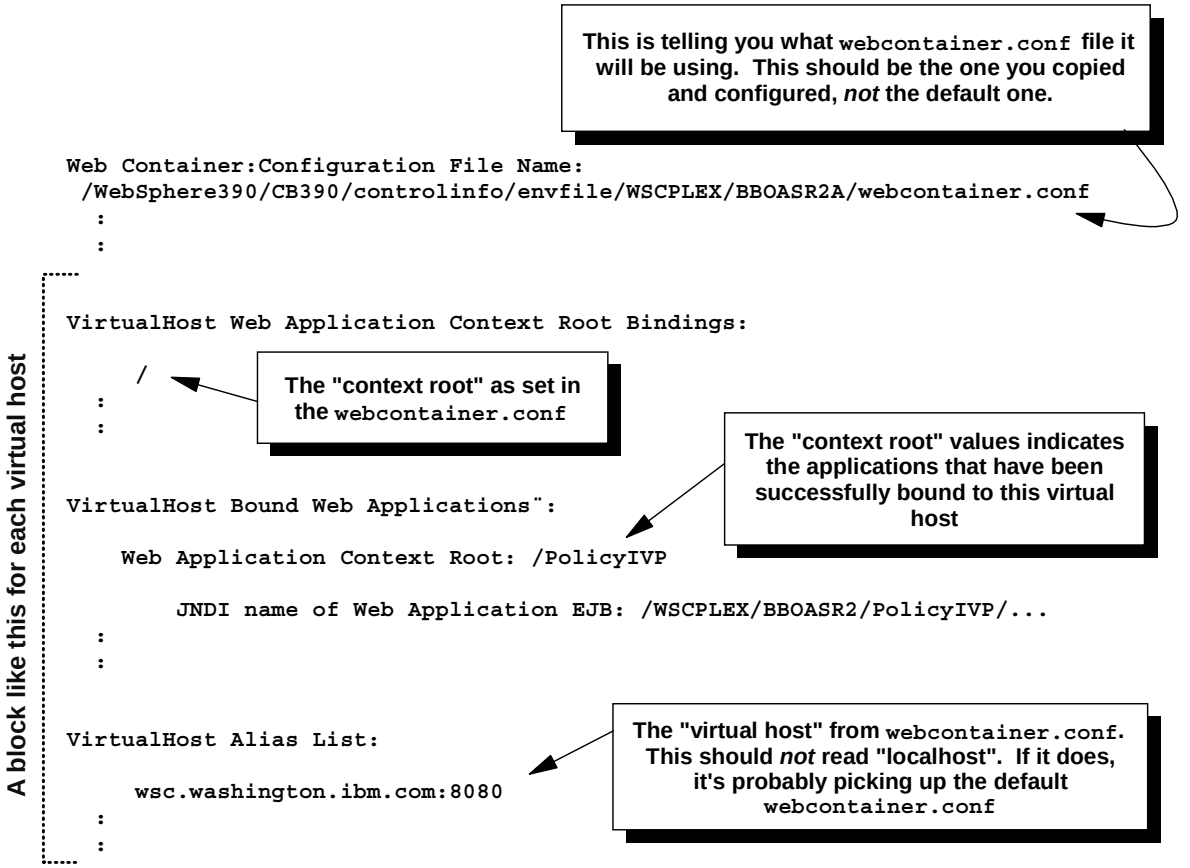
The SYSOUT of the webserver's started task will show the following information:

```
 :
WAS Startup Parameter -- Install Root = /usr/lpp/WebSphere
WAS Startup Parameter -- Configuration file = /etc/was.conf
 :
IBM WebSphere Application Server native plugin initialization went OK :-)
```

It is the "smiley face" that indicates that the plugin has successfully initilized. If you can't locate the smiley face, look for the "sad face" to indicate initialization has failed. If you can't find either, it may be that the plugin is still in the process of initializing. It may take 15 seconds or more to initialize the plugin.

Example: SYSPRINT of server region

The following picture illustrates what the SYSPRINT will look like and what things to look for as indicators of success or failure.



What to look for in the server region's SYSPRINT to validate web container configuration

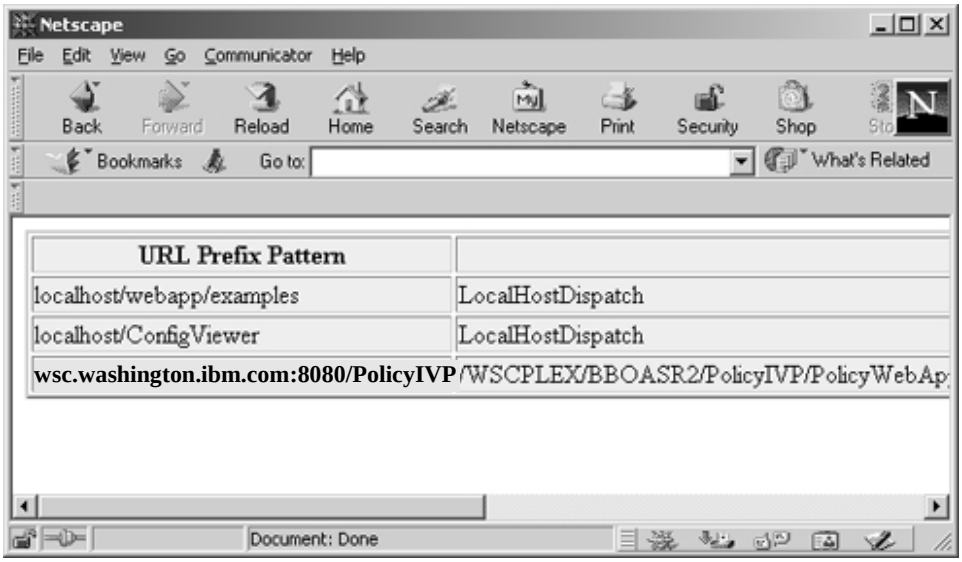
Example: Application Dispatching Information provided by plugin

The "webapp/examples" sample application provided with the plugin provides a very good way of verifying that the plugin understands the deployed webapps over in the WAS 4.0 web container environment. The URL used to reach the front page of this application is:

`http://wsc.washington.ibm.com:8080/webapp/examples/index.html`

Note: The default `was.conf` file contains all the definitions required to run the "webapp/examples" application. However, when you're setting up your `httpd.conf` file, you must provide a `Service` statement with a URL template of `/webapp/examples/*` to make this work.

On the HTML page that is returned, the "Show Server Configuration" link will invoke a servlet that runs in the plugin (not the WAS 4.0 runtime) that queries the plugin's configuration. The "Application Dispatching Information" link will bring up the following page, which indicates what web applications the plugin knows about:



Applications the plugin knows about

The picture above shows the virtual host `wsc.washington.ibm.com:8080` bound to the `/PolicyIVP` context root. The right side of the table shows where the application will be dispatched:

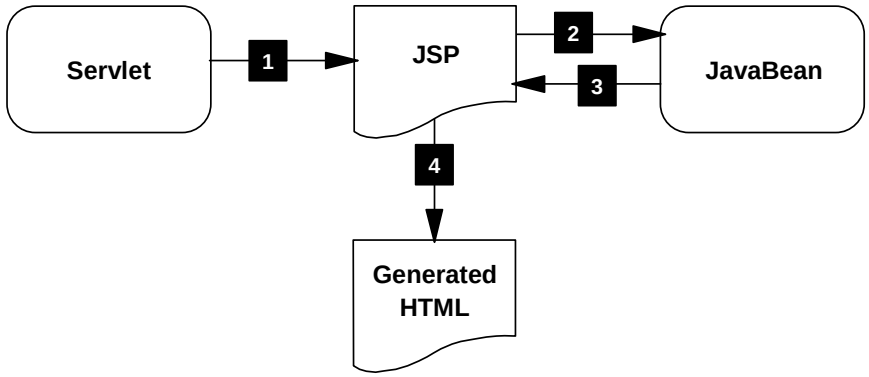
- LocalHostDispath: application run locally in the plugin
- A JNDI name (like what's shown): indicates the application will be routed over to the WAS 4.0 runtime.

The presence of the PolicyIVP information in this panel indicates the plugin and the web container have successfully communicated with one another, and that your configuration is likely correct. All that's left is to test it.

Example: SimpleJSPServlet from WAS 3.5 Standard Edition

Note: This section applies to both the plugin environment as well as the new Transport Handler environment, except for any references to configuring the web server or the plugin. But all the application stuff and the web container configuration information applies to both environments.

The WAS 3.5 Standard Edition for OS/390 product had as part of its default packaging a simple JSP application. It's structure looked like this:



Logical structure of SimpleJSPServlet and simple.jsp from WAS 3.5 SE

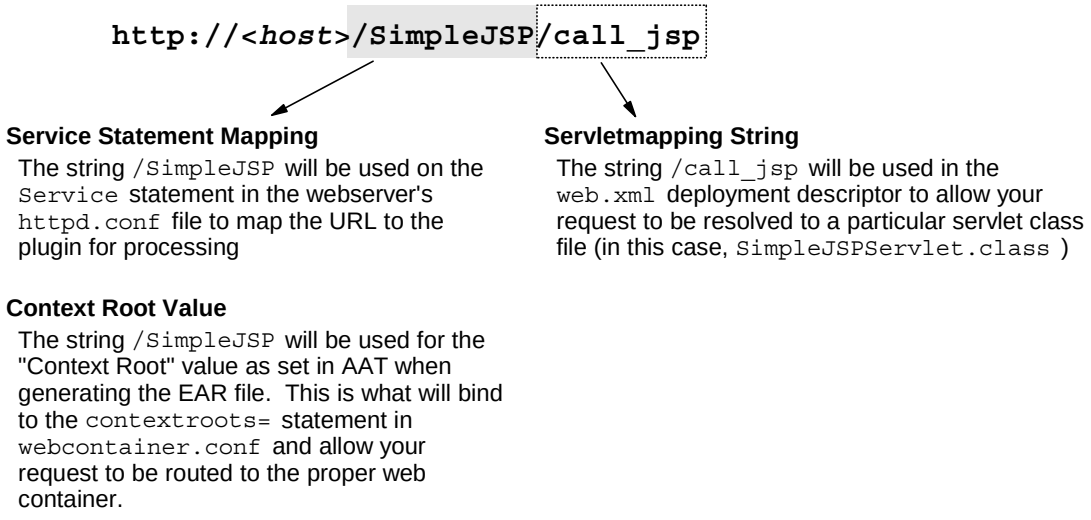
The numbered blocks refer to the following:

1. The servlet (`SimpleJSPServlet.class`) when called by you at your browser, will turn and invoke the JSP (`simple.jsp`)
2. The JSP will then turn and call the JavaBean (`SimpleJSPBean.class`). That bean does nothing more than return a string of characters back to the JSP
3. The string of characters is returned to the JSP
4. The JSP generates the HTML and ships it back to the browser

This application provided a very simple way of validating the WAS 3.5 SE environment using a servlet and JSP that had no external datasource requirement. It serves as a good example here for the same reason, and because it introduces JSPs which the PolicyIVP application doesn't have. So what this example will show is how to take the files from the WAS 3.5 SE environment and turn them into a webapp you can deploy into your WAS 4.0 environment.

Background: structure and settings for this example

Use the following chart to map the URL to the various settings so this will work:



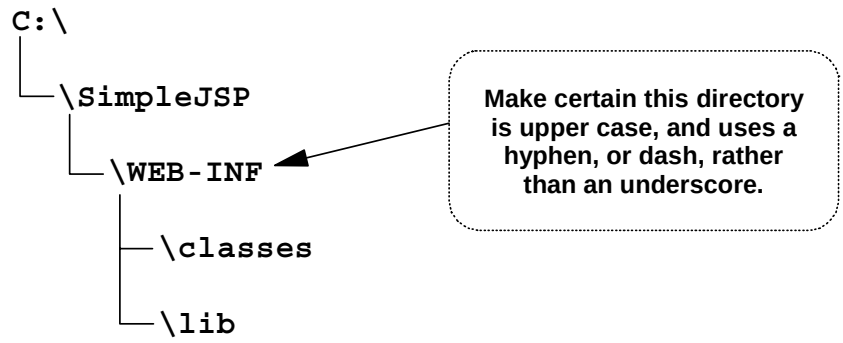
URL components and mapping to Service, Context Root and Servet Mapping settings

Background: creating a WAR file by hand

Normally you would create a WAR file using a tool like WebSphere Studio. But doing it by hand is relatively easy, so that's what you'll do here. First you have to create a directory structure on your PC that mimics the structure of the WAR. Then you'll populate the directories with the various files from the WAS 3.5 HFS on the 390 server. Finally, you'll create a WAR file using the Java "jar" command on the PC.

Activity: create WAR file directory structure on your workstation

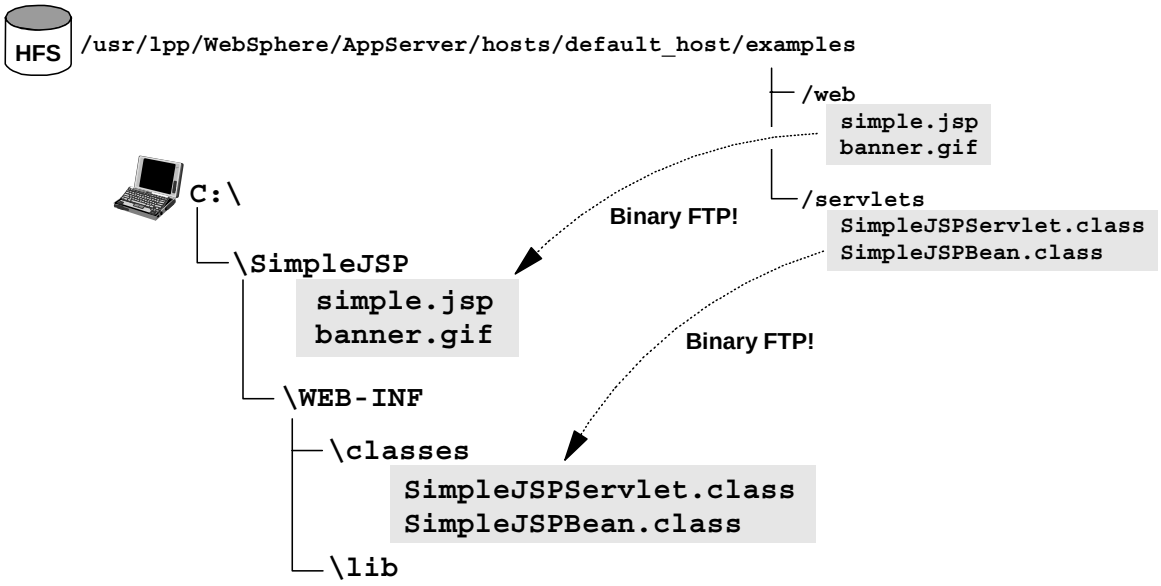
- ☐ Create the following directory structure on your PC



Directory structure on PC that mimics the standard WAR file structure

Activity: download files from WAS 3.5 SE and place in the proper directories

- ☐ Download the following files from the WAS 3.5 SE installation directory to your newly created WAR-like directory on your PC:

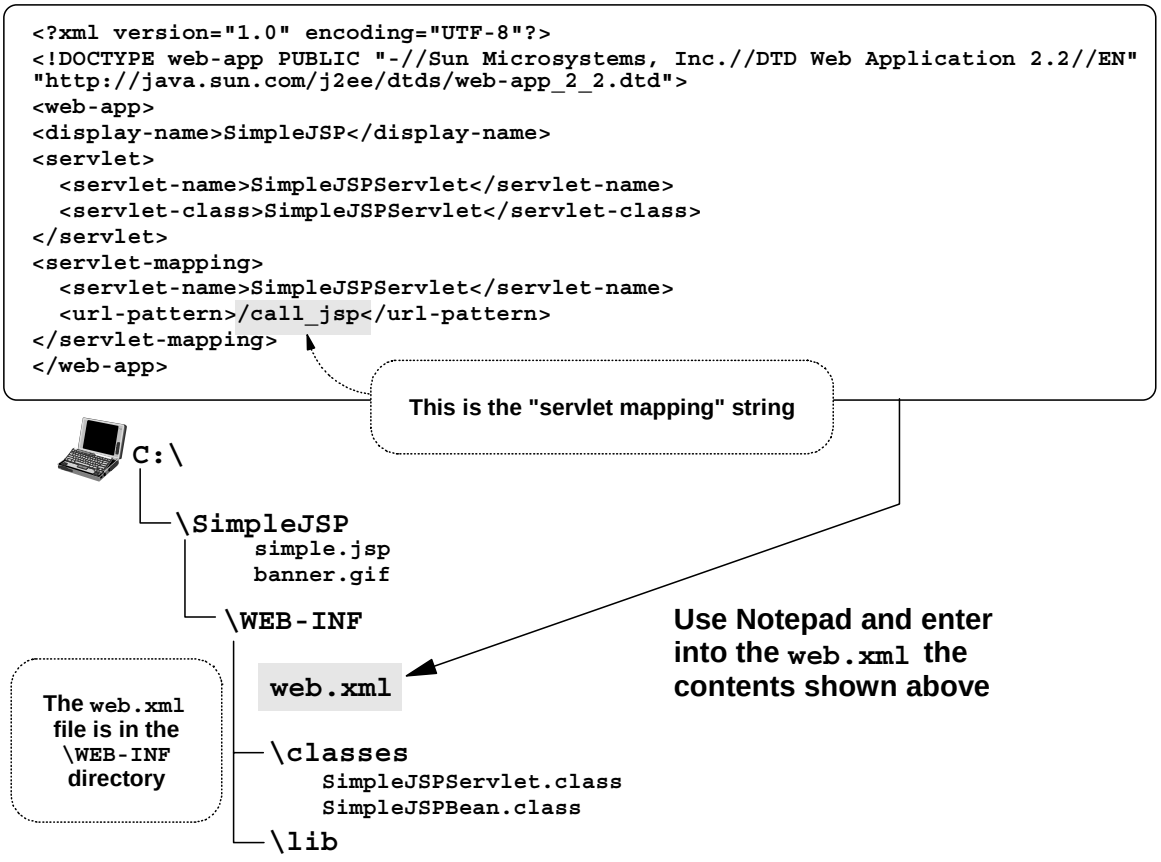


Download SimpleJSP files from host HFS to your PC

Activity: create web.xml file for WAR

The `web.xml` file is the "deployment descriptor" for the web application. Normally this file is created by the tool you use to create the WAR, but for this simple application the file is easy enough to just use Notepad and enter it by hand.

- ☐ Create a file called **`web.xml`** and place it in the directory structure as shown. Enter the data as shown into the file:



The web application's deployment descriptor file called web.xml

Activity: JAR the directory into a WAR file

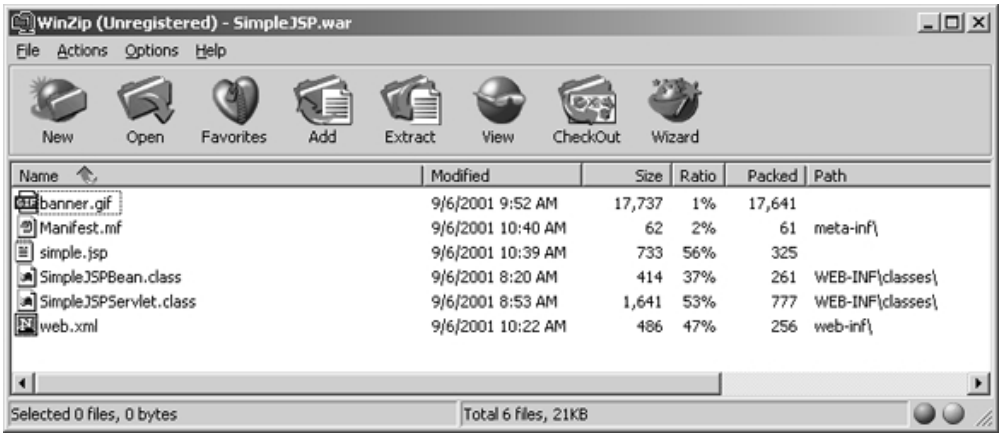
All your files are in place. You are now ready to "jar up" the directory into a WAR file.

Note: This assumes you have the Java 1.3 developer kit installed on your workstation, and the `/bin` directory of your SDK 1.3 installation is available on the `PATH` variable. The `jar.exe` command file resides in the Java `/bin` directory.

- ☐ Open up a command prompt on your workstation.
- ☐ Change directories so that you're in the `C:\SimpleJSP` directory
- ☐ Issue the following command:

```
jar -cf SimpleJSP.war *
```

- ☐ Now use WinZIP to verify the contents of the new WAR file. It should look like this:

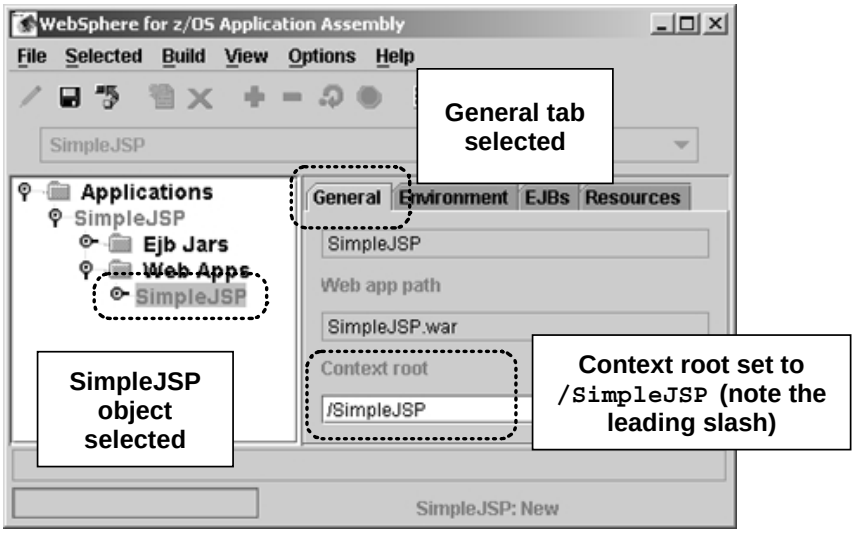


Using WinZIP to inspect the contents of your WAR file

Activity: use AAT to construct an EAR file

Now bring your new WAR file into AAT and create an EAR file. The EAR file you'll create will have no EJBs. It'll consist of one web application and that's it.

- ☐ Start AAT and add an application and call your new application `SimpleJSP`.
- ☐ Expand the tabs to expose the "EJB Jars" and "WebApps" folder. Select the "WebApps" folder, right-click and select "Import."
- ☐ Point to your new `C:\SimpleJSP\SimpleJSP.war`file and click on "OK"
- ☐ When the WAR file has been imported, select the "SimpleJSP" object, right-click and select "Modify". Then set the "Context Root" value to `/SimpleJSP` ("context root" is comparable to "rooturi" from the WAS 3.5 SE product):



Setting the context root of a webapp

- ☐ Save the modification, click on the the "SimpleJSP" *application* (not the webapp) and then right-click and select "Validate." When done, right-click and select "Deploy." Finally, right-click and select "Export" and put the file out as `C:\SimpleJSP\SimpleJSP.ear`.

Activity: provide webcontainer.conf file

This activity was discussed under "Activity: creating the webcontainer.conf file" on page 15. Go to that spot, perform the activities listed, then return to this spot.

Activity: use SME EUI to deploy into WAS 4.0 web container

With the EAR file generated, all you need do is start the SMS EUI and install this J2EE application into a server. This document assumes you have a server in which a web container has been configured.

- ☐ Create a conversation
- ☐ Select your EAR file for installation into the server
- ☐ Set the default JNDI value for the web application
- ☐ Validate, Commit and Activate the conversation.

Activity: check SYSPRINT of server region and insure application bound to virtual host

Please go to "Activity: check server region SYSPRINT" on page 28 for a discussion of checking the SYSPRINT to see if your application was bound properly. If your application server wasn't started when you deployed the web application, start the server control region and watch for it to register your new application. Then check SYSPRINT of the server region.

Activity: update httpd.conf with Service directive

Provided you have already performed "Activity: configuring the WAS 4.0 plugin code" on page 12, all you need do is add another Service statement with the URL pattern of `/SimpleJSP/*`.

- ☐ Edit your `httpd.conf` file and locate your `Service` directives.
- ☐ Duplicate one of the `Service` directives and then change the URL pattern to `/SimpleJSP/*`
- ☐ Save `httpd.conf`

Activity: start webserver and validate plugin's knowledge of new application

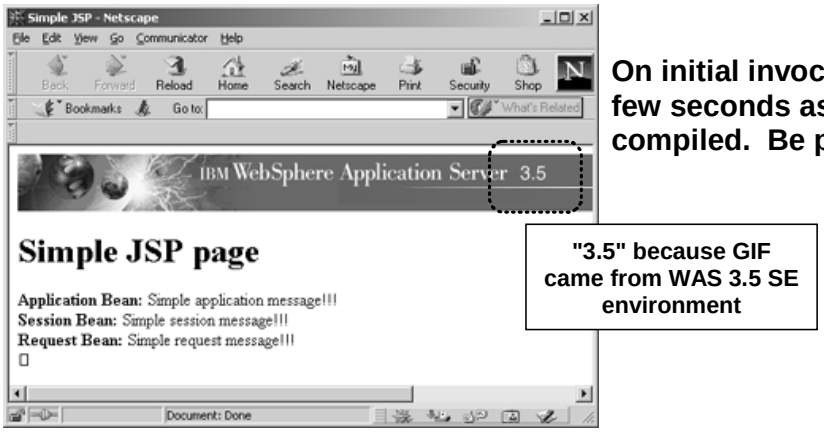
Please go to "Activity: check plugin Application Dispatching Information" on page 29 for information about this basic validation procedure.

Activity: drive SimpleJSPServlet code

- ☐ Use the following URL:

  `http://<your host>[:port]/SimpleJSP/call_jsp`

  You should get a screen that looks like this:



On initial invocation this may take a few seconds as the JSP is dynamically compiled. Be patient.

Results of SimpleJSP servlet execution

Activity: drive JSP directly, get GIF directly

You need not go through the servlet to drive the JSP, or get the GIF for that matter. Do the following:

- ☐ Issue the following URL to get the JSP directly:

  `http://<your host>[:port]/SimpleJSP/simple.jsp`

- ☐ Issue the following URL to get the GIF directly:

  `http://<your host>[:port]/SimpleJSP/banner.gif`

This illustrates how the WAS 4.0 web container can act as a JSP server for direct invocation, and it can serve as a simple file server.

Common Configuration Errors and the Symptoms Displayed

The purpose of this section is to show you some error symptoms and discuss what common configuration errors cause them. The first error symptom you'll see will be on the screen of your browser. Unfortunately, that error symptom is almost never enough to know exactly what's wrong. So further digging into the various traces is necessary.

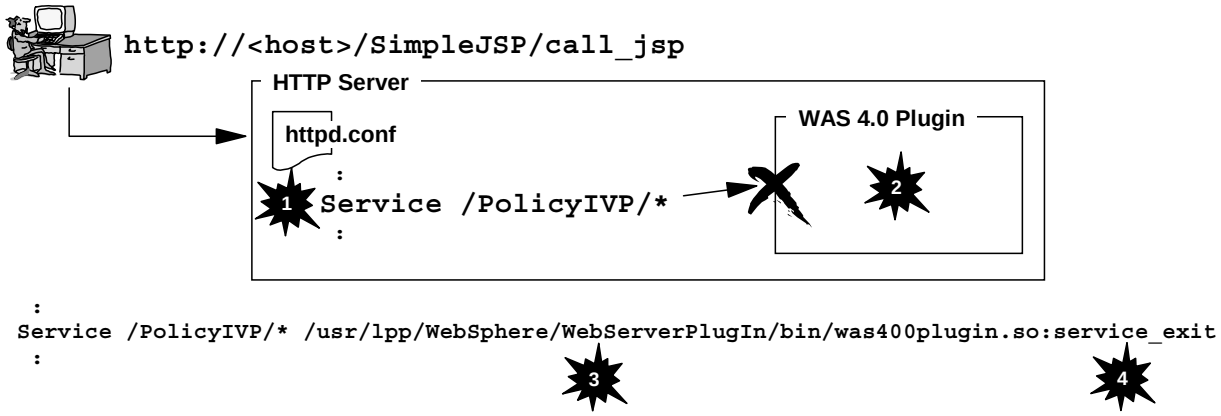
Note 1: The browser error symptoms shown in this document are based on what Netscape displays. Internet Explorer may sometimes display different things, particularly if you have the "Show Friendly Error Messsages" option turned on.

Note 2: This section was originally written to offer common configuration errors with the plugin environment, but there is some overlap with the new Transport Handler environment as well. See "Background: error conditions and the Transport Handler" on page 84 for debugging that environment.

Browser error messages

Errors related to request not reaching plugin

A request received by the HTTP Server is passed to the WAS 4.0 plugin with the Service directive in the httpd.conf file. There are quite a few reasons why that request might not make it "over the wall" into the plugin:



Some reasons why request may not reach the plugin

1. No Service directive coded that matches URL received

   For a request to make it to the plugin, a `Service` statement needs to be coded with a URL pattern that "catches" the request. It is very easy to overlook adding a new `Service` statement when adding a new webapp. And if no `Service` statement catches, the request falls through and catches on some `Pass` statement later. The plugin is never invoked.

2. Plugin not initialized

   You could have a perfectly coded `Service` statement, but if the plugin itself isn't initialized, then the `Service` statement has nowhere to send the request. The plugin may fail to initialize for several reasons. You should always check to insure the plugin has initialized before testing any new webapp.

3. Service directive has error in directory or filename of plugin code

   The `Service` directive has a rather lengthy portion where there directory and filename of the plugin code is specified. If you mistype any portion of that, the webserver will try to invoke the plugin, but will fail because no such directory or file exists. This error will *not* be caught at webserver startup; it only becomes evident when a request is mapped to that `Service`.

4. Service directive in the "exit" routine named on directive

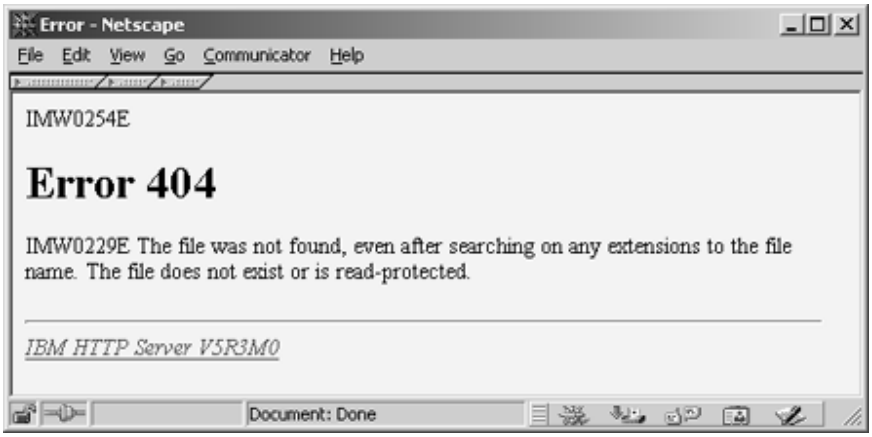
   The plugin has three different "exit routines" and the one invoked on the `Service` statement is the `service_exit`. That is, unless you mistype that value. Then problems occur.

Each of these is discussed next, with the error symptom associated with the error.

No Service directive coded that matches URL received

The request fails to map to a coded `Service` statement in the `httpd.conf` file. The webserver continues to evaluate the request against other directives, and eventually the request maps to a `Pass` statement (probably the `Pass /*` statement), or fails to map at all.

Browser error symptom



Log or trace symptom

This problem will show itself in the webserver's "vv" trace. There will be no matches on `Service` directives, and you will probably see it match against the `Pass /*` directive, but the file implied won't be found.

How to correct

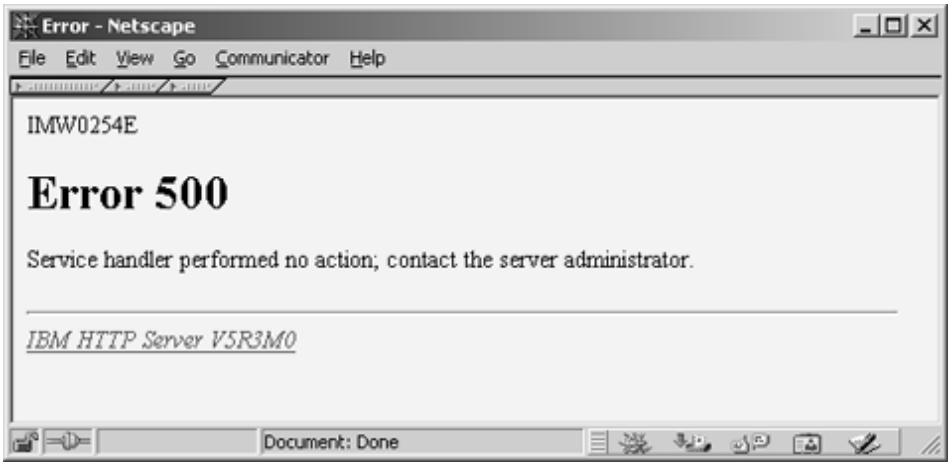
Make certain the URL being sent from the browser will map against one of your defined `Service` statements. If necessary, code another `Service` statement and restart the webserver.

Plugin not initialized

A `Service` statement matched, but the plugin to which the request is intended is not initialized. The request has nowhere to go.

A plugin not initializing can be due to errors in the `was.conf` file, an improperly coded `ServerInit` statement, or a missing `JAVA_HOME` variable in the `httpd.envvars` file. You check for initializing by searching for the "smiley face" in the SYSOUT of the webserver's started task (see "Activity: validation and basic debugging of plugin" on page 13).

Browser error symptom



The key here is the phrase "Service handler performed no action." That means the Service handler was invoked, but did nothing. So a `Service` statement was matched.

Log or trace symptom

This problem will show itself in the webserver's "vv" trace, but it is very obscure. There will be a match on a `Service` directive and then you'll see the webserver trying to match on other Service directives. But no indication of why the first match wasn't honored is given. Ultimately you see the `ERROR 500` message in the trace.

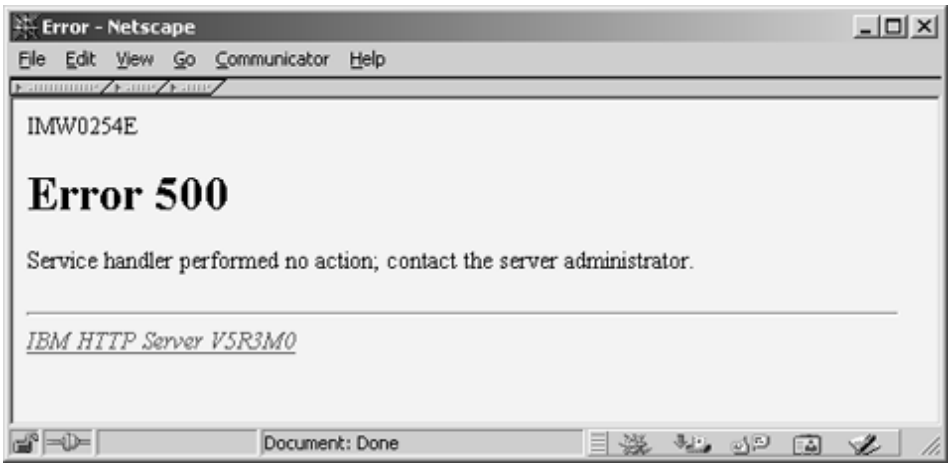
How to correct

The check the SYSOUT of the webserver's started task and look for the smiley face [`:-)`] or the sad face [`:-(`]. You will likely see the sad face or neither. Some initialization failures will be cited in the "vv" trace (for example, a `was.conf` file not found will be flagged there). Other causes for plugin initialization failures can be found in the plugin's "native" log.

Service directive has error in directory or filename of plugin code

In this scenario the plugin is initialized, and a `Service` directive is coded to match the URL. But the directory or filename of the WAS 4.0 plugin code has some error in it that causes the webserver to fail to find the plugin. If it can't find the plugin, it can't invoke the plugin with your request.

Browser error symptom



This is the same error browser symptom displayed as when the plugin isn't initialized.

Log or trace symptom

Unfortunately, the "vv" trace for this problem displays the same information as was displayed when the plugin wasn't initialized. You see a match on the `Service` statement, then without explanation the webserver starts trying to match the other `Service` statements. It finally "bails out" with a 500 error.

How to correct

First, verify that the plugin initilized. If it has, then visually inspect the `Service` statement for typos. Some common problems:

- Lowercase "s" in "Websphere" rather than correctly typed "Web**S**phere"
- Lowercase "i" in "WebServerPlugin" rather than correctly typed "WebServerPlug**I**n"

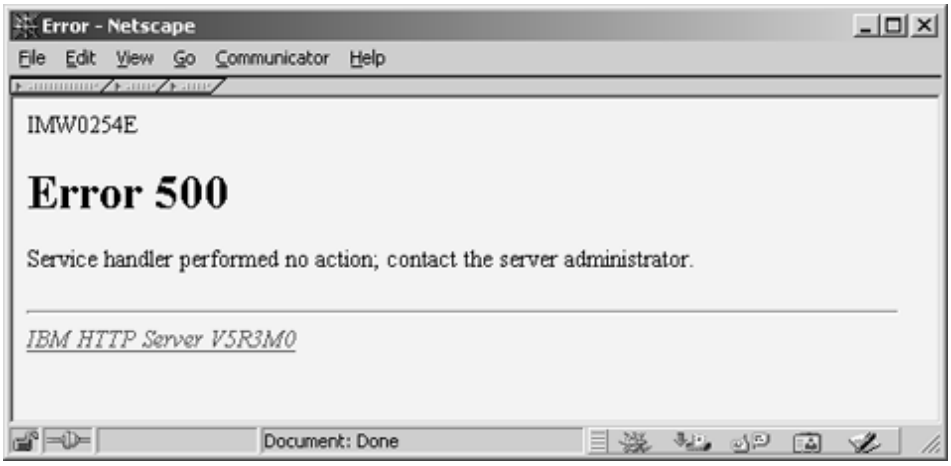
Fix and restart the webserver.

Service directive has error in the "exit" routine named on directive

The `Service` directive has an exit routine of `service_exit`, which points to the portion of the plugin code to be invoked when a request is received. If you made a mistake in coding that, the plugin will fail to initialize. (A common mistake is replicating the `ServerInit` statement to make a `Service` statement, and forgetting to change the exit routine from `init_exit` to `service_exit`).

The external symptom is similar to other things that cause the plugin to initialize, but the indication in the "vv" trace is different.

Browser error symptom



Log or trace symptom

In the "vv" trace you will find a string that flags the problem::

```
Failed to load function <exit routine>:
   EDC5214I Requested function not found in this DLL
```

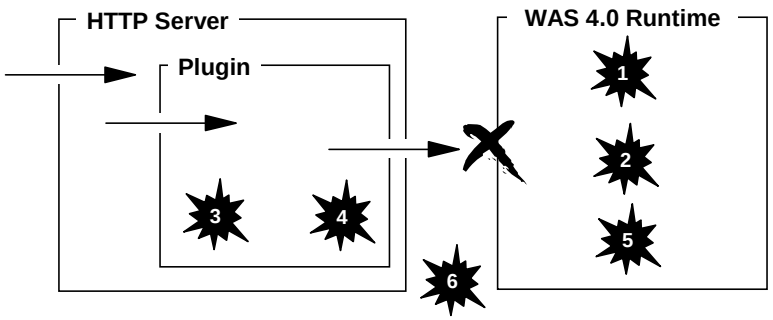
You will find no smiley face and no sad face. This will be the only indication of this problem.

How to correct

Carefully inspect the exit routines specified after the colon on the `Service` statement. Each one should have `:service_exit`. Any variations on this will cause a failure.

Errors related to plugin not passing request to web container

Once the request has been passed to the plugin, the plugin then has to get it over to the WAS 4.0 runtime. There are several things that can keep that from happening:



Some reasons why request may not be sent to runtime environment

1. WAS 4.0 application server not started

   Your webapp might be perfectly configured and deployed into the application server, but if that server isn't started then the plugin won't have much success routing the request.

2. WAS 4.0 web container not configured

   The process of configuring the web container is a manual one, and it's easy to forget to do that if you are in test mode and creating many different application servers. With no web container configured, the deployed webapp will be recognized (sort of) but it won't be bound to any virtual host. Therefore, your attempts to access it will result in failure.

3. Plugin tried to run the webapp locally

   If in the past you ran the webapp in the plugin, but are in the process of migrating your webapp to the WAS 4.0 web container, you might forget to remove the definitions from the local `was.conf` file. That means the plugin will try to run the webapp locally, probably with no success.

4. URL doesn't map to any defined context root or virtual host

   Your URL might match a `Service` statement and get passed to the plugin properly, but if the URL as received doesn't match any "context roots" found in the "string matcher table" (see "Background: how the plugin determines if a request is to be sent to WAS 4.0 runtime" on page 32 for an explanation of what that table is), then the request won't have any place to go.

5. Application doesn't bind to a virtual host

   If the context root setting for your web application isn't able to bind to a virtual host defined in the web container, your application will be unrecognized. If you're using a `contextroots=` statement of a single slash (`/`), your webapp will always bind. But if you're using a `contextroots=` value of something more specific, it might not bind.

6. Plugin isn't connected to WAS 4.0 system you think it is

   The plugin connects to the Systems Management Server (SMS) based on the `RESOLVE_IPNAME` and `RESOLVE_PORT` environment variables in the `httpd.envvars` file for the webserver in which the plugin runs. If you have multiple WAS 4.0 runtime environments, it's possible that your plugin isn't connected to the runtime you think it is. If that's the case, your webapp might not be recognized by the plugin.

WAS 4.0 application server not started

The WAS 4.0 plugin works in conjunction with the WAS 4.0 application server. If that server isn't started, then the plugin has nowhere to redirect the request.

Browser symptom

Log or trace symptom

You'll see the following error on the console:

`+BBOU0516E LOCATE REQUEST FAILED FOR SERVER -` (*server name*).

The plugin found its way to the SMS server (via the `RESOLVE_IPNAME` and `RESOLVE_PORT` variables in `httpd.envvars`) and was given the names of the application servers that have deployed webapps. Then when the plugin went to communicate with the application server, it was unable to locate the server. If the server has not been started, then the locate will of course fail.

Any requests for applications deployed in the not-yet-started application server will fail with the "Virtual Host or Web Application Not Found" message.

If you then look in the plugin's "ncf" log you'll see an indication of the error:

```
ServletHost   W Web.Group.Not.Found:."/SimpleJSP/simple.jsp"
ServletReques X Web Group Not Found
The web group /SimpleJSP/simple.jsp has not been defined
```

How to correct

Start the application server, provide enough time for the plugin to communicate with the server, and issue the URL again.

Web container not configured in WAS 4.0 application server

It is possible to deploy a web application into a WAS 4.0 application server even though the web container has not been configured. WAS will place the webapp code and files into the HFS and will even recognize that a webapp has been deployed. WAS will then make use of the default `webcontainer.conf` file, but that default copy has no virtual host defined. Failing to bind to a virtual host means the plugin will see the webapp as a "localhost" dispatch," but will fail to load it because the executable code isn't available to the plugin.

Configuring the web container is covered in "Activity: creating the webcontainer.conf file" starting on page 15.

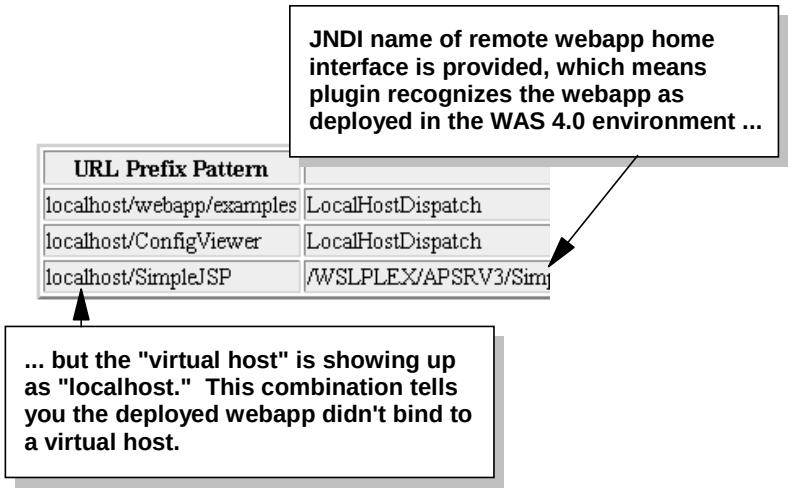
Browser symptom

Log or trace symptom

A peek in the "ncf" log will reveal the following:

```
ServletHost   W Web.Group.Not.Found:."/SimpleJSP/simple.jsp"
ServletReques X Web Group Not Found
The web group /SimpleJSP/simple.jsp has not been defined
```

Further, the "application dispatching" panel will show:



Symptom when web container not configured

The final proof of this can be found in the `SYSPRINT` of the server region for the application server:

```
Web Container:Configuration File Name:
     /usr/lpp/WebSphere/bin/webcontainer.conf
  :
VirtualHost Web Application Context Root Bindings
     /
  :
VirtualHost Alias List
    localhost
```

The key information here is the use of the *default* `webcontainer.conf` file (located at `/usr/lpp/WebSphere/bin`) and the `localhost` alias. A properly configured web

container will have your *custom* copy of `webcontainer.conf`, and the virtual host will be your IP host name for the server.

How to correct

This problem can be caused by several things:

- You simply forgot to configure the web container

  If this is the problem, then follow the instructions found at "Activity: creating the webcontainer.conf file" on page 15.

- You made a mistake in the pointer to the `webcontainer.conf`

  This pointer is found in the `jvm.properties` file, and a mistake in the typing of *any portion* of this pointer will cause the server to use the default `webcontainer.conf` file (which contains no virtual host definitions). Check the `SYSPRINT` of the server region and see if the default configuration file is in use. Then check the pointer out of `jvm.properties` and make sure everything -- case, spelling -- *everything* is correct.

Plugin tries to run the code locally

The general rule of thumb is this: if the plugin sees a match on "rooturi" in the local `was.conf` file, it'll try to run the webapp locally (in the plugin). Otherwise, the request will be passed over to the web container if the webapp is defined over there. It is good practice to follow this rule of thumb and make certain no `was.conf` definitions for your webapp exists if you intent is to run the webapp in the WAS 4.0 web container.

However, it's very easy to imagine a scenario where in the act of migrating a webapp from the plugin environment to the web container environment you accidentally forget to remove the definitions from `was.conf`. Somewhat surprisingly, this may or may not result in an error. It all depends on whether you have an explicitely coded virtual host in your local `was.conf`.

Note: The concept of virtual hosts in the WAS 4.0 web container is discussed in "Background: binding applications to virtual hosts" on page 19. The version of WAS found in the plugin also has the concept of a "virtual host" but the coding is different. All web applications defined in the local `was.conf` file must have a `deployedwebapp.<name>.host=` statement. The value found on that statement points to a `host.<name>.alias=` statement also found in the `was.conf` file. What follows the `alias=` on that statement is the virtual host. By default the value is the keyword `localhost`, but you may also have an explicitely coded IP name. If the virtual host is an explicitely coded IP name, and that IP name is *identical* to a virtual host IP name coded in the WAS 4.0 `webcontainer.conf` file, then the plugin will ignore the web container in favor of the local definition.

The best way to view this is to look at the following example of the "Application Dispatching" information from the supplied "configuration viewer":

URL Prefix Pattern	JNDI Name
`wg31.washington.ibm.com:8080/SimpleJSP`	`WSLPLEX/APSRV3C ...`
`localhost/SimpleJSP`	`LocalHostDispatch`

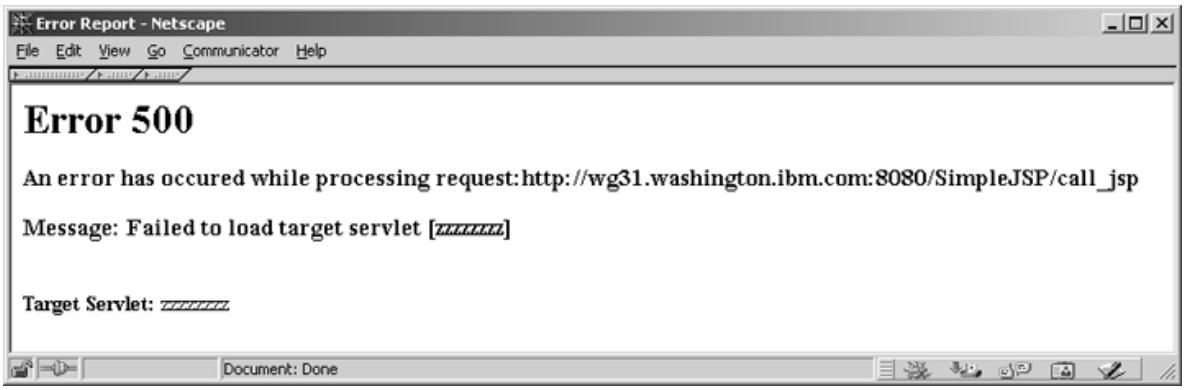
This is showing the URL pattern `/SimpleJSP` being defined in the web container *and* the plugin's local `was.conf`. The difference is this: the web container's version is bound to virtual host `wg31.washington.ibm.com:8080`, while the local plugin is using the default `localhost` virtual host.

In this example, the request will flow to the web container *only if the URL's host value matches the virtual host of* *`wg31.washington.ibm.com:8080`*. Any other host value will be run local to the plugin.

If, however, you have coded a virtual host of `wg31.washington.com:8080` in your `was.conf` *and* your `webcontainer.conf`, and the webapp `/SimpleJSP` is bound to that virtual host in *both* locations, the plugin will *only recognize the local copy*. It'll see the conflict and *reject the web container's definition*.

The following discussion will show the error when the plugin tries to run it locally and fails.

Browser symptom



Log or trace symptom

Look in the "ncf" trace of the plugin. With `appserver.loglevel=WARNING` set, you'll see the following:

```
"zzzzzzzz"
"Failed to load servlet"
javax.servlet.ServletException: Servlet [zzzzzzzz]:
      Could not find required servlet class - SimpleJSPServlet.class
```

Note: In this example the problem illustrated is the servlet class file not being found. This would be the case when a servlet is moved from the plugin environment to the WAS 4.0 web container environment. Lots of other problems could occur: servlet class file invalid, permission bits too restrictive, etc. The point is the plugin is *trying* to run the servlet, when it should be routing the request over to the WAS 4.0 environment.

How to correct

Edit the `was.conf` file and remove (or comment out) the definitions for the web application that you wish to run in the WAS 4.0 environment.

URL doesn't contain value that matches defined context root or virtual host

This problem has two forms:

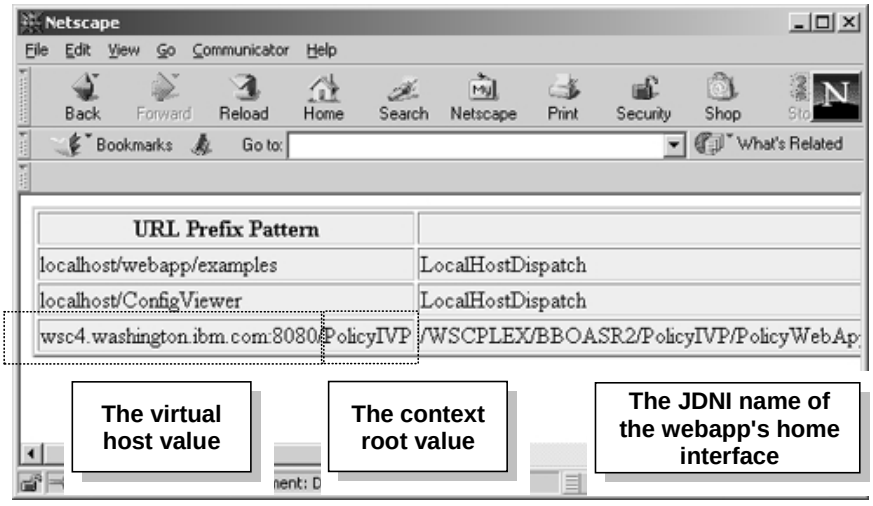
- URL doesn't match any `Service` statement and therefore doesn't get "over the wall" to the plugin. This problem will manifest itself in the way described in "No Service directive coded that matches URL received" on page 52.
- URL gets "over the wall" but the doesn't match any context root settings in the web container. This problem is very similar to that described under "WAS 4.0 application server not started" on page 56. The difference here is that the application server is up and running.

Browser symptom

Log or trace symptom

This problem is caused by the URL received not matching an entry in the "string matcher table" maintained by the plugin. The contents of the "ncf" trace will be the same as illustrated for "WAS 4.0 application server not started" on page 56.

The webapps deployed into the WAS 4.0 server will show on the "application dispatching" panel (see "Activity: check plugin Application Dispatching Information" on page 29):



"Application Dispatching Information" screen

If you see the application you're trying to invoke on this screen, that means the WAS 4.0 application server is up and the plugin has successfully communicated with the server. What that leaves is an error in your URL: either the virtual host is incorrect, or the "context root" string doesn't match.

How to correct

Provided the "application dispatching" screen shows your application and verifies that the plugin is talking to the WAS 4.0 web container, visually inspect your URL and make certain the following two things:

- The IP host name on the URL is identical (including port information, if any) to the virtual host shown on the "application dispatching" screen. Without an exact match here the plugin will *not* associate your URL with the application.

- The string that follows the first slash matches the characters on your URL exactly, including matching the case of the characters. This is the "context root" value, and WAS uses that string to match your URL request with a deployed webapp.

Anything not matching between URL and the information shown on the "application dispatching" screen will prevent the request from being honored and will result in the "Web group not defined" message.

Your application didn't bind to a virtual host

This is different from the preceeding problem. In that one your URL was incorrect. In this one the application you deployed into the web container contains a "context root" that didn't bind to any virtual host defined in the `webcontainer.conf` file. If it doesn't bind to any virtual host, then the plugin has no knowledge of the application at all.

If your `webcontainer.conf` file is making use of the single-slash "catch all" `contextroots=` setting, then this problem will not occur (that's because the single slash will allow any and all web applications to bind to the virtual host). But this problem may pop up if you are coding more explicit contextroots= values. For example, consider the setting:

```
host.default_host.contextroots=/PolicyIVP
```

and a web application `<context-root>` setting of `/SimpleJSP`. There's no match possible there. The web application will *not* bind to the virtual host.

Browser symptom

Log or trace symptom

The key indicator of this problem is the content of the "application dispatching" panel, which will fail to show your application:

URL Prefix Pattern	JNDI Name
localhost/webapp/examples	LocalHostDispatch
localhost/ConfigViewer	LocalHostDispatch

Your application doesn't appear in the "application dispatching" information at all (not even under "localhost"). Means webapp wasn't able to bind to any defined virtual host in webcontainer.conf

Application dispatching when web application doesn't bind to any virtual host

Furthermore, if you look in the SYSPRINT of the application server region, you'll see something like this:

```
VirtualHost Web Application Context Root Bindings:
    /SimpleJSP
VirtualHost Bound Web Applications: <none>
VirtualHost Alias List:
    wg31.washington.ibm.com:8080
```

In this example the test webapp used to force this condition was the *only* webapp in my web container, so the value <none> is appearing in under "Bound Web Applications." If you had other webapps that *did* bind properly, they would appear, but the webapp you're debugging would not. Look for the webapp you're debugging. If it's not showing up as bound, then this particular problem is occurring.

How to correct

Modify your webcontainer.conf file and update the contextroots= statement so your web application will bind. That statement will allow multiple string, separated by commas:

```
host.default_host.contextroots=/SimpleJSP, /PolicyIVP, /XYZ
```

Your solution may be something as simple as adding another string to the statement. Changing the webcontainer.conf file requires a restart of the application server.

Plugin not connected to the WAS 4.0 runtime you think it is

The plugin will attempt to communicate with whatever WAS 4.0 Systems Management Server (SMS) it finds based on the RESOLVE_IPNAME and RESOLVE_PORT variables in httpd.envvars. If you fail to code those environment variables, the plugin will by default go to port 900 on the TCP/IP stack on which the plugin itself is operating. If you have multiple WAS 4.0 systems running, it's possible to make a mistake and point your plugin to the wrong WAS 4.0 server. If that happens, the web application you *think* should be deployed in the web container might not be accessible by the plugin.

This problem will show itself in ways nearly identical to "WAS 4.0 application server not started" and "URL doesn't contain value that matches defined context root or virtual host" (pages 56 and 59).

Browser symptom

Log or trace symptom

The problem here is the URL won't match what's found in the "string matcher table." Therefore, the message you'll see in the "ncf" trace is:

```
ServletHost   W Web.Group.Not.Found:."/SimpleJSP/simple.jsp"
ServletReques X Web Group Not Found
The web group /SimpleJSP/simple.jsp has not been defined
```

You'll not see anything in the logs that indicates the plugin is pointed to the wrong IP name and port. You must simply review the configuration of `httpd.envvars` and make sure you have it coded to the proper values. Checking the "application dispatching" information helps isolate this problem.

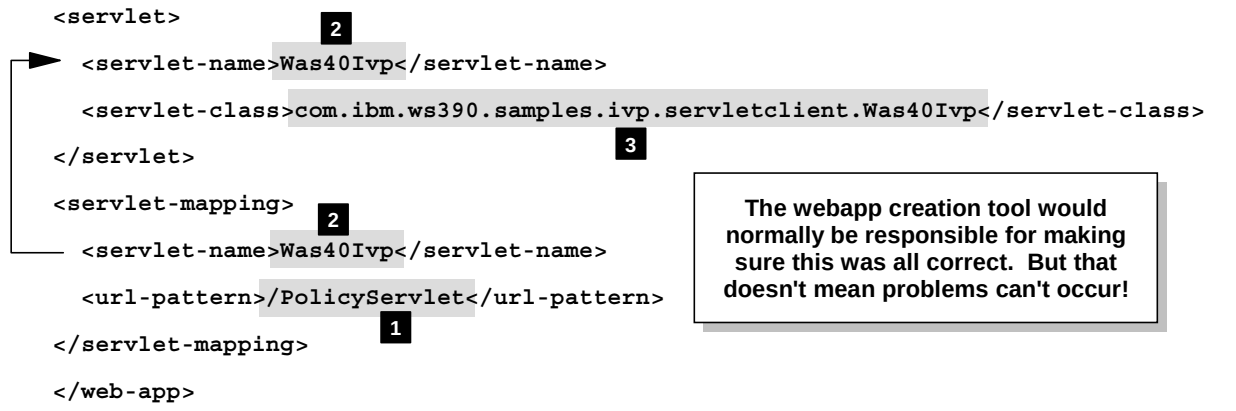
How to correct

Visually inspect the `RESOLVE_IPNAME` and `RESOLVE_PORT` values in `httpd.envvars` and correct if necessary.

Errors related to request not resolving to web application class file

The URL request may map to a `Service` request and be thrown "over the wall" into the plugin; it may map to a virtual host and context root in the "string matcher table" and be routed over to the web container, and then *still fail*.

The ability of a URL request to make its way over to the web container is based on your coding of the `webcontainer.conf` file and the value of the `<context-root>` XML tag found in `application.xml` of the deployed EAR file. But there's more to the webapp puzzle than that. There is the `web.xml` file inside the webapp's WAR file, and that's where all manner of problems can be introduced:



Where problems can be introduced into the web applications deployment descriptor

1. The `<url-pattern>` tag contains the "servlet mapping" string. If the value on the URL doesn't match any `<url-pattern>` string defined in any webapp deployed in the container, then the request will fail.
2. The `<servlet-name>` string is what ties together the `<servlet-mapping>` stanza of the XML file with the `<servlet>` stanza. If the values don't match one-for-one, then the request will fail. (Surprisingly, neither the AAT tool nor the SMS GUI checks for this error).
3. The `<servlet-class>` tag points to the actual class file that is to be invoked. It's quite possible the class file named is incorrect, or the class file itself is corrupt or otherwise invalid.

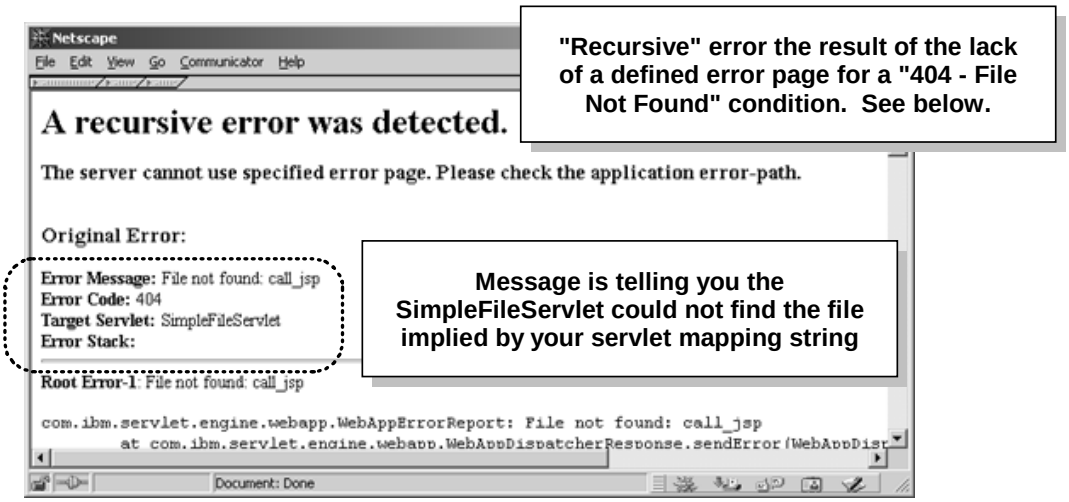
Servlet mapping string doesn't match

This will be a quite common problem. Between the time you create the web application and set its `<url-pattern>` and the time you issue your first URL against that servlet, you'll forget the format of the servlet mapping string. You'll take a guess, and the guess will be

wrong. The URL will get passed to the plugin based on a match to a `Service` statement, and it'll get routed to the web container based on the `<context-root>` match. But without a `<url-pattern>` match, WAS won't know what specific servlet to invoke.

What happens next is described in "Background: WAS 4.0 serving of static files and JSPs" on page 25. The request will eventually filter down to being considered a request for a static file, and the name on the URL won't be the name of a file WAS sees in the HFS. So it'll issue the following error:

Browser symptom



Default error page when servletmapping not found and WAS looks for static file without success

This error is somewhat ugly in that the page is really telling you two things: the static file wasn't found (resulting in a 404 error), *and* the error page for the 404 condition wasn't found either. Error pages for web applications are defined in the `web.xml` file, and up to this point the sample `web.xml` files in this document have not included that XML coding. Here's what that XML stanza would look like to specify a custom "404" error page (this example will use the `web.xml` file as provided in "Activity: create web.xml file for WAR" on page 45):

```
<webapp>
 :
<servlet>
  <servlet-name>SimpleJSPServlet</servlet-name>
  <servlet-class>SimpleJSPServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>SimpleJSPServlet</servlet-name>
  <url-pattern>/call_jsp</url-pattern>
</servlet-mapping>
<error-page>
 <error-code>404</error-code>
 <location>/404.html</location>
</error-page>
</web-app>
```

The server will look for this page in the root of the HFS directory structure *that represents the deployed WAR file*. If the `<location>` string was `/subdir/404.html`, it would look in the subdirectory `/subdir`. If you don't have an error page defined, or WAS can't find the error page you specify, you get the "recursive error" symptom.

Log or trace symptom

The symptom of this problem will occur in the WAS 4.0 application server region's SYSPRINT. The HTTP Server's "vv" trace will show the request being passed to the plugin based on a match to a Service statement. The plugin's "ncf" trace will show normal operation because the URL will match to a virtual host and context root string. The browser symptom shows enough detail of the problem so that looking in the SYSPRINT is not required.

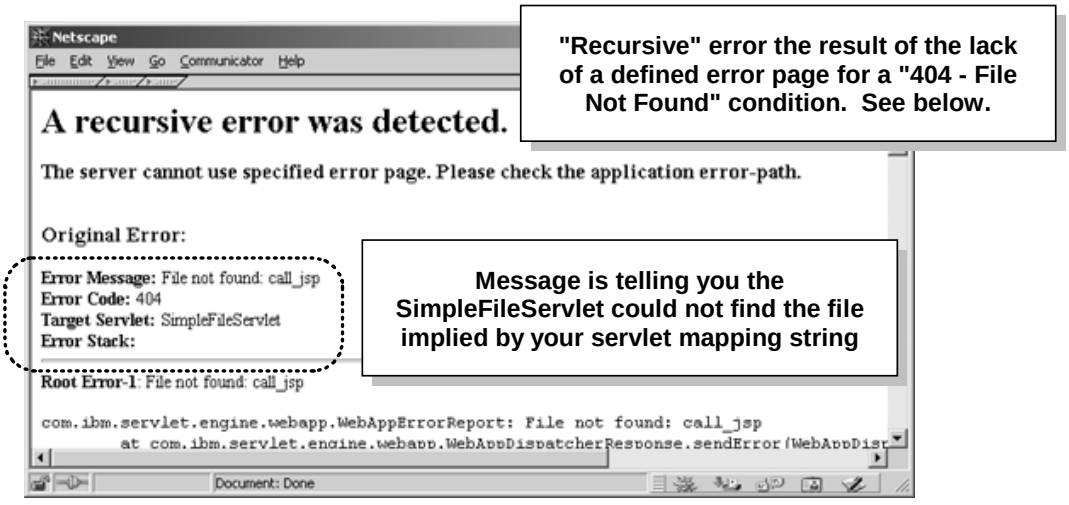
How to correct

Correct the format of your URL, or change the <url-pattern> value in the web.xml file, refresh your WAR file, regenerate your EAR file and redeploy the application.

Mismatch in servlet name in deployment descriptor

This problem is fairly obscure, and would occur only if you're hand-building the web.xml file in the WAR file. A webapp construction tool would likely not create this problem. Nevertheless, this symptom would occur any time the <url-pattern> value is defined and found by WAS, but no associated servlet can be found in the web.xml file.

Browser symptom



Default error page when servletmapping found, but no associated servlet defined

It turns out WAS will treat this problem just like when it can't get a hit on a "servletmapping" string: it falls back and assumes the request is for a static file. It'll then go looking for the file, and if it fails it'll throw the default "recursive error" page.

Log or trace symptom

The symptom of this problem will occur in the WAS 4.0 application server region's SYSPRINT. The HTTP Server's "vv" trace will show the request being passed to the plugin based on a match to a Service statement. The plugin's "ncf" trace will show normal operation because the URL will match to a virtual host and context root string. The browser symptom shows enough detail of the problem so that looking in the SYSPRINT is not required.

How to correct

Reconstruct the WAR file with a corrected web.xml file. Make certain the <servlet-name> string is present in both the <servlet-mapping> stanza as well as the <servlet> stanza.

Class file incorrect

In this case the servlet mapping string is correct, but the class file referenced in the `<servlet-class>` tag of the `web.xml` file isn't correct, and therefore the class file can't be found. The WAS web container will try to locate the class file, but will fail.

Browser symptom

Unfortunately, the symptom for this problem is very cryptic. What you will see on the browser screen is the following:

Log or trace symptom

This problem lies entirely within the WAS 4.0 runtime, so the webserver's "vv" trace and the plugin's "ncf" trace are of no use. The output provided to the server region's `SYSPRINT` is by default minimal. If your trace settings in `current.env` are the default, you will not see any evidence of this problem in the `SYSPRINT`.

However, if in your `current.env` file you have the following coded:

```
   :
TRACEALL=1
TRACEBUFFLOC=SYSPRINT
TRACEPARM=00
   :
```

then you'll get some information out to your `SYSPRINT` that'll indicate the problem. Here's what you'll see:

```
   :
"Failed to load servlet": javax.servlet.ServletException:
    Servlet [SimpleJSPServlet]:
   :
Could not find required servlet class - SimpleJSPServletx.class
   :
Unexpected internal engine error while sending error to client:
    "/SimpleJSP/call_jsp"
   :
```

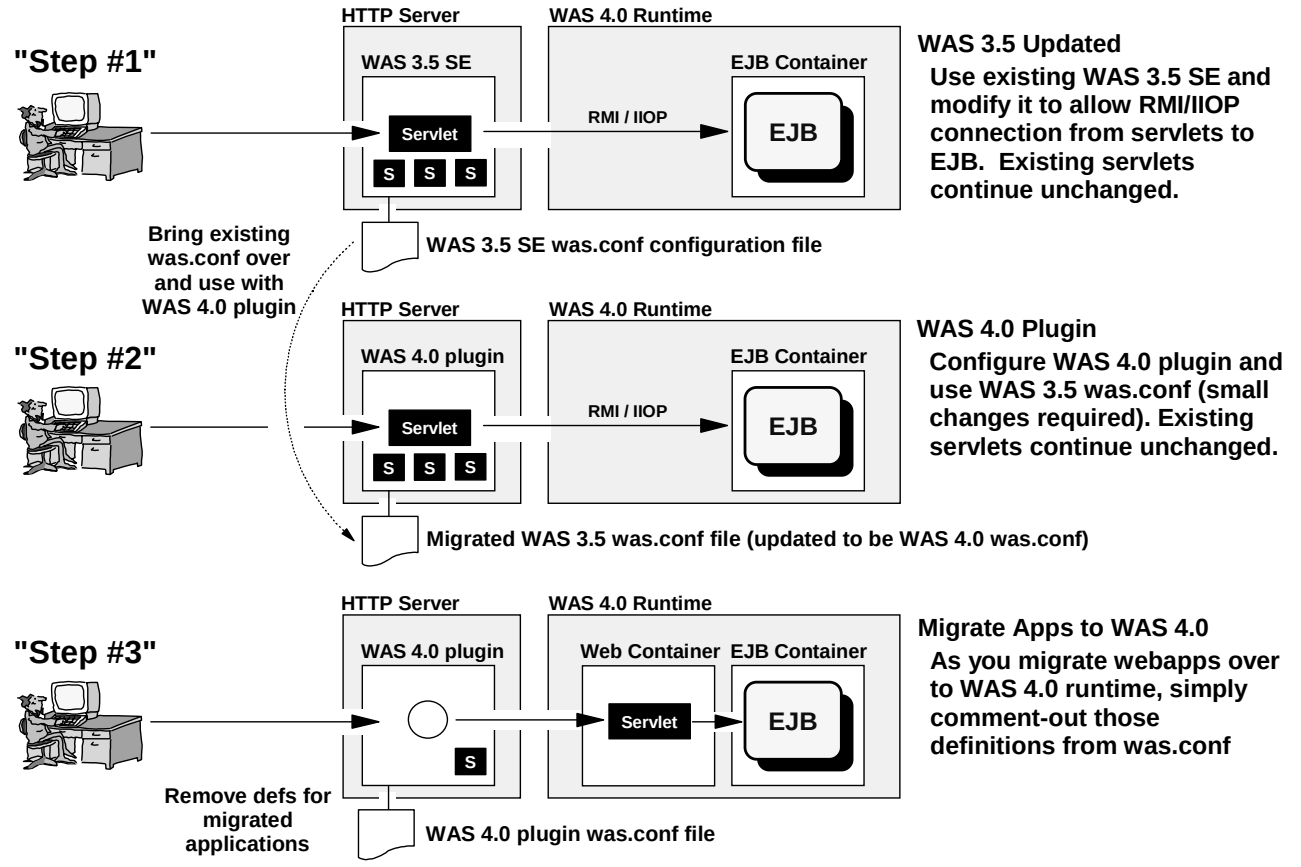
Why the resulting message to your browser "contained no data" is still a mystery.

How to correct

Very carefully inspect your `web.xml` file and make sure the pointer to the class file in the `<servlet-class>` tag is correct. Watch for misspellings of the file name, or possibly an error in any of the qualifiers of a longer package name. If you spot an error, correct it, re-assembly the WAR/EAR file and redeploy the application.

Migration Scenarios

This section covers what to do when migrating your web application environment from the WAS 3.5 SE environment to the new WAS 4.0 environment. The news here is good, because the new WAS 4.0 plugin is capable of running servlets locally or routing the request over to the WAS 4.0 runtime. The migration path looks like this:



Migration path from WAS 3.5 SE environment to WAS 4.0 runtime

Background: overview of the three steps of migration

The migration path is a fairly straight-forward thing:

Step 1: update WAS 3.5 SE to communicate with WAS 4.0 runtime

Assuming you have a WAS 3.5 SE environment presently operating on your system, you probably have servlets configured and operating in the WAS 3.5 plugin. The objective of this step is to update your WAS 3.5 SE environment so servlets designed to communicate with EJBs can do so.

Step 2: configure WAS 4.0 plugin and use existing was.conf configuration file

In this step you change your plugin environment from WAS 3.5 SE to the new WAS 4.0 plugin. Because the new plugin is capable of running servlets, your existing servlet base can easily be moved to the new WAS 4.0 plugin by simply using your existing `was.conf` configuration file with the new WAS 4.0 plugin.

Step 3: migrate web applications over to WAS 4.0 web container environment

Once you have the WAS 4.0 plugin configured, you may migrate your webapps over to the WAS 4.0 web container environment at your leisure. This involves packaging the webapps

into WAR files and deploying them into the web container, and then removing from `was.conf` the `deployedwebapp` and `webapp` definitions for that web application.

Activity: configuring the WAS 3.5 plugin code to allow communication with EJB

This section assumes you already have a WAS 3.5 SE environment working on your system, and that your objective is to update that environment so that servlets written to communicate with EJBs can do so. This section does not provide the full instructions on how to configure the WAS 3.5 SE plugin.

The WAS 3.5 plugin configuration is very similar to that of the WAS 4.0 plugin. But some of the directories are different, so don't assume the WAS 4.0 plugin directions apply to the WAS 3.5 plugin environment.

Very Important Note: Any given webserver may have *either* the WAS 4.0 plugin configured, *or* the WAS 3.5 plugin configured, *but not both at the same time*! See "Question: can both plugins be configured in the same webserver?" on page 5.

Do the following:

Note: This document assumes you'll do the appropriate backing up of any files that you're changing.

- ☐ Edit the `was.conf` file and add the following to the `appserver.classpath` property:

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

  Why? This provides the client (the servlet) access to the necessary client-side Java components to access the WAS 4.0 EJB runtime. Without this, the client would try, but fail, to make the connection.

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

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

  Why? The client (the servlet) needs to be know where the "initial context factory" code resides so that it can look up the home interface of the target EJB and create the object. Without this, the client would never be able to locate the bean's home interface.

- ☐ Edit the `httpd.envvars` file and add the following two variables:

  `RESOLVE_IPNAME=`*<fully qualified IP host name of server on which WAS 4.0 SMS exists>*
  `RESOLVE_PORT=900` *(or port on which WAS 4.0 SMS server is listening if not default)*

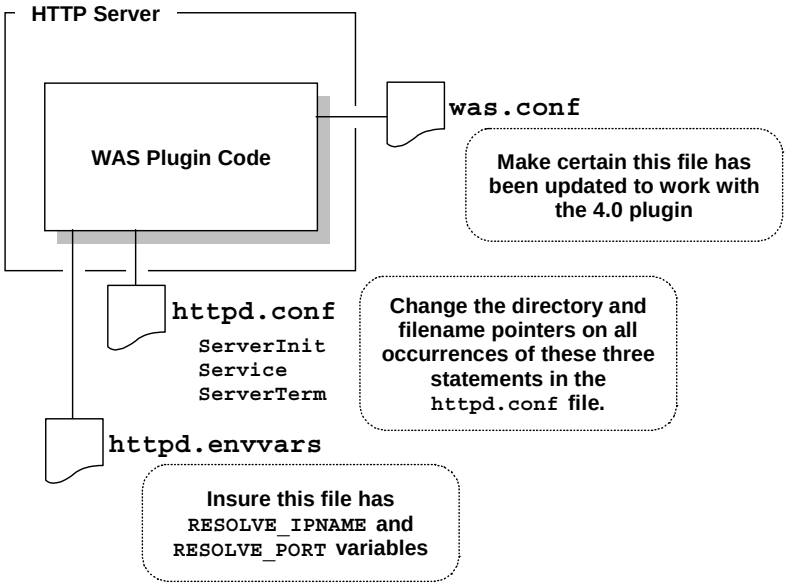
  Why? The client (the servlet) needs to know the location of the Systems Management Server so that it can connect to the WAS 4.0 runtime and request the services of the runtime. Absent this update, the client wouldn't have a clue where to go to satisfy its desire to run the EJB.

With these changes made, stop and restart the webserver to pick up the changes. Your WAS 3.5 plugin environment is now ready to accept servlets written to access EJBs.

"Step #1" from the picture at the beginning of this section has been satisfied.

Activity: changing plugin from WAS 3.5 to WAS 4.0 plugin

Assuming that you have a WAS 3.5 SE plugin running in your webserver, you are now ready to change the environment to use the new WAS 4.0 plugin. This involves the following things:



Modifications needed to WAS 3.5 plugin environment to make it run the WAS 4.0 plugin

Question: can WAS 3.5 SE was.conf file be used with WAS 4.0 plugin?

Yes. The format of the `was.conf` is largely identical between the two. There are a few parameter changes you need to make when migrating a WAS 3.5 SE copy of the `was.conf` for use with the WAS 4.0 plugin. This is good news for those who have invested considerable time configuring webapps for the WAS 3.5 plugin and now wish to migrate to the WAS 4.0 plugin environment. What is required to use the WAS 3.5 `was.conf` with the WAS 4.0 plugin is explained next.

Activity: preparing a WAS 3.5 was.conf for use with WAS 4.0 plugin

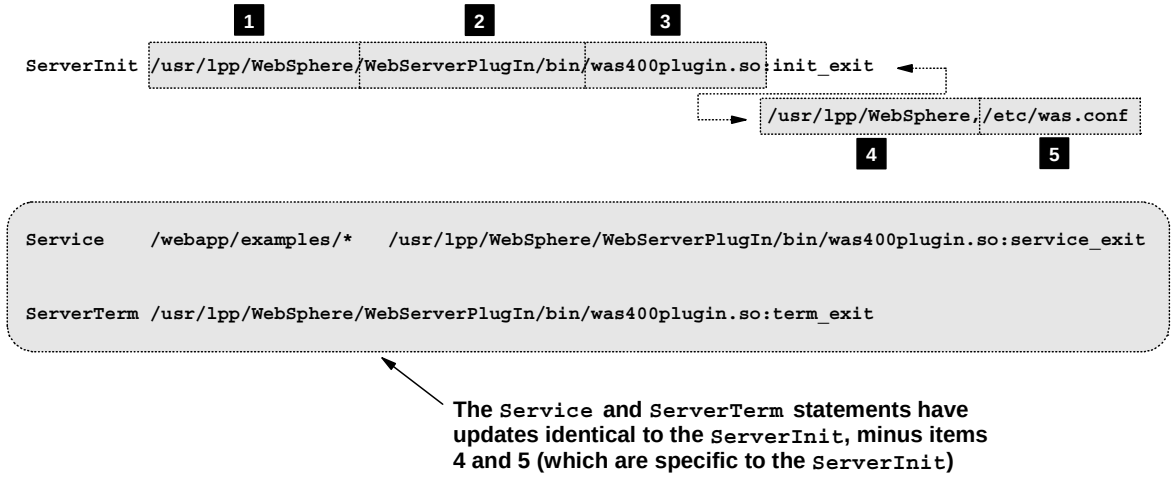
If you wish to use an existing copy of a WAS 3.5 SE `was.conf` with your new WAS 4.0 plugin, do the following:

Note: This document assumes you'll do the appropriate backing up of any files that you're changing.

- ☐ Edit the copy of `was.conf` you wish to use with the WAS 4.0 plugin. Locate the `appserver.version` property and change its value from `3.50` to `4.00`.
- ☐ Remove all `deployedwebapp` and `webapp` statements for applications you intend to run in the web container environment of the WAS 4.0 runtime. (For the WAS 4.0 plugin to correctly route requests over to the web container for execution, it must see that no local webapp definitions exist in the `was.conf`.)
- ☐ If you wish (but it is not critical), you may remove the `ws390crt.jar` file from the `appserver.classpath` statement and remove the `appserver.java.extraparm` statement from the was.conf file. These are the updates outlined in "Activity: configuring the WAS 3.5 plugin code to allow communication with EJB" on page 68.

Activity: changing the plugin pointers in the httpd.conf file

Assuming you have a working WAS 3.5 plugin environment, the steps necessary to point to the new WAS 4.0 plugin involve changing three things in the `httpd.conf` file:



Changes necessary to point to new WAS 4.0 plugin

Each numbered block in the picture is described next (the "to do" activities follow):

1. This is the directory in which the WAS product is installed. By default this is `/usr/lpp/WebSphere` *for both WAS 3.5 and WAS 4.0*. Clearly if you have both versions installed on the same system, both can't be installed at the same mount point. Therefore you have to be careful when you code the WAS 4.0 plugin's updates because you'll need to point to where WAS 4.0 is installed.
2. This is the directory under the install root (numbered block #1) which contains the plugin code. For WAS 3.5, this value was `/AppServer/bin`. For WAS 4.0 it is `/WebServerPlugIn/bin`.
3. This is the file name of the plugin code. For WAS 3.5 it was `was350plugin.so`, for WAS 4.0 it is `was400plugin.so`.
4. This is the first parameter on the `ServerInit` statement, and is separated from the rest of the statement by a blank space (and all coded on one line). This points to the install root of the WAS code. This value should be identical to the value you coded for numbered block #1. Make sure you're pointing to the right directory for WAS 4.0 and not back to the directory where WAS 3.5 was installed.
5. This is the second parameter on the `ServerInit`. It points to the directory and file name for the plugin configuration file. It is separated from the first parameter by a comma. This should point to the `was.conf` you updated as described in "Activity: preparing a WAS 3.5 was.conf for use with WAS 4.0 plugin" on page 69.

Do the following:

Note: This document assumes you'll do the appropriate backing up of any files that you're changing.

- ☐ Edit the `httpd.conf` for your webserver and locate the `ServerInit` statement. There should be only one, and it if you're presently running WAS 3.5 SE it'll point to WAS 3.5.
- ☐ Inspect the install root directory specified on the `ServerInit` (numbered block #1 in the picture) and make change it to point to where *WAS 4.0 is installed on your system.*
- ☐ Inspect the directory and filename of the plugin code itself on the `ServerInit` statement (numbered blocks #2 and #3 in the picture). Change this to:

  `/WebServerPlug`**`I`**`n/bin/was400plugin.so:init_exit`

  Note the upper-case "I" in "PlugIn" of the directory `/WebServerPlugIn`. That's something easy to overlook, and if overlooked it will cause the plugin to not be found.

- ☐ Inspect the first parameter on the `ServerInit` (numbered block #4) and change it to the install root of *WAS 4.0*. This should match the value you coded for numbered block #1.
- ☐ Inspect the second parameter on the `ServerInit` and make certain it points to the directory and file of the `was.conf` you wish to use.
- ☐ Now go through *every* `Service` statement in your `httpd.conf` and make certain that the directory and plugin module points to the install root (block #1), the plugin directory (block #2) and plugin module name (block #3).
- ☐ Finally, locate the `ServerTerm` statement and change the directory and plugin module name to equal that of what you provided the `ServerInit` and `Service` statements.

Note: It is important to note that function name on the module (what follows the colon after the module name of `was400plugin.so`) is different for the `ServerInit` vs. `Service` vs. `ServerTerm`. The function name for `ServerInit` is `:init_exit`, for `Service` `:service_exit` and `ServerTerm` `:term_exit`.

Activity: making certain the httpd.envvars file is correctly configured

The steps here are the following:

- ☐ Edit your `httpd.envvars` file and make certain the JAVA_HOME variable is present and set to the following:

  `JAVA_HOME=/usr/lpp/java2/J1.3`

  or wherever the Java 1.3 Developer Kit for Java is installed on your system. You'll probably already have this if you had WAS 3.5 SE running. Both WAS 3.5 SE and the WAS 4.0 plugin require the IBM Developer Kit for Java Platform 1.3.
- ☐ Stay in your `httpd.envvars` and add the following to the `NLSPATH` variable:

  `/usr/lpp/WebSphere/WebServerPlugIn/msg/%L/%N`

  or whatever your WAS 4.0 install root happens to be.
- ☐ Add the following two variables to `httpd.envvars`:

  `RESOLVE_IPNAME=`*<fully qualified IP host name of WAS 4.0 SMS server system>*
  `RESOLVE_PORT=900` *(or port on which WAS 4.0 SMS server is listening if not default)*

Note: If your HTTP Server (and therefore the plugin as well) is on the same system as your WAS 4.0 runtime, and you configured the SMS server to use the default port value of 900, you don't need these two values. But coding them is relatively easy, and it avoids confusion. So go ahead and code these even though strictly speaking they're not always necessary.

Activity: restart webserver and validate plugin initialization

At this point you're ready to see if all your changes were correctly entered, at least as far as allowing the plugin to initialize properly. Go to "Activity: validation and basic debugging of plugin" on page 13 for instruction on validating the initialization of the plugin.

Activity: migrating web applications from plugin to WAS 4.0 runtime

There are two steps involved with this:

1. Packaging your webapp into an WAR file format and deploying that application into the WAS 4.0 runtime environment.
2. Removing from the `was.conf` file any application definitions for the application.

This document is not intended to cover the packaging and deployment activities. However, the information provided in "Webapps Running in WAS 4.0 Runtime and Driving EJB" on page 15 and "Example: PolicyWebApp in the PolicyIVP Application" on page 37 covers some of the background on this process.

The reason why the application definitions (`deployedwebapp` and `webapp`) in `was.conf` are removed is because the WAS 4.0 plugin will seek to run the application locally (within the plugin rather than over in the web container) if it gets a "hit" on a `deployedwebapp` "rooturi" definition in the `was.conf`.

Note: In reality, the match is done on the [virtual host | rooturi] pair. If the URL received matches both the rooturi value and the defined virtual host, the plugin will try to run the request locally. On the other hand, if it sees a match for virtual host and context-root (essentially the same thing as rooturi) over in the web container, it'll route the request over there. Using the keyword `localhost` as a virtual host in the `was.conf` will not take precedence if an exact match on virtual host and context-root is found in the web container. See "Background: when same virtual host is defined in both environments" on page 92 for more on this subject of having the same virtual host values coded in both environments.

Therefore, to make the request flow over to the WAS 4.0 runtime after the application has been deployed there, you must remove the application definitions from `was.conf`. If you want to deploy the application in both environments, but run the application in the plugin for a while, make sure the virtual host and rooturi values coded in `was.conf` match the received URL. The request will then be run in the plugin. Then, when you're ready, remove the rooturi definition from `was.conf` and the request will then flow to the web container.

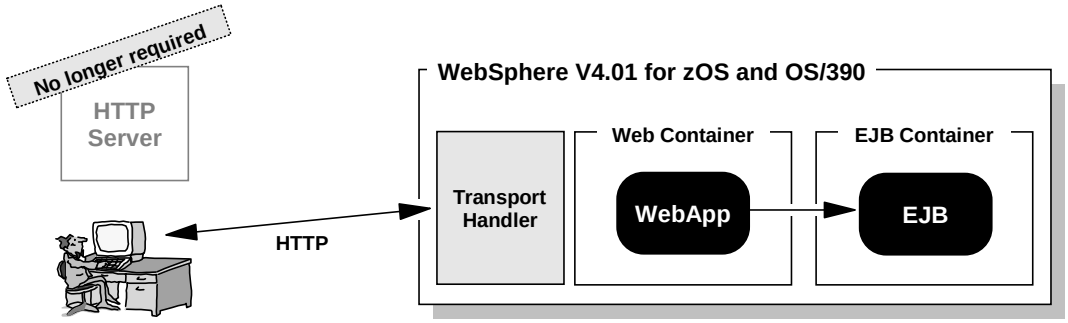
Question: how does the Transport Handler figure into this migration?

The new Transport Handler function provides an integrated HTTP listener, and greatly reduces the complexity of the configuration. However, it does *not* provide a servlet execution environment. Therefore, it can't be brought into the mix until at least *some* of the webapps have been moved to the WAS 4 web container environment. Once some webapps are deployed into the web container, the new Transport Handler can be employed to be the HTTP listener for those migrated webapps.

It is possible to have both the plugin and the Transport Handler as part of the configuration (see "Question: how do I configure both the plugin and the Transport Handler for a given server?" on page 77). Therefore, the migration picture will likely be similar to what was shown at the beginning of this section, with the Transport Handler introduced sometime after "Step 2" and run concurrently with the plugin for some period of time.

The WAS 4.0.1 Transport Handler

When V4.01 of WAS for OS/390 was delivered in October of 2001, one of the new features is an integrated "Transport Handler". That's a fancy way of saying "integrated HTTP listener." This means that you no longer must have an HTTP Server with the WAS 4.0.1 plugin configured. Your environment can now be as simple as this:



The V4.01 integrated HTTP listener -- aka, the "Transport Handler"

Before getting into how to enable this new feature, let's first take care of a few obvious questions.

Question: does this mean the plugin no longer exists?

The plugin still exists. The plugin operates just as it did with V4.0, and all the information in this document pertaining to the plugin still applies.

Question: does the new feature have all the capabilities of the HTTP Server?

No (although the intent is to over time aggressively provide additional feature into the Transport Handler and make it the strategic HTTP handler for J2EE servers). The new Transport Handler is designed to listen for and accept HTTP requests off the network and get them over to the web container. To that end, things like the ability to run CGI programs and code `Pass` directives are not part of the design.

Two key restrictions do exist, according to "Assembling J2EE Applications" (SA22-7836-02):

- The Transport Handler does not support the authentication policy as specified in the deployment descriptor of the webapp. There is a way in which you can specify a surrogate ID that will be used for all requests.
- SSL can't be used as the transport between browser and Transport Handler.

Question: can the plugin and the new Transport Hander coexist?

Yes, with one caveat: both will bind to a TCP port; therefore, they must be configured to listen on different ports. The plugin (actually, the HTTP Server itself) will bind to whatever port you have coded on the `Port` directive in the `httpd.conf` file. The new Transport Handler will bind to whatever port you provide on the new configuration parameter `BBOC_HTTP_PORT=n` specified in the `current.env` file for the server. Those two values cannot be the same number.

Note: There is a mechanism in TCP/IP to allow two or more processes to share a TCP port. That subject is beyond the scope of this document. For now, the general rule of thumb is this: avoid having your `BBOC_HTTP_PORT=n` value conflict with another process and its port value.

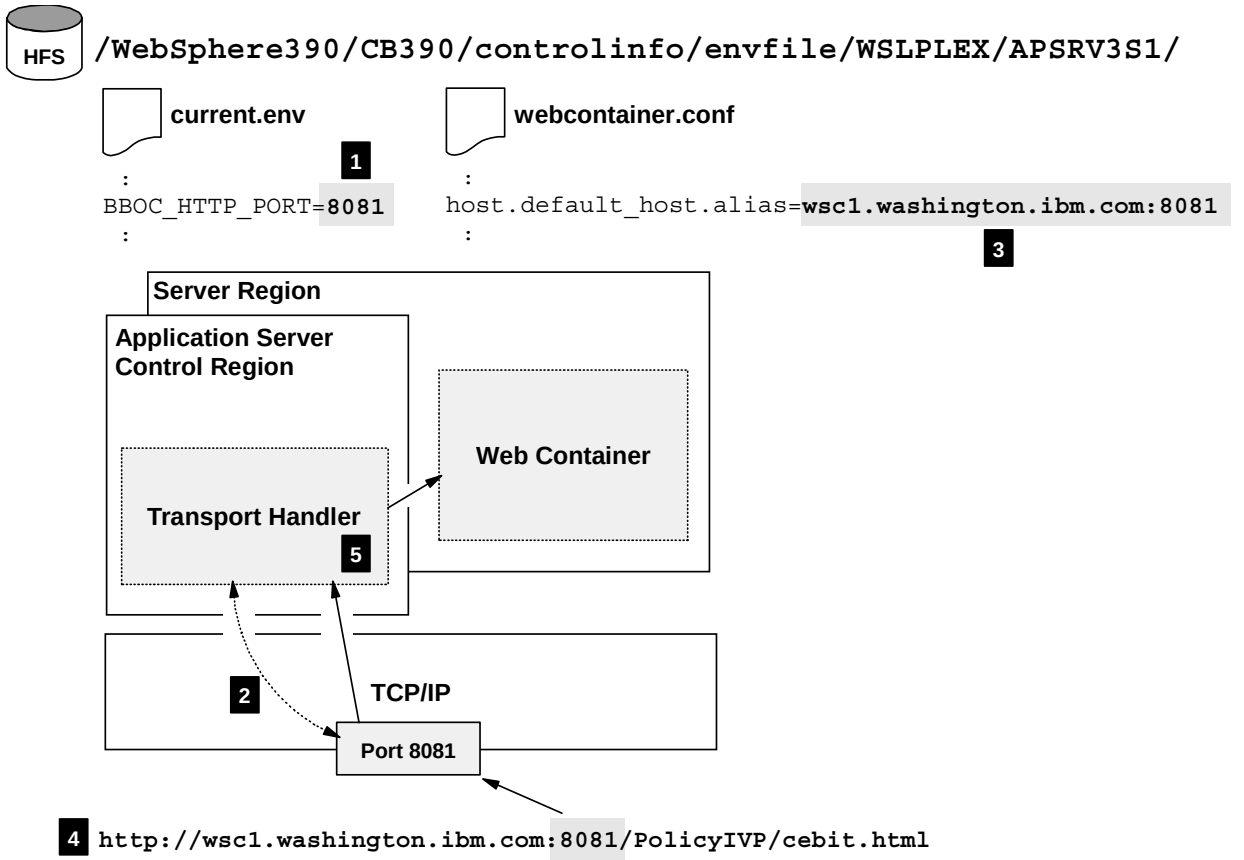
Please see "Question: how do I configure both the plugin and the Transport Handler for a given server?" on page 77 for more detail on how both can exist within a given configuration.

Question: does the introduction of V4.01 negate the information in this document?

No. The stuff contained here is still applicable.

Background: how the new Transport Handler works

The new Transport Handler is an integrated function of any application server you define to your system:



High-level of how the new Transport Handler operates

Let's walk through some of these things:

1. A new parameter (actually, several new parameters, but I'll only focus on one for the moment) is added to the `current.env` file found in the private directory of your J2EE application server. This parameter has the form:

   ```
   BBOC_HTTP_PORT=n
   ```

   where "n" is the port number on which you wish the new Transport Handler to listen.

2. The server is started and the new Transport Handler function binds to and starts listening on the specified port.

3. In the `webcontainer.conf` file for your application server's web container you have defined some number of virtual hosts, and you have applications that bind to those virtual hosts. This process is exactly the same as was the case with the plugin. In fact, all the information in this document relating to the issue of virtual hosts and binding applications still applies.

4. Someone at a browser somewhere issues a URL. If the network is operating as you hope, that URL will find its way to your TCP stack, and will get picked up by the port on which the Transport Handler is listening. This is business-as-usual network processing.

5. The Transport Handler takes the request off the port and compares it against virtual host and context-root pairs it knows about from the webcontainer.conf. If there's a match (and there is in this example), the request flows to the container to be serviced by the application. If no match occurs (for example, the virtual host is wrong, or the context-root string doesn't map to any bound application), the request is rejected.

If you've read the other sections in this document and you think it can't be this simple ... yes, it is this simple. Like anything, it can be made more complex with the use of the more advanced parameters. But to achieve basic functionality, that's all that's required.

Question: how many Transport Handlers can exist in a WAS environment?

As many as you have J2EE application server instances configured. The Transport Handler is implemented as a function of the server control region. It is intended to be the HTTP listening device for that server instance. So if you had ten J2EE application server instances configured, each with the Transport Handler configured, you would have ten Transport Handlers in the mix.

Since each Transport Handler will try to bind to the port specified by the parameter `BBOC_HTTP_PORT`, you could have conflict if you specify the same value for each instance's Transport Handler. If each instance is on a different LPAR, or a different TCP/IP stack, or if you've configured TCP/IP port sharing, you're okay. Otherwise, the server will fail to start if the port is already taken. Some care must be exercised to make sure you don't code a port value that will conflict with one already in use.

Note: This is no different from any other TCP application in that the port you define here must not already be bound by another application. If some other process has already taken possession of the port before you start your server, your server will fail to start when TCP rejects your server's request to bind to the port. As mentioned earlier, there is a way in TCP/IP to permit the sharing of a port. This document won't go into that subject.

Question: how do I get the new parameter into the current.env file?

There are two ways:

- Hand-edit the `current.env` file and include the new parameter. The downside to this is that the update will be lost the next time a conversation is activated in the SMS GUI tool. This is fine for initial ad hoc testing, but not a permanent solution.
- Use the SMS tool and update the "Environment Variable List" for either the Server or the Server Instance. The upside to this method is that the change will persist, even if subsequent conversations are activated.

Clearly the SMS tool is the proper way to do this for the long term.

The SMS tool will allow you to set environment variables at either the SYSPLEX level, the Server level or the Server Instance level. This brings up an interesting question: at which level should the variable be set?

Setting it at the SYSPLEX level makes no sense for that would mean every server would try to grab the same port. Setting different ports per Server *Instance* would technically work, but would imply coding different URLs based on the instance to which you wish to connect. The principle behind Server Instances is that *to a client* one instance is indistinguishable from another. Having different port numbers defeats this principle. Therefore, *the general recommendation is to set it at the Server level*. That means each instance will have the same port number. To avoid conflict on common port between instances, each started instance of a

server would have to be on different TCP/IP stacks, or in different LPARs (or through some fancy TCP/IP port sharing).

Question: how can I know the Transport Handler is ready to accept requests?

The most basic way is to issue a `TSO NETSTAT` command and check to make sure the Server *control region* is bound to and listening on the specified port:

```
WSIVP2A1 000027F8 0.0.0.0..8880        0.0.0.0..0          Listen
```

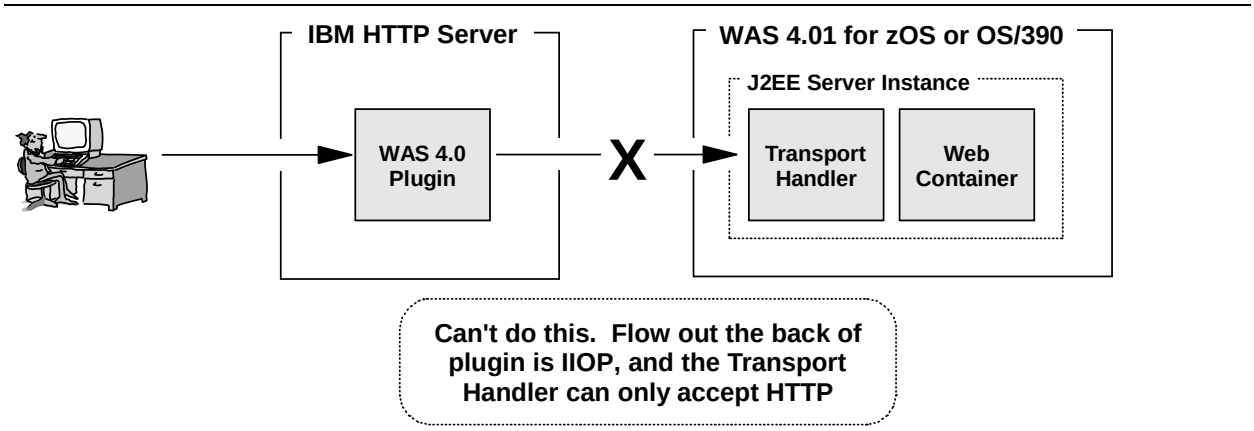
Note: The control region listens on several different ports and not just the Transport Handler port. So locate the control region name and look through all the ports it's holding.

Question: can the new Transport Handler listen on port 80?

Yes. The parameter `BBOC_HTTP_PORT` can be set to 80. That implies there's no webserver listening on the default port 80 somewhere on that TCP stack. Recall that if the port to which the Transport Handler will bind is already held by another process, the start of the server will fail. The Transport Handler may use port 80 so long as nobody else has it first (or you've instituted port sharing, which is a topic beyond the scope of this document).

Question: can I route requests from the plugin to the Transport Handler?

This question is illustrated with the following picture:



URL request can't flow from plugin to transport handler

The answer is "no." This question typically comes up for one of two reasons:

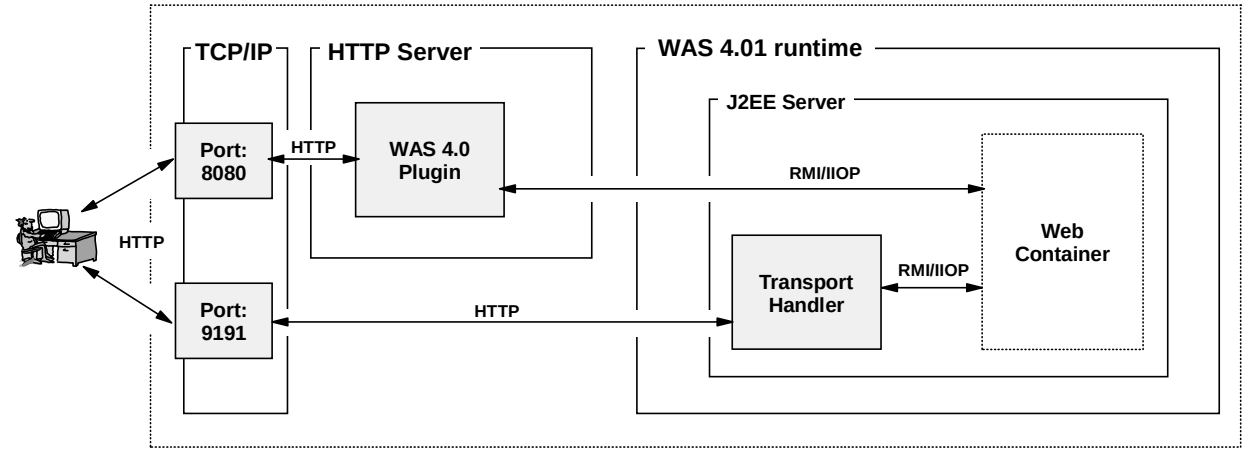
- What's really behind the question is a desire to design a DMZ and put an HTTP listener in the DMZ, or
- There's some confusion about how both the plugin and the new Transport Handler can be part of a single configuration.

These two issues are discussed in greater detail elsewhere in this document, as pointed to in the following table:

Issue:	Please refer to:
Unsure of how plugin and Transport Handler can be part of same configuration	"Question: how do I configure both the plugin and the Transport Handler for a given server?" on page 77
Desire to configure DMZ	"Question: how can I design a DMZ into my configuration?" on page 78

Question: how do I configure both the plugin and the Transport Handler for a given server?

As stated earlier, the plugin and the new Transport Handler may coexist in the same configuration. The picture of what this would look like is the following:



How both the plugin and the Transport Handler can coexist

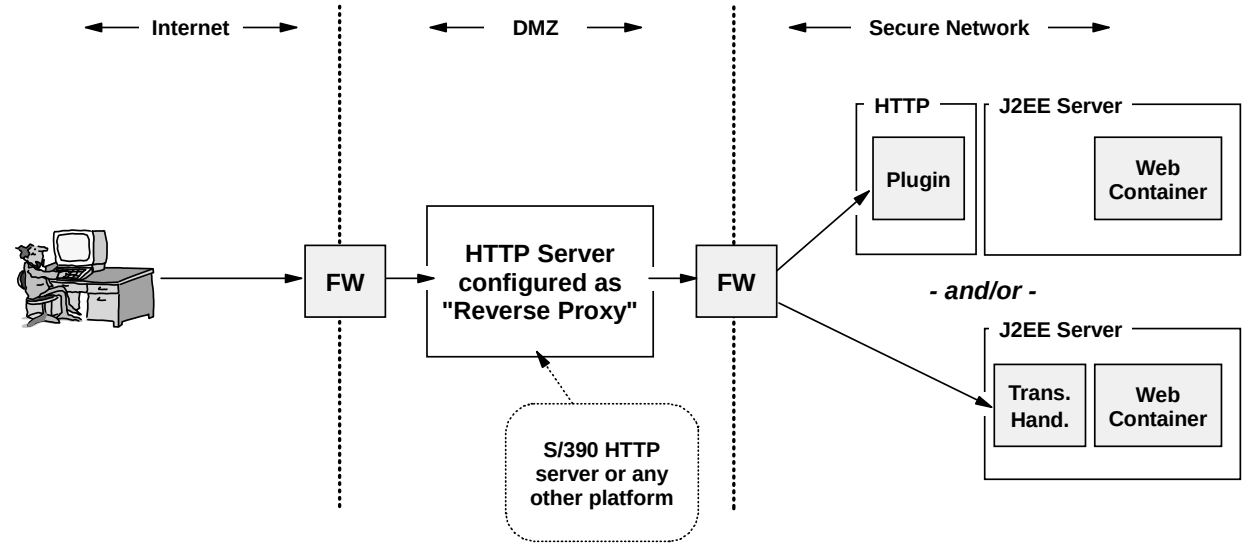
What this picture is illustrating is the plugin providing one path for a browser to reach webapps, and the Transport Handler providing a second. Notice how the two processes (plugin and Transport Handler) are on different ports. This picture illustrates a key point: there's really no coordination or connection between the plugin and the Transport Handler. They're two different things providing essentially the same basic function.

Question: why would I want to configure both the plugin and the Transport Handler?

If you presently have the plugin configured and serving webapps out of the WAS 4.0 web container environment, you may wish to maintain that environment while phasing in the Transport Handler. At the present time the Transport Handler has some notable restrictions (see "Question: does the new feature have all the capabilities of the HTTP Server?" on page 73), so using it and not the plugin may not be in your best interest at the present time. The plan is to enhance the function of the Transport Handler to make it the strategic HTTP listening device.

Question: how can I design a DMZ into my configuration?

The short answer is this: with a "reverse proxy server" in the DMZ and either the WAS 4 plugin or the new Transport Handler in the secure network:

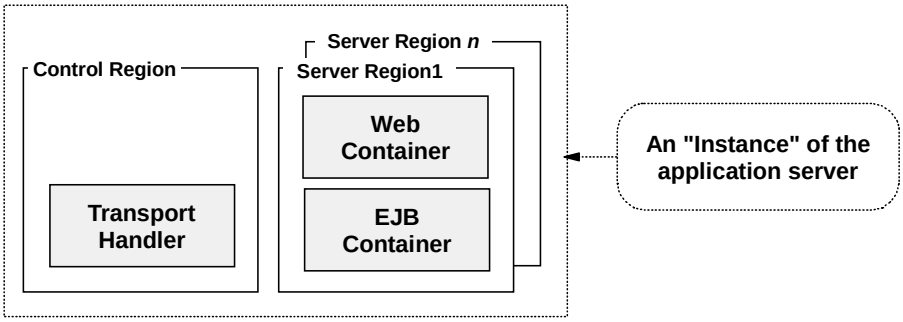


High-level view of DMZ configuration with WAS 4 webapps

This document won't get into how to configure an HTTP server as a reverse proxy. The reason why the reverse proxy is in the picture is due to some architectural constraints imposed by the plugin and the Transport Handler. Those two issues are covered next.

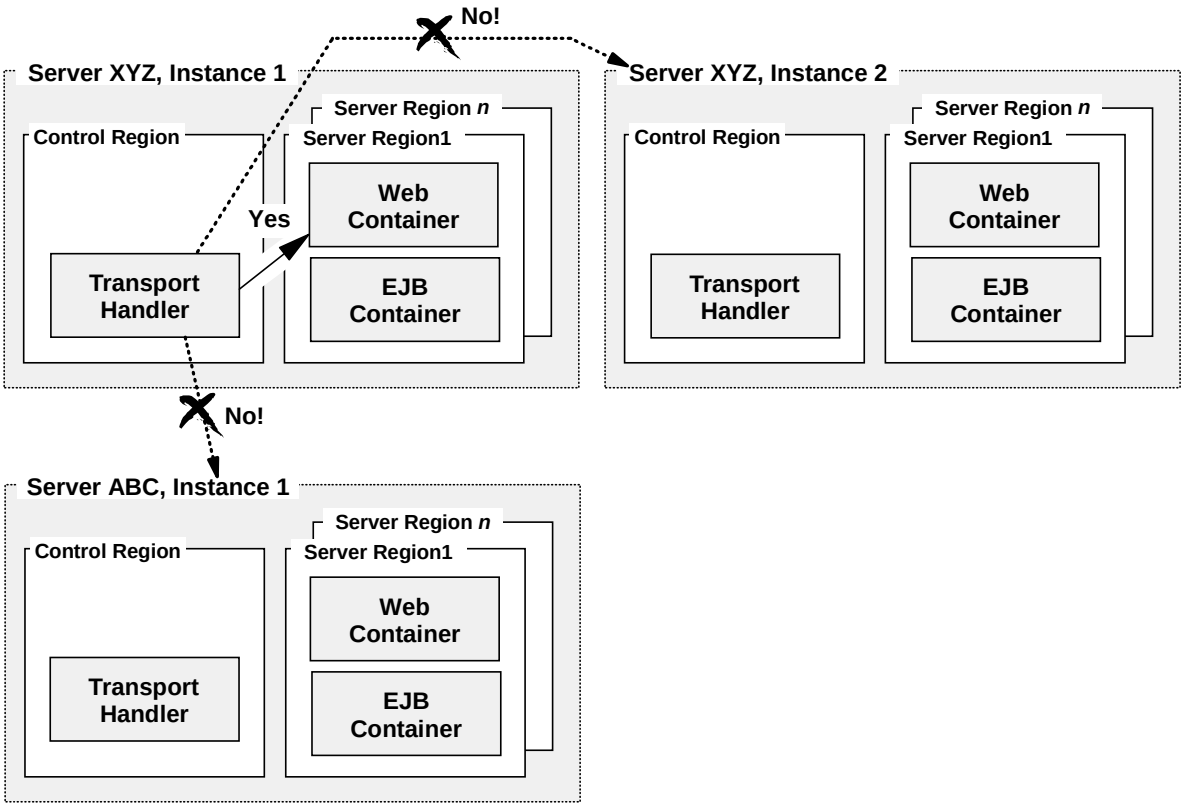
Background: why Transport Handler should not be in DMZ

The new Transport Handler function of WAS 4.0.1 is implemented in the *control region* of the J2EE application server. The web container and the EJB container reside in the *server regions(s)* of the server:



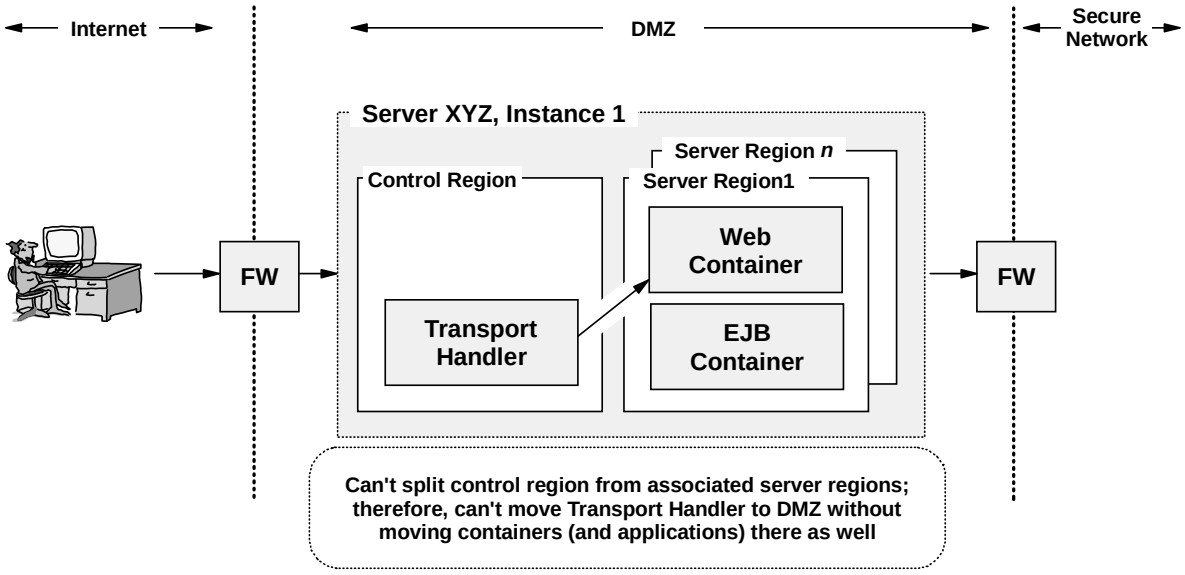
Where the Transport Handler resides relative to the containers

The Transport Handler serves as the HTTP listener for the web applications deployed in the web container of *that server instance*. The Transport Handler of one server isn't designed to route the requests to other server instances:



Transport Handler services only webapps in <u>its</u> instance server regions

Further, for any given instance of an application server, the control region and the server regions must reside in the same MVS image. *You can't split the control region and server regions*. Therefore, you can't move the Transport Handler into the DMZ without also moving the server regions into the DMZ:



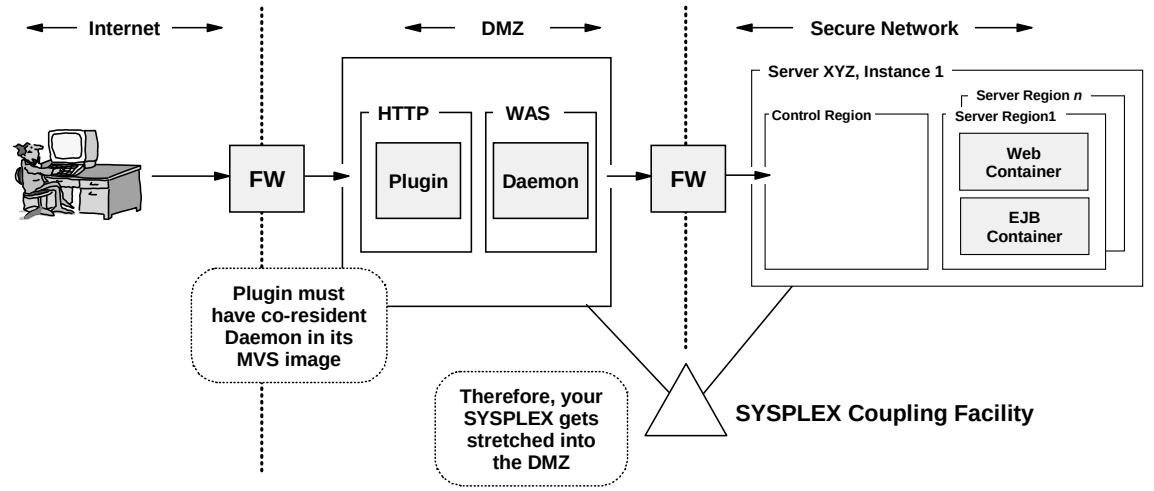
Can't move Transport Handler into DMZ without dragging applications into DMZ as well

Therefore, trying to move the Transport Handler to the DMZ doesn't make much sense because your applications are moved into the DMZ as well. But what about the plugin? Can you move that to the DMZ? That's discussed next.

Background: why WAS Plugin should not be in DMZ

There are two reasons why you don't want to move the WAS 4 Plugin out into the DMZ:

- The flow from the Plugin to the web container is RMI/IIOP, and configuring the firewall to carefully flow IIOP between the DMZ and the secure network is a more challenging task than configuring it for HTTP.
- The Plugin has a requirement that it run in an MVS image that also includes a running copy of the WAS daemon region. That implies stretching the SYSPLEX into the DMZ to accomodate that requirement:

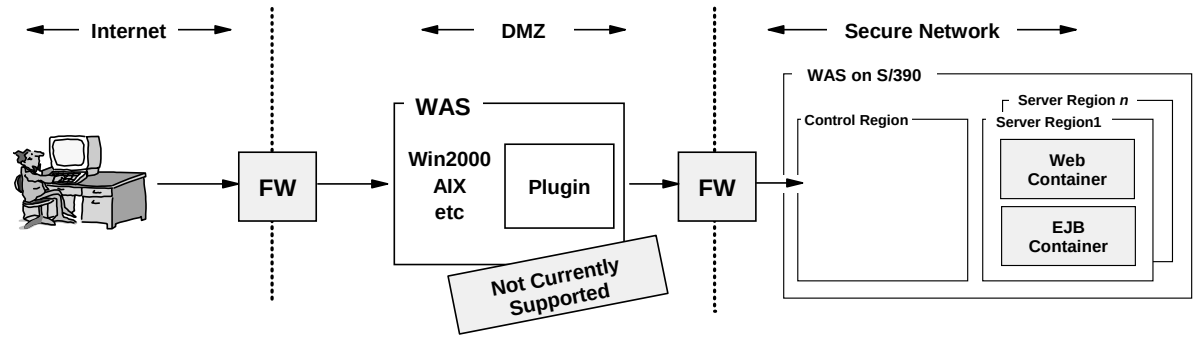


Moving Plugin to DMZ stretches SYSPLEX into DMZ as well

This is not a good thing. Therefore, trying to move the WAS 4 Plugin into the DMZ isn't recommended.

Question: can the WAS Plugin be configured on a distributed platform in the DMZ?

The next logical question is this: may I configure WAS on one of the distributed platforms out in the DMZ and use its HTTP listening capability to route requests to the webapps?



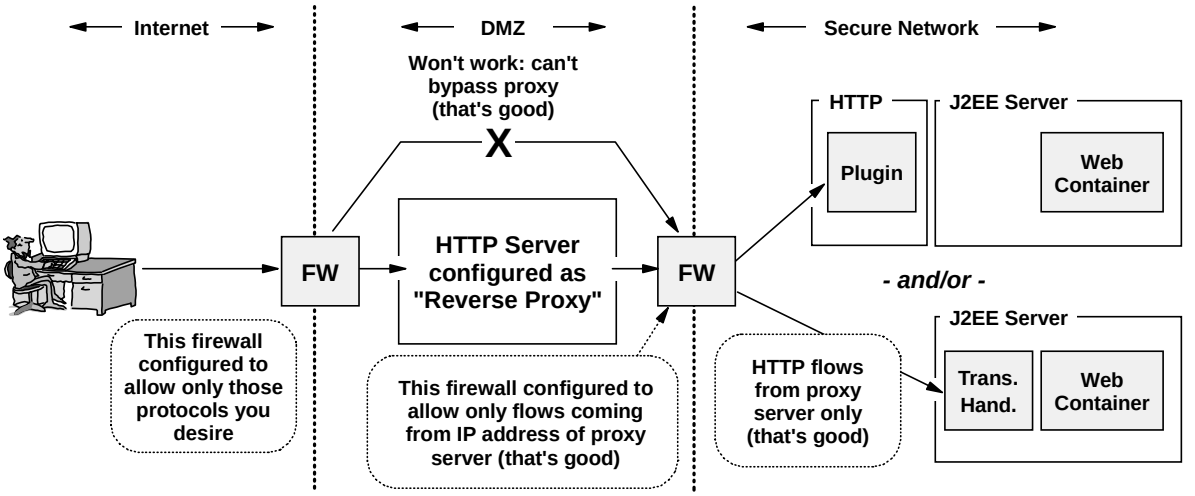
Unsupported configuration of distributed platform WAS in DMZ acting as "plugin"

And the answer, as indicated in the picture, is no. This configuration is not currently supported.

Note: Change the picture slightly to have a standalone S/390 in the DMZ and the answer is still "no." The plugin must co-reside with a running copy of the WAS/390 Daemon, and that daemon must be part of the SYSPLEX containing the web container to which you wish to communicate.

That leaves a reverse proxy in the DMZ

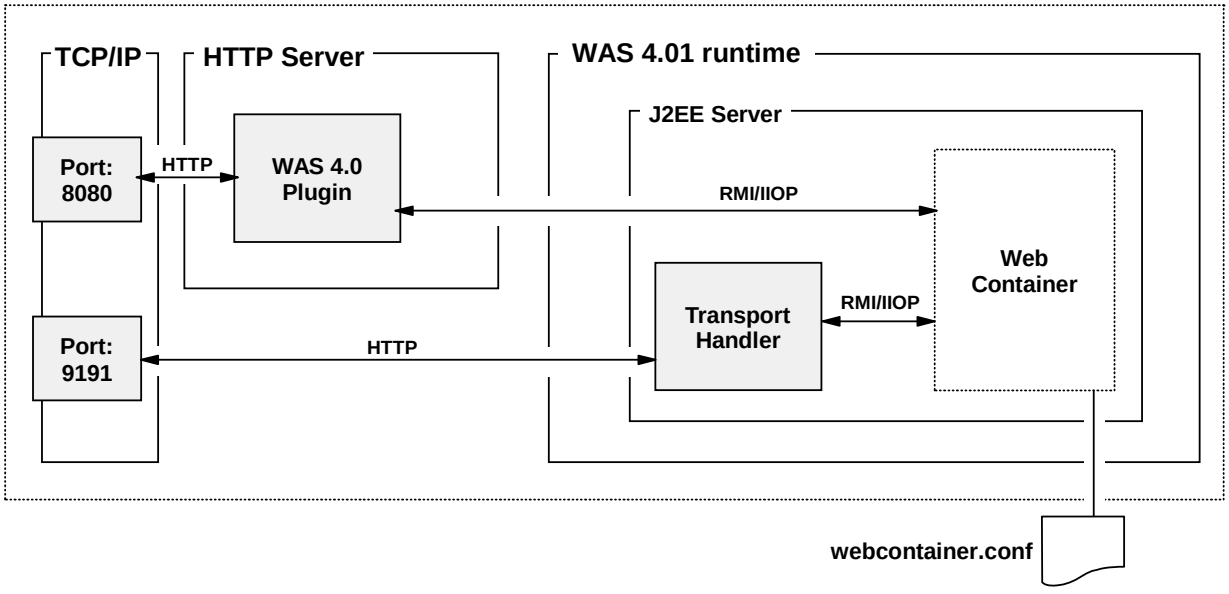
And so we're back to the original "short answer" to the question. By placing an HTTP server in the DMZ -- on any platform and operating system, by the way; the reverse proxy function is fairly standard and not a WAS 4 function -- you achieve the desired results:



Background: binding your webapp to a properly defined virtual host

The issue of binding applications to virtual hosts is largely separate from the issue of which HTTP listener you use. You still need to define virtual hosts in your `webcontainer.conf` file, and you still need to code the `contextroots=` statement in that file so web applications can bind to the virtual host. But if you have *both* the plugin and the Transport Handler configured, things may at first look confusing, particularly as viewed from the "Application Dispatching Information" panel put out by the plugin.

Consider the following picture, which illustrates an environment with both the plugin and the Transport Handler configured:



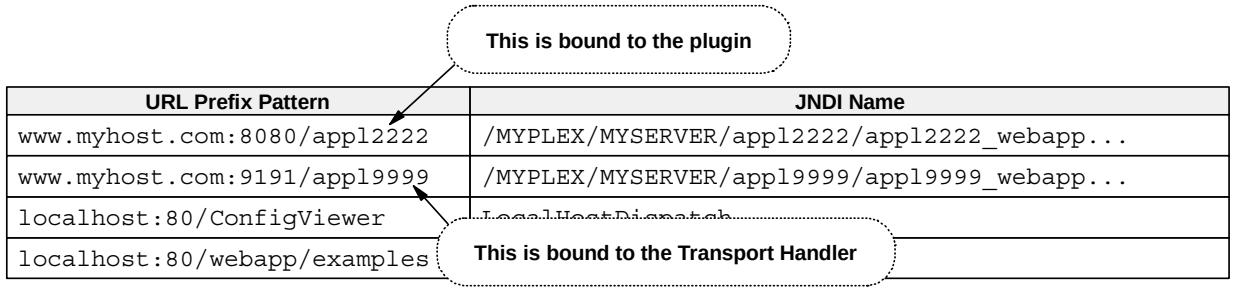
Example where both plugin and Transport Handler configured

Unless you configure TCP/IP port sharing, the plugin and the Transport Handler will listen on different ports. And for all ports other than the default 80, the port number must be specified along with the IP host name on the "alias" statement in the `webcontainer.conf` file (see the illustration above).

With different port numbers, the two different listeners are accessed with two different virtual hosts. You would also need to take care to make sure the `contextroots=` statements allow the various applications to bind to the virtual host you intend.

Note: It is possible to bind an application to two (or more) virtual hosts. This is illustrated at "Question: can I bind same application to both plugin and Transport Handler virtual host?" on page 83.

The plugin's "Application Dispatching Information" process will happily show you all the applications that are bound to virtual hosts in the web container -- for both the plugin's own virtual host as well as the virtual host of the Transport Handler:



Application Dispatching showing applications bound to both plugin and Transport Handler

Is this a problem? No, not at all. But it could become one if you get your applications and virtual hosts mixed up in your mind and you end up coding the wrong URLs to access the applications:

URL Prefix Pattern	JNDI Name
www.myhost.com:8080/appl2222	/MYPLEX/MYSERVER/appl2222/appl2222_webapp...
www.myhost.com:9191/appl9999	/MYPLEX/MYSERVER/appl9999/appl9999_webapp...
localhost:80/ConfigViewer	LocalHostDispatch
localhost:80/webapp/examples	LocalHostDispatch

http://www.myhost.com:8080/appl9999/servletmapping

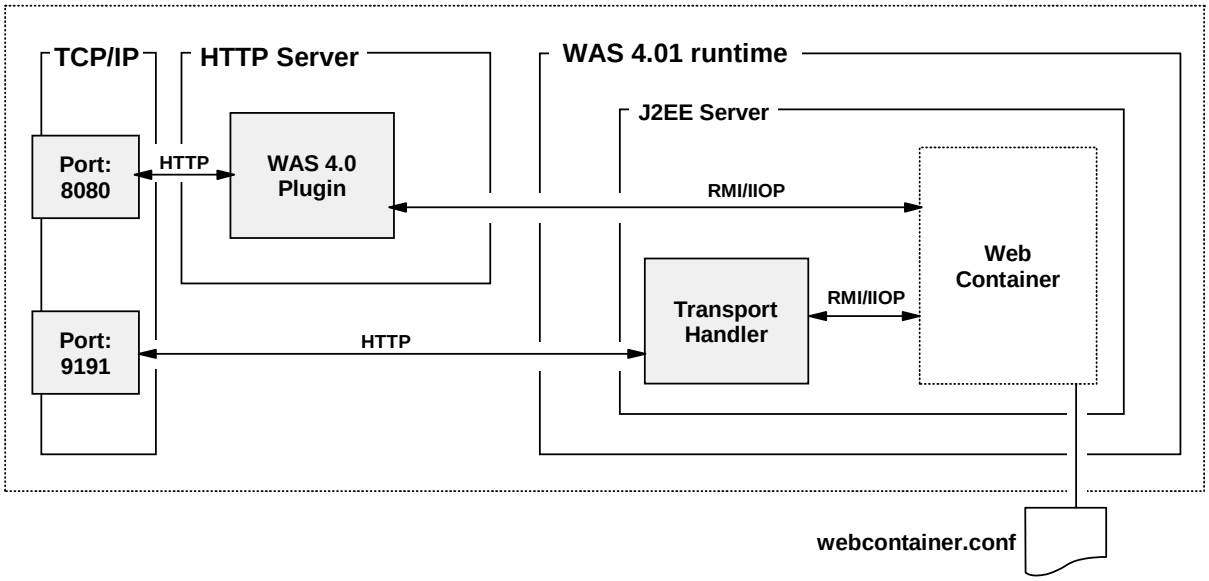
This will fail because the port supplied on the URL doesn't match the port on the virtual host for the application. This is an easy error to make, particularly if you've become accustomed to the port for your webserver plugin and you configure the Transport Handler for the first time.

URL that will fail because the host doesn't match the virtual host for the application

The message here is to be careful and make certain your URLs match the virtual hosts to which the application is bound.

Question: can I bind same application to both plugin and Transport Handler virtual host?

Yes. This is accomplished by coding two different host values on the `alias=` statement, and having your `contextroot=` statement point to that alias statement:




```
host.both.alias=www.myhost.com:8080,www.myhost.com:9191
  :
host.both.contextroots=/appl2222
```

How to bind an application to both the plugin and Transport Handler

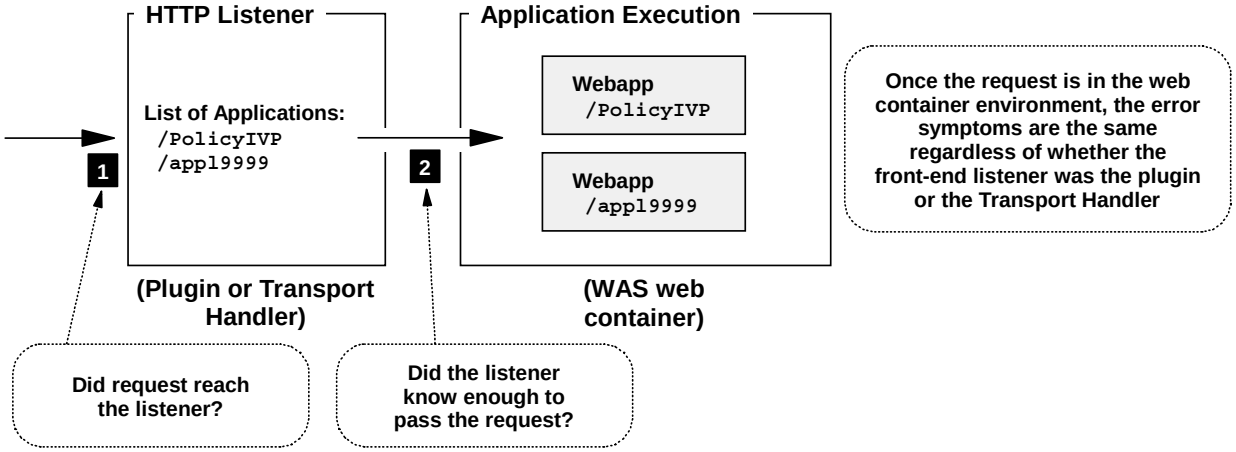
In this example, the application `/appl2222` is bound to both `www.myhost.com:8080` and `www.myhost.com:9191`. That means you can access the application through either the plugin or the Transport Handler.

Question: what other Transport Handler parameters are available?

"Appendix A, Environment and JVM properties files" of the manual "Assembling J2EE Applications" (SA22-7836-02) covers in detail the eight `current.env` parameters related to the new Transport Handler, including `BBOC_HTTP_PORT=n`

Background: error conditions and the Transport Handler

Back in "Common Configuration Errors and the Symptoms Displayed" starting on page 50 the topic of error handling was covered when the plugin was the HTTP listener. For the most part the error symptoms are the same when the listening agent is the Transport Handler. Why would this be the case? Because the HTTP "catcher" (be that the plugin or the Transport Handler) has a rather limited role in all of this:



HTTP "catcher" and the limited potential for error conditions

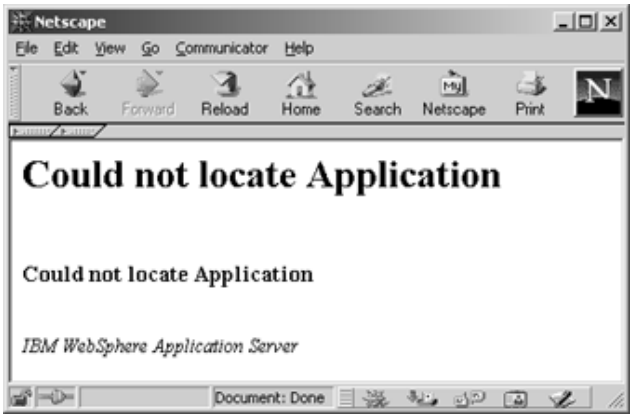
The first condition is caused by errors such as mis-typing the URL, or having the wrong port number, or having a router down in the network, or some other error that would prevent the HTTP request from reaching the desired listening port. The symptom you'll see here is typically some kind of browser pop-up window indicating the request didn't connect to anything out there on the web. These problems are beyond the scope of this document.

Once the request is into the listener, the world of the Transport Handler is *much simpler* than the plugin. All the issues relating to the `ServerInit`, `Service` and `ServerTerm` directives in `httpd.conf` go away; all the issues about whether the plugin is connecting to the SMS server go away; and all the issues about whether the application request will run "locally" or "remotely" go away. You're left with a very simple environment: does the URL match any known application deployed in the web container?

It appears the Transport Handler uses a two-level check to see if the received request matches a known virtual host, and then whether the context root matches. If your virtual host doesn't match you get one error message, but if your virtual host is correct but the context root wrong you get a different error message.

Error: when the virtual host doesn't match

If your URL gets to the Transport Handler but the virtual host you have coded (the IP host plus the optional port specification) doesn't match what the Transport Handler knows about, it'll fail with the following:



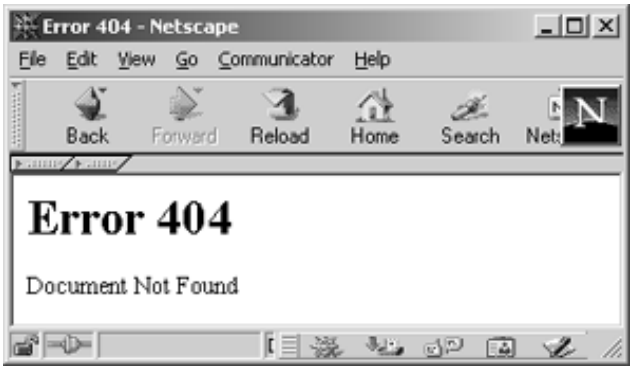
Transport Handler's message when virtual host doesn't match

This error will occur most often as a result of you mis-typing the virtual host value in the `webcontainer.conf`. The URL coming from the browser will resolve to the proper IP address based on the DNS entries, and it'll get to the Transport Handler. The Transport Handler will then try to match it against the known virtual hosts defined in the `webcontainer.conf` file for the server instance. If you made a mistake in the virtual host string in the `webcontainer.conf`, no match would occur and this error would result.

There's a slightly less probable reason for this error: your DNS allows more than one host specified on URLs to resolve to the IP address of your server. Let's say your DNS allows `www.myhost.com` *and* `www.yourhost.com` to resolve to `9.85.101.200`. Let's say that your `webcontainer.conf` has only `www.`**`myhost`**`.com` specified as a virtual host alias. You send a URL with `www.`**`yourhost`**`.com`, the DNS resolves it to your server, and the request makes it to your Transport Handler. The virtual hosts the Transport Handler will pass back to the web container is limited to what's in the `webcontainer.conf` file, or `www.myhost.com` in this example. Presto: you're rejected with the "Could not locate Application" message.

Error: when the virtual host is correct but the context root is wrong

Assume that your virtual host correctly matches what's in `webcontainer.conf`. The next thing the Transport Handler will check is the context root setting. If that doesn't match any of the values for bound applications, you'll get this error:



Transport Handler's message when context root value doesn't match

The problem will arise for two reasons:

- The URL's context root portion isn't typed correctly
- The actual context root of the bound application is different from what you think it is

These are really the same thing, both pointing back to a mis-match between the context root on the URL and the list of context roots for the virtual host.

Error: when the virtual host and context root are right but the servletmapping is wrong

If your virtual host and context root are correct, the request will get "thrown over the wall" into the web container. At that point, the error messages you'll see will come from the web container. The type of HTTP listener you're using doesn't make a difference. Those errors are detailed *starting* at "Errors related to request not resolving to web application class file" on page 63.

Security Issues

Note: The topic of security is a broad one and outside the scope of this document. However, as new topics arise that relate to the overall topic of configurating web applications, those topics will be included here until they are incorporated into other documents directly related to security. At that time the information will be removed from here and a pointer provided to the other document.

HTTP Authentication based on web.xml definitions rather than Protect statements

Authentication involves challenging the user at the browser to supply a userid and password. That user is allowed to proceed only if the information they supply is properly validated. Basic authentication has long been a feature of the IBM HTTP Server. That authentication may now be performed based on security definitions in the web applications deployment descriptor (`web.xml` file inside the WAR file) rather than with `Protect` statements in the `httpd.conf` file. However, at the time of this writing the IBM HTTP Server is still required to be part of the picture.

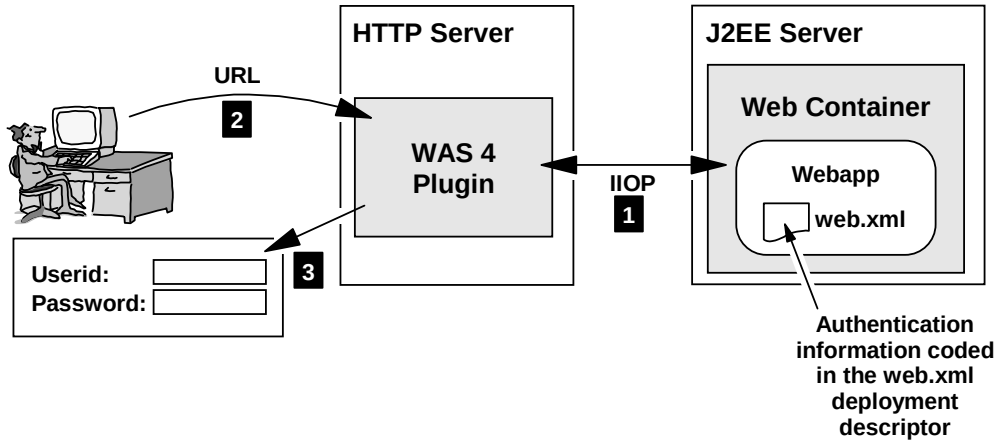
Quick summary of updates required

The following chart summarizes the updates required to enable HTTP authentication out of the web container. These are *in addition to* normal updates required as discussed in other parts of the document.

`httpd.conf`	None (no `PROTECT` statements for the resource are needed)
`httpd.envvars`	`JAVA PROPAGATE=NO`
`was.conf`	None
`webcontainer.conf`	None
WAR file contents	`web.xml` file updated with: • URL resource to be protected specified in `<security-constraint>` stanza with `<url-pattern>` tag • Define the `<auth-constraint>` role to be applied to this protected resource (on `<role-name>` tag) • Define `<security-role>` with the RACF `EJBROLE` `<role-name>` to which authenticated users must have access See "Background: example of definitions in web.xml file" on page 89 for an example of these definitions.
RACF	• Grant HTTP Server's ID `READ` access to `CB.CBIND.`*`server_name`* profile • Grant `READ` access for each authorized userid to the `EJBROLE` profile defined in the `web.xml` file of the web application's WAR file.

Background: how the Web Container performs HTTP authentication

This works as a coordinated effort between the definitions in the `web.xml` file and the HTTP Server:



High-level view of HTTP authentication from web container

1. The web container reads the deployment descriptor of each deployed webapp and passes to the Plugin information related to HTTP authentication.
2. The user sends in a URL that matches the `<url-pattern>` defined in the `web.xml` file. The `<url-pattern>` tag defines the template, or mask, used to match against an inbound URL to determine if protection is required.
3. If a match on the `<url-pattern>` value is made, the plugin asks the HTTP server to pop the logon window requesting the user's userid and password.

   Note: The webserver pops the login panel, but *not* based on a `Protect` statement in its `httpd.conf` file. This is based on information passed it by the web container regarding the security constraints defined in the `web.xml` file for deployed webapps in the container.

If the userid and password is valid *and* the userid has at least `READ` access to the `EJBROLE` profile defined in the `web.xml` file, the user is allowed to access the resource implied on the URL.

Question: should I still code the Protect directive in the httpd.conf file?

You should *not* code authentication in both the webapp's deployment descriptor (`web.xml` file) *and* the `httpd.conf` file. Pick one or the other, but don't use both for the same URL resource.

The servlet specification 2.2 defines the location of security constraints and authentication rules to be the webapp's deployment descriptor. If you wish to develop and deploy web applications that adhere to that specification, you should code your security constaints in the deployment descriptor and *not* code `httpd.conf Protect` statements.

Question: can I use this with the new Transport Handler?

At the time of the writing of this version of this document, no. But plans are in place to incorporate into the Transport Handler this function in the near term.

Background: example of definitions in web.xml file

The following example illustrates the contents of the `web.xml` deployment descriptor that relate to HTTP authentication:

Note: The `web.xml` file is typically generated by the tooling program used to create the webapp (for example, WSAD). You wouldn't normally hand-edit the `web.xml` file to set these properties.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Sample Web Resource Collection</web-resource-name>
    <url-pattern>/secret/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>PUT</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description>Sample Security Constraints:+:</description>
    <role-name>Manager</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Sample_Security_Realm</realm-name>
</login-config>
<security-role>
  <description>RACF EJBROLE Manager</description>
  <role-name>Manager</role-name>
</security-role>
```

Example of web.xml deployment descriptor and authentication properties

The `<url-pattern>` tag defines the URL string that, if matched, signifies a URL that is to be protected. In this example any URL with `/secret/*` is to be protected. The `<role-name>` tag defines the `EJBROLE` profile that applies to this protection mechanism, in this example `Manager`. Anyone who wishes to access the URL `/secret/*` resource must have `READ` access to the `Manager` profile.

Activity: set security constraint properties for your webapp

This will be done in the tooling you use to create your webapp, and each tool has a different way of setting these values. The bottom line is the WAR file you wish to deploy into the J2EE server must have the `web.xml` file updated with the security constraint values.

Activity: httpd.envvars

- ☐ Code `JAVA_PROPAGATE=NO` in the `httpd.envvars` file. Failure to do that will result in the authentication failing with an obscure message in the "ncf" log:

  ```
  login for userid <userid>with password failed -- rc = 9D000498
  ```

- ☐ Restart the HTTP Server to pick up this new environment variable.

Activity: RACF updates

- ☐ Grant the HTTP Server's ID (the ID under which the server process runs) `READ` access to the `CB.CBIND.`*`server_name`* profile, where *`server_name`* is the name of the J2EE server in which the web container resides.
- ☐ Grant `READ` access to the `EJBROLE` profile named in the `web.xml` file to every userid you wish to have authority over the URL resource. For example, if the `EJBROLE` name

is `Manager`, and you wish the userids `SMITH` and `JOHNSON` to have access to the resource, then grant both userids `READ` access to the `EJBROLE` profile `Manager`.

Question: what's the advantage of web containter authentication vs. Webserver?

The two provide essentially the same level of authentication. However, specifying authentication within the security constraint of a web application's deployment descriptor is one of the things defined in the servlet specification 2.2. For a web application developer who wants to abide by the J2EE specification, coding security constraints within the webapp's deployment descriptor is the correct method.

As stated earlier, to accomplish this you need the WAS 4 plugin running inside the HTTP Server. The plugin has been developed to communicate with the web containers and to understand the security constraints defined in the webapps deployed there. At the present time the Transport Handler is not capable of performing this function, though that should be corrected in the near term.

Advanced Webapp Topics

This section contains information that is more advanced than basic configuration and validation. *It is by no means a comprehensive reference for all such information*: the WAS 4.0 manuals serve that purpose. This section will contain things we at the Washington Systems Center came across in our testing and development and thought might be useful to include in a document such as this.

Note: This section is frequently updated with new information. Check the date in the footer of each page and compare against other copies of this document to see if you have an older version.

Background: WebSphereSampleApp.ear shipped with WAS

This sample application provides a function very similar to what is provided in the `/webapp/examples` application in the plugin. (See "Activity: check plugin Application Dispatching Information" on page 29.) The difference is one runs out of the plugin and the other runs out of the web container in the WAS 4 runtime.

The reason for bringing this up in this document is this: the `<context-root>` value for the webapp inside `WebSphereSampleApp.ear` is `/webapp/examples`, which is *the same as the supplied servlet for the plugin*. That means if you deploy `WebSphereSampleApp.ear` into your WAS 4.0 web container, the issuance of the URL: `/webapp/examples/index.html` may result in the request being serviced by the plugin, or it may result in the request being serviced by the web container. How can you tell which is which? By what's on the page:

From Plugin:

From Web Container:

"Webapp Examples" from plugin vs. web container

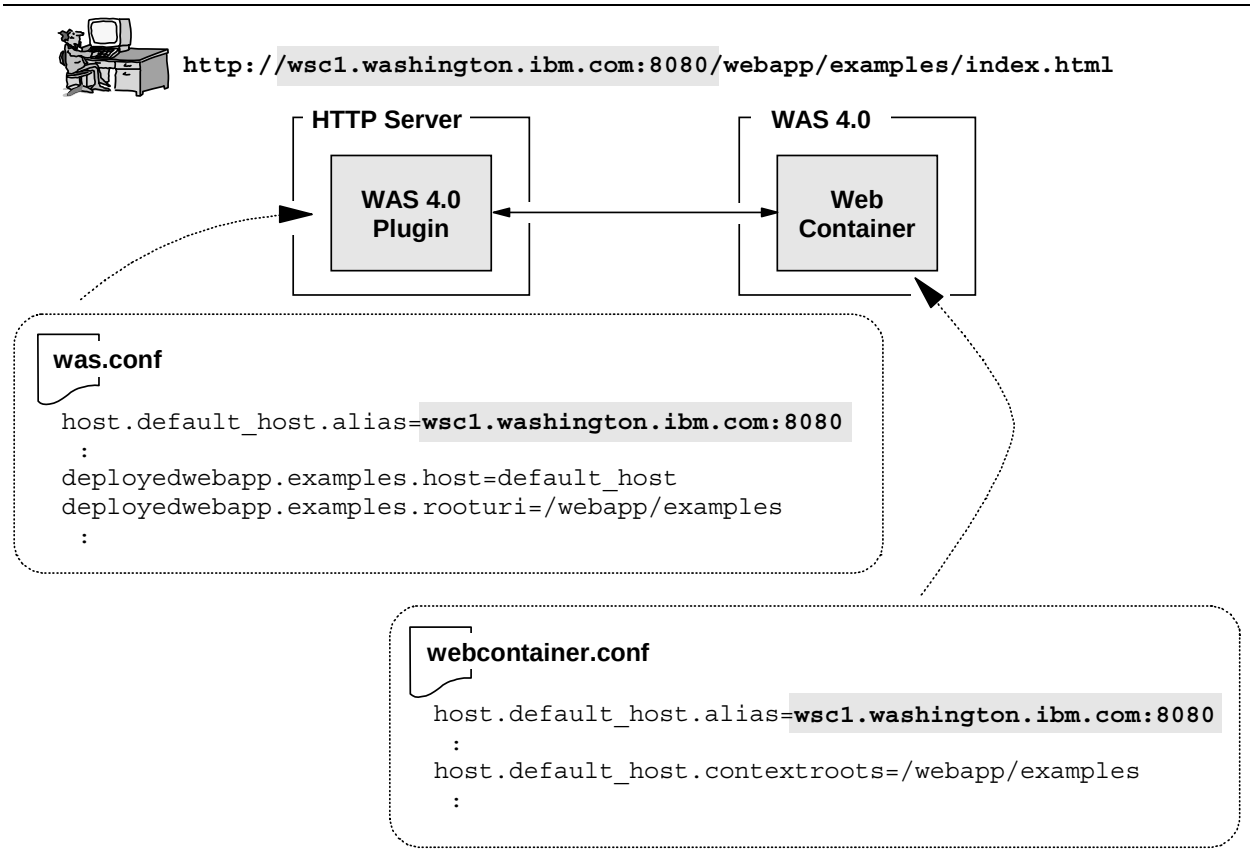
The key differences are the inclusion of the "Sample Error Servlet" included with `WebSphereSampleApp.ear`, and what results when you click on "Show server configuration."

If you have a `rooturi=/webapp/examples` in the plugin and a bound application with `<context-root>/webapp/examples</context-root>` in the web container, which one takes precedence? It depends on which environment -- plugin or web container -- has the exact match on virtual host to the URL received. That's discussed next.

Background: when same virtual host is defined in both environments

For those familiar with Websphere 3.5 Standard Edition, you'll know that the concept of "virtual hosts" was present there as well. Virtual hosts for applications are defined in the `was.conf` file, and provide pretty much the same function they do in the `webcontainer.conf` file.

The reason this is being brought up is because how the virtual hosts are defined in both places may have a bearing on where an application is run. This is particularly true if you have deployed the `WebSphereSampleApp.ear` file into the WAS web container, because its "context root" is the same as the plugin's default IVP application. Consider the following picture, which illustrates this:



Virtual host definition in both the plugin vs. the web container

In this example the same virtual host is defined in both the `webcontainer.conf` file and the `was.conf` file, and an application that'll respond to `/webapp/examples` is deployed in both environments. In this case, *the request will invoke a webapp in the plugin environment*.

Rule: Whenever there is a match on the [virtual host | context-root] pair in *both* the plugin and the web container, the plugin takes precedence. If the match is made only in the web container, the request will flow to the web container. If the match is made only in the plugin, then naturally the request stays with the plugin. If it matches both, then the plugin takes it.

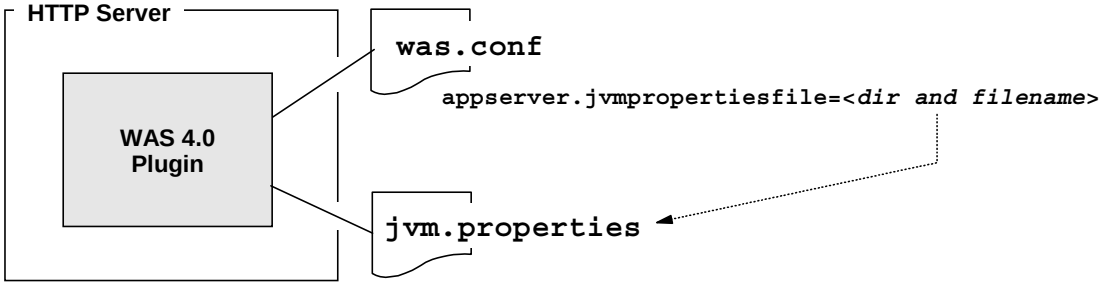
Back in "Background: use of localhost value for virtual host" on page 23 it was stated rather emphatically that the value `localhost` should *not* be used for the virtual host in the `webcontainer.conf` file. The value `localhost` is also available for use in the `was.conf`

file, and while its use in the was.conf is also discouraged, it's easier to use in the plugin than it is in the web container. *An exact match on an explicit virtual host will always take precedence over the value localhost*. Therefore, an application deployed in the webcontainer with an explicit virtual host coded will take precedence over the same application deployed locally in the plugin when `localhost` is the virtual host.

Background: the plugin's JVM properties file

The new WAS 4.0 plugin runs inside the HTTP Server's address space, but interestingly has within the plugin itself a copy of the JVM (Java Virtual Machine). That makes sense: the plugin is capable of running servlets (just like WAS 3.5 SE did), so it would need a JVM to do that. That copy of the JVM uses, like all JVMs everywhere, a "properties" file. The JVM properties file provides the JVM information about how it is to behave.

For most people and their plugin environment, the *default* JVM properties file is what is used. That's because most instructions (including those provided earlier in this document) don't indicate to do what's necessary to bring into play a custom copy of the JVM properties file (the default is fine for most implementations). But to do some of the "advanced" things you need to modify the plugin's JVM properties, and to do that you need to point your WAS 4.0 plugin to its own custom copy of the file:



The pointer to a custom JVM properties file

The default copy of the properties file (the one that is used if you don't specify any directory or filename on the `appserver.jvmproperties` statement) is the following:

`/usr/lpp/WebSphere/WebServerPlugIn/properties/default_global.properties`

It is this file you would copy to a custom directory and point to out of your `was.conf`. Then you could update your copy of the file with some of these advanced functions.

Activity: create custom JVM properties file for your WAS 4.0 plugin

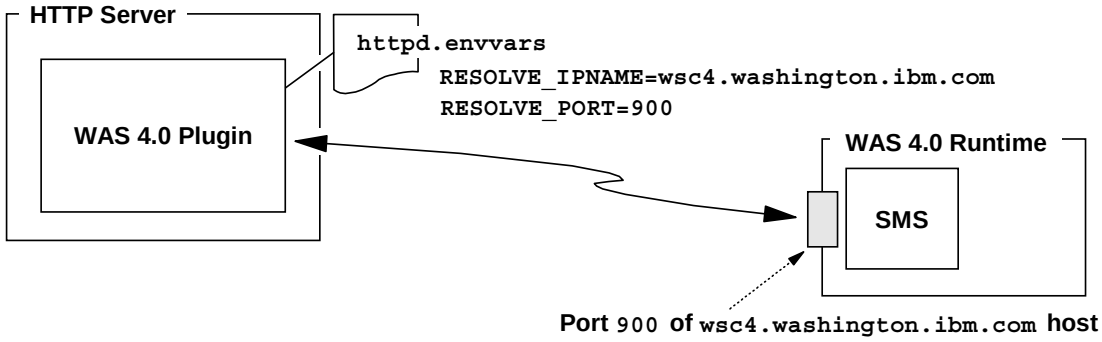
- ☐ Copy `default_global.properties` from the from the `/WebServerPlugIn/properties` directory to the directory in which your plugin's `was.conf` file resides. Make sure the file has permissions of at least 644.
- ☐ Rename the copied file to something other than `default_global.properties`. It is no longer a "default" properties, so something like `jvm.properties` would be better.
- ☐ Edit `was.conf` and update the `appserver.jvmproperties=` statement and provide the directory and filename of your copied and renamed file.

Your plugin now has its own copy of the JVM properties file. The change won't take effect until you restart the webserver.

Background: how the plugin communicates with the WAS 4.0 runtime

For the plugin to do its job, it needs to know the hostname and port number for the Systems Management Server (SMS) to which you wish the plugin to connect. You tell it this in the

`httpd.envvars` file, where the `RESOLVE_IPNAME` and `RESOLVE_PORT` variables provide that information:



Pointers to the SMS of the runtime serviced by the plugin

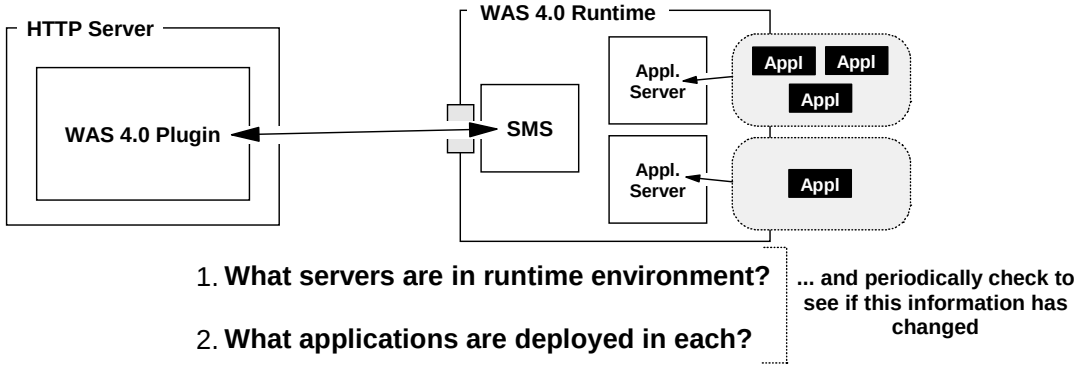
The following restrictions apply to where the plugin may reside relative to the rest of the WAS 4.0 runtime:

- The plugin must reside in the same Sysplex as the WAS 4.0 SMS to which the plugin will communicate
- The plugin must reside on a box or LPAR with a running instance of the WAS 4.0 daemon (BBODMN).

This brings up some interesting questions about constructing a network topology that includes a "demilitarized zone" (DMZ). You could extend your Sysplex out and include the system with the plugin out in the DMZ, but that's generally not recommended. The recommendation is to put an HTTP Server out in the DMZ and configure it as a "reverse proxy," and have it communicate with the WAS 4.0 plugin running in the secure network. This topic was explored back in "Question: how can I design a DMZ into my configuration?" on page 78.

Background: what the plugin wants to know from the SMS

There are two basic pieces of information for which the plugin is interested:



Two basic things the plugin wants to know

How to modify the default behavior of the plugin for these things is provided next.

Activity: how to limit the number of J2EE servers with which the plugin will communicate

The default behavior of the plugin is seek knowledge of *all application servers* in the runtime environment. But you can limit the plugin to maintain knowledge of a specified list of servers. This would be applicable in an environment where, for example, you had twenty application servers, one of which had web applications deployed and nineteen of which did not. Why

spend energy checking those nineteen servers you know don't contain web applications? With this setting you could limit the search to just the server you wish:

- ☐ Edit the plugin's `jvm.properties` file (not the 4.0 runtime's application server JVM properties, but the new plugin's JVM properties. See "Activity: create custom JVM properties file for your WAS 4.0 plugin" on page 93 for a discussion of setting up a custom JVM properties for your plugin).
- ☐ Set the following property:

  `com.ibm.ws390.wc.includeWebContainers=`*<list of servers delimited by a coma>*

  Note: This provides a way to limit down to the server level, but not all the way down to the server instance level. So if you had a server -- let's say `APSRV3` -- and that server had five instances defined (let's say `APSRV3S1, APSRV3S2 ... APSRV3S5`), then the property:

  `com.ibm.ws390.wc.includWebContainers=APSRV3`

  would limit the plugin's scope to the web containers in the five instances defined for the server `APSRV3`.

  You may code more than one server on this property. Simply separate each server with a comma and a space:

  `com.ibm.ws390.wc.includWebContainers=APSRV3, APSRV4, APSRV5`

- ☐ Stop and restart the webserver to pick up this change.

Activity: how to alter the interval between which the plugin checks for new J2EE servers

By default the plugin will poll the SMS every 10 minutes to see if additional servers have been defined. You may wish to set a longer time, particularly if your environment is relatively stable regarding the number of servers configured.

Do the following:

- ☐ Edit the plugin's `jvm.properties` file.
- ☐ Set the following property:

  `com.ibm.ws390.wc.serverCheckInterval=`*<interval in minutes>*

- ☐ Stop and restart the webserver to pick up this change.

Activity: how to alter the polling interval used by the plugin to check for new applications

By default the plugin will query *each application server* every two minutes to see if any new applications have been deployed. You may wish to set this value higher, particularly if your rate of new application introduction is low, or you have a large number of servers and wish to minimize the amount of polling.

Do the following:

- ☐ Edit the plugin's `jvm.properties` file.
- ☐ Set the following property:

  `com.ibm.ws390.wc.webappupdateinterval=`*<interval in minutes>*

  Note: This value should be set to something less than the `serverCheckInterval`.

- ☐ Stop and restart the webserver to pick up this change.

(This page intentionally left blank)

Index

(This page intentionally left blank)

End of Document