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

A Problem Determination Methodology for the Web Environment

This document can be found on the web at:
`www.ibm.com/support/techdocs`
Search for document number **`WP100295`** under the category of "White Papers"

Version Date: July 18, 2002

IBM Washington Systems Center

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

(This page intentionally left blank)

Table of Contents

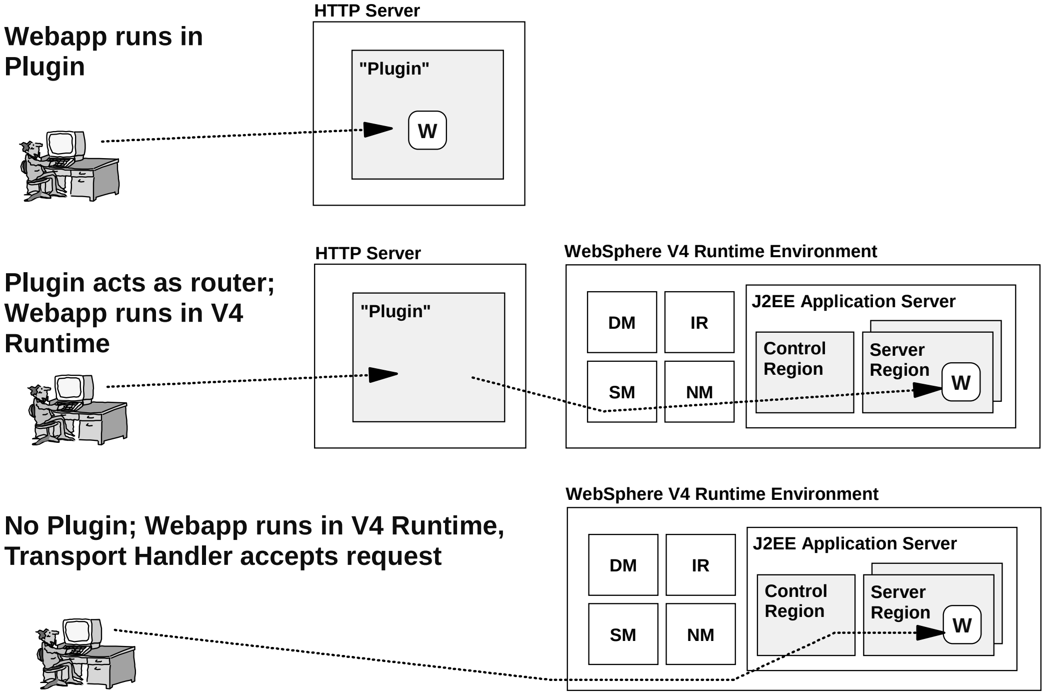
Document Change History

Date	***Nature of Change or Update***
July 17, 2002	Original Document
July 18, 2002	Discovered that Transport Handler (Case #3) actually does a two-step check on virtual host and context root. Two different error messages can be generated. That detail was added.

Introduction

This paper is a result of a presentation I created over a year ago for the "Gen 2" class (which covered WebSphere V3.5 Standard Edition). The purpose of the original presentation was to provide the Gen 2 students a basic methodology of problem determination in that "Plugin-only" environment. I was often asked how it was that I could figure out a lab team's problems so quickly. That question had two answers: first, I had seen the problems many times before in previous classes, and I had a sense for what problems were likely to appear; and second, I had formed in my mind a flowchart of the process of getting a URL to a webapp, and from that flowchart I was able to do debugging based on the symptoms I saw in the class labs.

This paper grew out of that original WebSphere V3.5 presentation. With WebSphere V4.1 there are a few other ways of accessing and running web applications. Those new methods were folded into the presentation. This paper outlines three "cases," or scenarios:


Webapp runs in Plugin
HTTP Server
"Plugin"
W
Plugin acts as router; Webapp runs in V4 Runtime
HTTP Server
"Plugin"
WebSphere V4 Runtime Environment
DM
IR
SM
NM
J2EE Application Server
Control Region
Server Region
W
No Plugin; Webapp runs in V4 Runtime, Transport Handler accepts request
WebSphere V4 Runtime Environment
DM
IR
SM
NM
J2EE Application Server
Control Region
Server Region
W

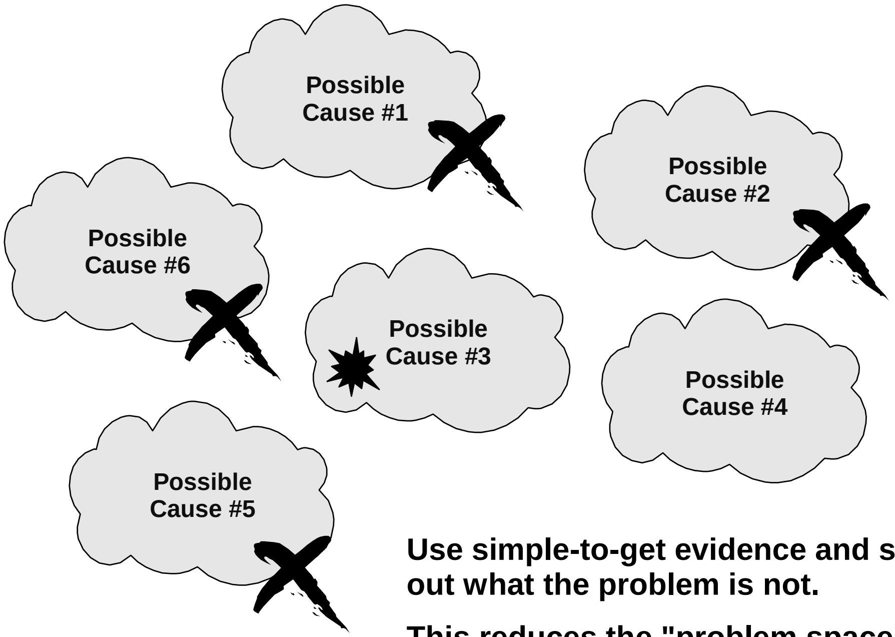
The starting point for each case is provided here:

Note: There's a good deal of common stuff between the three cases, and rather than duplicate the same information over and over, the first telling of it is simply referenced when the time comes to tell the same story. For example, the process of getting the URL into the Plugin is exactly the same for Case 1 as it is for Case 2. Therefore, this paper is best approached as a presentation and read through from front to back.

This paper is not the exclusive "every problem known is covered here" document. (Although I would like it to be that, I haven't figured a way to capture and document every known problem.) But it does cover most of the issues related to getting the URL into the execution environment and resolved to a particular class file. And that alone can save you a good deal of time by allowing you to quickly eliminate areas of the system and focus on where the problem *really* lies.

Initial Problem Determination Strategy

Initial Problem Determination Strategy



Use simple-to-get evidence and symptoms to figure out what the problem is not.

This reduces the "problem space" and makes problem determination easier.

Avoid going into traces right away. There are some simple things you can do that'll tell you quite a bit about what's going wrong.

▪

Problem determination of anything first involves using the external symptoms available, and from that eliminating as many possible causes of the problem as you can. By reducing the "problem space" early in the process, you avoid spending time needlessly in areas where, based on the symptoms you have, you can know that the problem isn't there.

Example: You call up your auto mechanic and say that your car won't start. The first question they are likely to ask in return is, "When you turn the key, does the engine turn over but never starts, or does the nothing at all happen?" What the mechanic is doing is limiting the possible causes of the problem. An engine won't start for a number of reasons: no gas to the engine, no spark to the spark plugs, bad starter motor, low battery, failed electronic component ... and perhaps others.

Now if the symptom is that the engine doesn't turn over *at all*, the mechanic will eliminate from consideration problems associated with lack of gasoline to the engine (no gas in the tank, bad fuel pump, fuel filter clogged). Why? Because if the engine won't turn over at all, the presence or absence of gasoline doesn't make any difference. The mechanic will then focus on problems related to getting that starter motor to turn (faulty ignition switch, dead battery, bad starter motor).

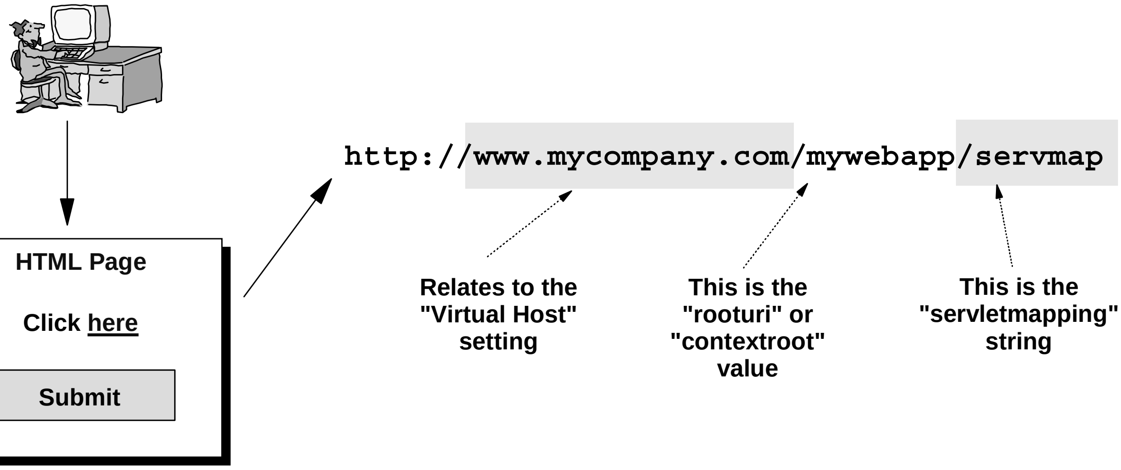
But even before the mechanic asks you the first question, he or she will automatically eliminate from consideration a whole bunch of other problems a car might have, simply based on the symptom of "the car won't start." For example, the car's suspension system has nothing to do with the starting of the car, so all the problems associated with shocks and struts and stablizers and tie rods and such won't even come into play.

Debugging the WebSphere V4 web environment is the same way. There are ways to pare back the problem space based on the error symptoms you see. There are simple tests you can perform to limit the problem space further. What you don't want to do is go neck-deep into the traces right away.

They require system resources to run, and are sometimes difficult to follow unless you're trained to read those things.

Finally, this debugging process depends on you knowing at least somewhat how the gears and wheels of WebSphere turn under the covers. If WebSphere was a complete "black box," following a debugging methodology would be very difficult. (The auto mechanic would have a very difficult time figuring out what's wrong with a car if how a car operated was a mystery.) So this presentation will provide some understanding of how WebSphere's web environment works, and from that we'll look at how you can systematically debug problems.

Understand the Request URL



HTML Page

Click here

Submit

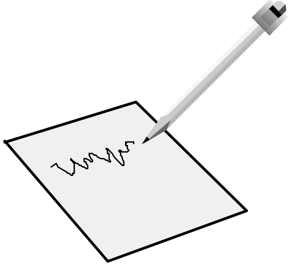
http://www.mycompany.com/mywebapp/servmap

Relates to the "Virtual Host" setting

This is the "rooturi" or "contextroot" value

This is the "servletmapping" string

Never assume you know what the request structure is ... verify it and write it down.

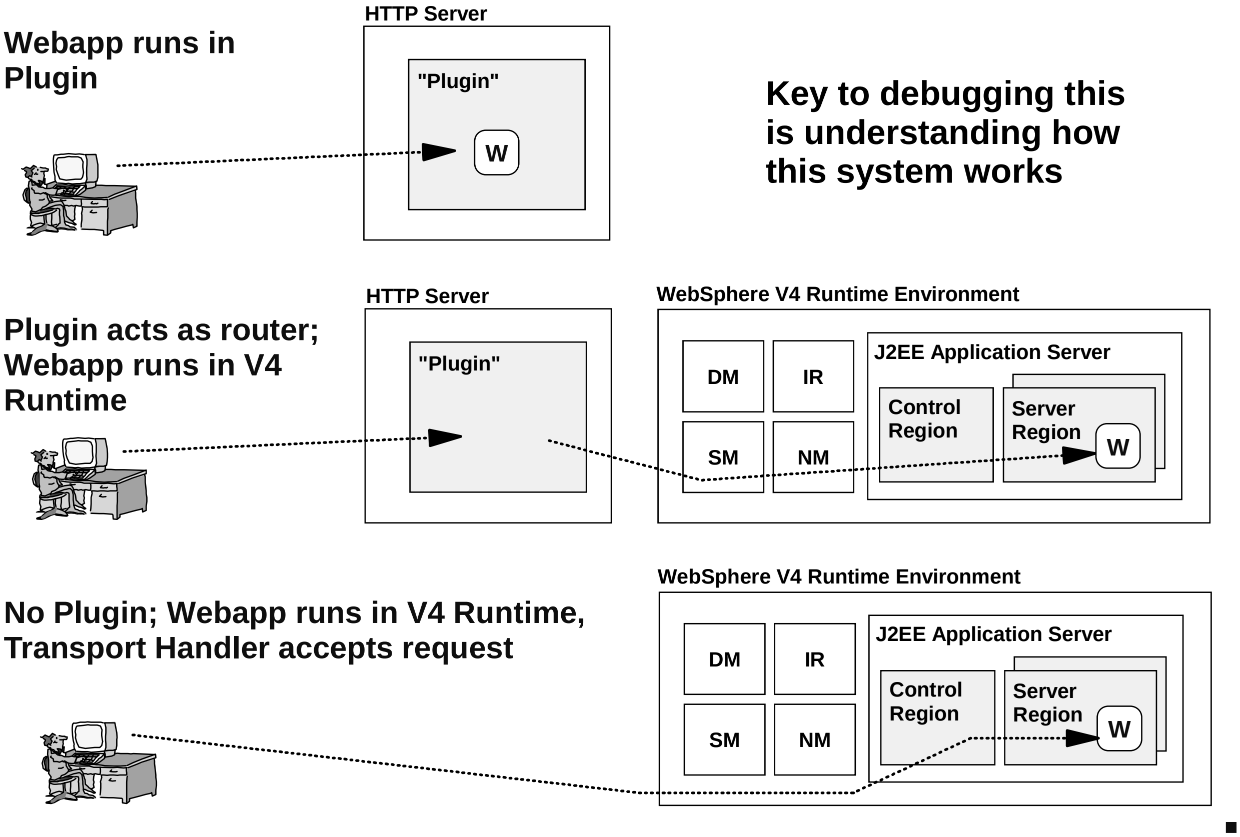


The very first step in the debugging process is to understand the structure of the URL that came into WebSphere. I can't stress this enough: *everything WebSphere does is a reaction to the request it receives*. If you're not sure of what the URL looks like that caused the problem, you'll have a really hard time narrowing the problem space. *Find out what the URL is and write it down*.

Note: It always surprised me how often this common-sense thing was overlooked in the Gen 2 class. When a student would call me over to help with a problem, my first question was usually, "What's the URL you sent in?" And to that question I would often receive a shy, "I'm not sure."

Three Ways to Access Webapps

Three Ways to Access Webapps



Webapp runs in Plugin

Key to debugging this is understanding how this system works

Plugin acts as router; Webapp runs in V4 Runtime

No Plugin; Webapp runs in V4 Runtime, Transport Handler accepts request

WebSphere V4 for zOS provides two different places in which a web application can run, and two different ways an HTTP request can be pulled off the network. What that boils down to is the picture above, which illustrates the three different scenarios that will be covered in this presentation.

They are:

- **Web application runs in the HTTP Server Plugin**

  This is just like it was with the WebSphere V3.5 Standard Edition product. The WebSphere V4 "Plugin" is essentially WebSphere V3.5 with some additional function on top to allow the Plugin to talk to the WebSphere V4 runtime.

- **Web application runs in the J2EE application server, and is accessed through the plugin**

  The Plugin acts like a router, simply receiving requests and forwarding them back to the approriate J2EE application server.

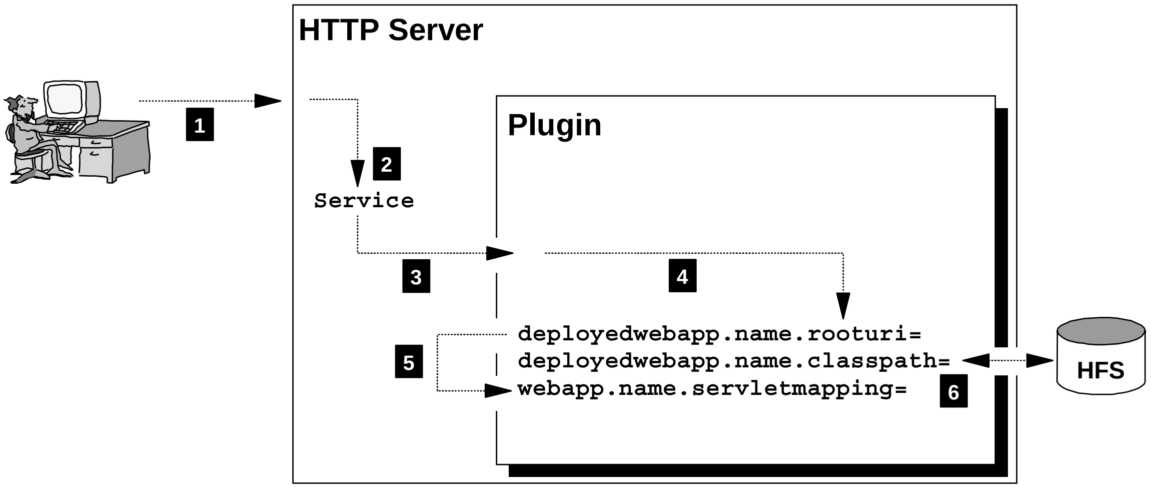
- **Web application runs in the J2EE application server, and is accessed through the Transport Handler**

  The Plugin drops out of the picture, and HTTP requests are pulled off the network by the Transport Handler and fed directly back to the J2EE application server where the webapp runs.

As mentioned earlier, the key to approaching problem determination is to have at least some understanding of how the system works. We'll cover each of these scenarios and provide a peek at what WebSphere does to get the reqeust to the web application, and then relate that to symptoms and problem determination tests you can run to isolate the problem.

Case 1: Webapp Runs in Plugin

Case 1: Webapp Runs in Plugin


HTTP Server
1
2
Service
Plugin
3
4
deployedwebapp.name.rooturi=
deployedwebapp.name.classpath=
webapp.name.servletmapping=
5
6
HFS

For things to work properly ...

1. **Request must get to the webserver**
2. **Request must map to a Service statement**
3. **Request must successfully pass to WebSphere Plugin**
4. **Request must match rooturi and virtual host definition**
5. **Request must match servletmapping definition**
6. **Class file specified must exist, its directory named on classpath, and permission allow it to be read**

We turn first to the scenario where the web application is running in the Plugin. For this to work properly the following things must happen:

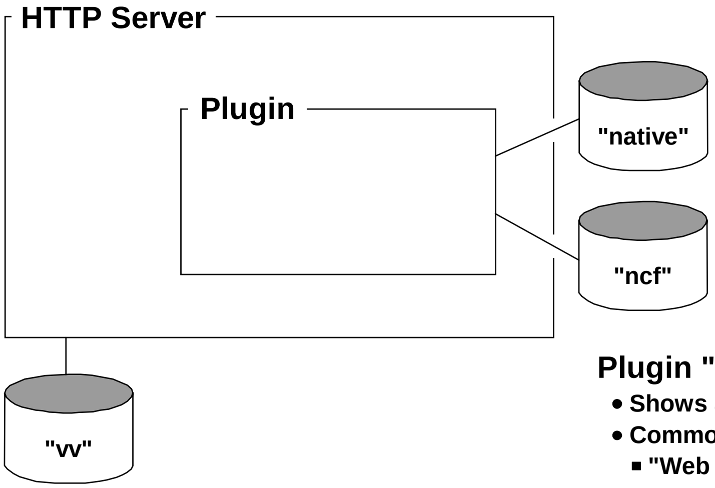
1. The request from the browser must be able to get to the HTTP Server. If there's a network problem between the browser and the web server that prevents that request from every getting to the server, then clearly the webapp will never be run properly.
2. The request must map to a `Service` statement in the webserver's `httpd.conf` file. This is what passes the request over to the Plugin so the webapp can be executed. If you forget to code a `Service` statement for the expected URL, the webserver will treat the URL as if it was a request for a static HTML page. That will probably result in a 404 error.
3. The request must successfully pass to the Plugin. Just because your request maps to a `Service` statement does not mean the request will get over into the Plugin. What happens if the Plugin isn't initialized? The webserver will attempt to pass it, but the attempt will fail. The result is a "500: Service Handler Performed No Function" message. The "Service Handler" (a function of the HTTP Server) tried but could not perform its duty ... there was nothing on the other side to "catch" the request.
4. The request must match "rooturi" and "virtual host" definition in the Plugin's `was.conf` file. Once the request gets into the Plugin, the request must map to a block of definitions in the was.conf file for the servlet to be run. The first step in this mapping process is to have the URL request match a "rooturi / virtual host" combination. If it can't match, you'll get a "Virtual Host or Web Application Not Found" message.

5. The request must match a servletmapping definition. Assuming you got a hit on the "rooturi / virtual host" combination, the Plugin will next go looking for a match with a servletmapping definition. This is what the Plugin uses to narrow the request down to a specific servlet to run. If get past the "rooturi / virtual host" match, but fail to match on a servletmapping string, you'll get a "404: No Target Servlet Configured" message.
6. The class file for the servlet must exist, be found on the `CLASSPATH`, be able to be read by the Plugin, and be a valid, executable Java class file. If you made it through steps 1 through 5, the Plugin will now have a bead on what Java class file to go load and execute. But just because it knows to go load `MyServlet.class`, that doesn't mean `MyServlet.class` will be found, or will have the right permissions, or even by a good class file. Assuming all those things are okay, the servlet will be executed.

As you can see, this is a logical system, and at each step of the system different error conditions will result. Depending on the error symptom you see, you can tell *where* in the process the failure occurred. That doesn't necessarily tell you *why* the problem occurred, but at least it helps you narrow the problem space.

Sources of Trace Information

Sources of Trace Information



Plugin "native" trace

- **Provides information on the plugin, but not Java execution within plugin**
- **Somewhat cryptic**
- **Often has size of 0 bytes**

Plugin "ncf" trace

- **Shows activity inside of plugin**
- **Common things flagged here:**
  - **"Web group not found" - no hit on rooturi**
  - **"Class not found" - classpath problem**
  - **"Unable to load class" - permissions**

Webserver's "vv" trace

- **Provides information on webserver activity**
- **Useful in validating receipt of request**
- **Useful in validating request mapping to WAS**
- **Goes silent after request goes to the Plugin**

Over the next several charts I will make mention of various traces where confirmation of a problem can be had. For the Plugin, three traces come into play:

- **Webserver "vv" trace**

  This tells the story of what the the HTTP Server is doing when it receives a request. For instance, it will tell you whether or not a URL gets mapped to a `Service` statement. It should be noted that once a request does get mapped to a `Service` statement, this trace goes silent. The handling is then in the Plugin, and the Plugin traces take over.

- **Plugin "native" trace**

  The "native" trace is of limited value, particularly when debugging Java execution errors. This trace contains information about how the Plugin is running, but not about programs executing in the Plugin (that's in the "ncf" trace, described next). This *amount* of tracing is set with the `appserver.loglevel=` property in the `was.conf`, and the output is directed with the `appserver.logdirectory=` properly.

- **Plugin "ncf" trace**

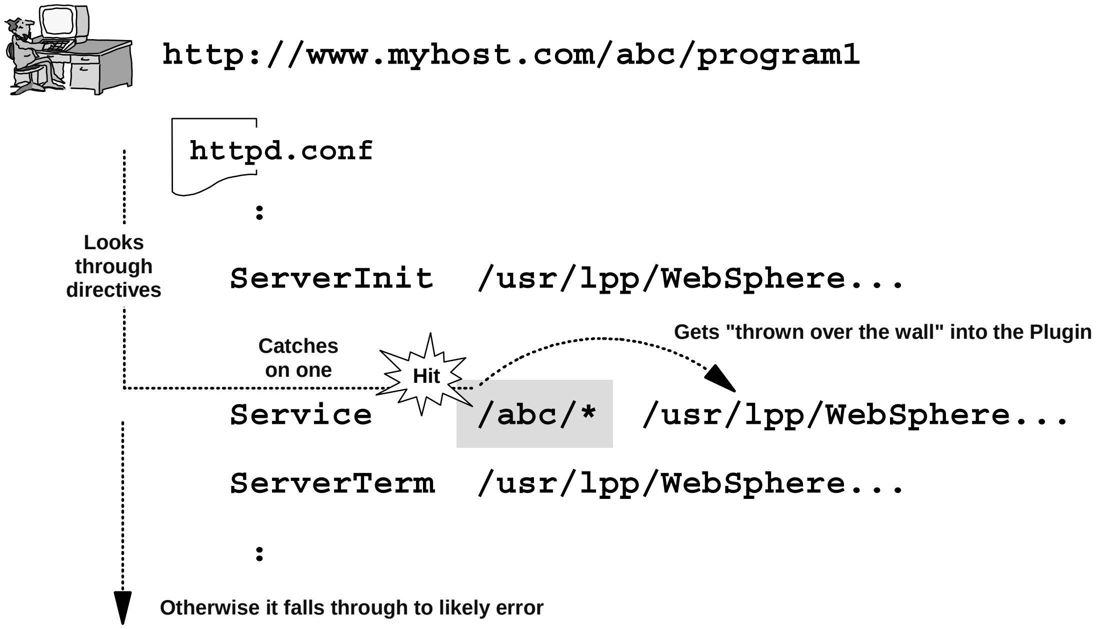
  This trace is where information on the execution of the Java program takes place, and that means this is where you can get details on many of the problems we'll describe in this presentation. Thankfully it's not necessary to go into this trace to do some of this debugging. But once you've narrowed the problem, this trace will prove beneficial.

Note: These two traces (native and ncf) are separate files if the logging is directed to the HFS. But if the logging goes to `SYSOUT`, the two are mixed together.

This presentation is not intended to be a lesson in reading these traces. They are often long, full of information not directly related to any problem, and sometimes difficult to read. They are, however, invaluable when you've isolated the problem down to a particular spot, but need more detail.

How the URL Gets Into Plugin

Focus on #2 and #3



```
http://www.myhost.com/abc/program1

httpd.conf
  :
ServerInit  /usr/lpp/WebSphere...
Service     /abc/*  /usr/lpp/WebSphere...
ServerTerm  /usr/lpp/WebSphere...
  :
```

Looks through directives
Catches on one
Hit
Gets "thrown over the wall" into the Plugin
Otherwise it falls through to likely error

Two key things here:

- Service directive must be coded to "catch" URL and "throw over the wall"
- Plugin must be initialized to receive URL, otherwise URL is thrown but bounces back

Let's now take a closer look at the `Service` processing done by the HTTP Server. This is steps #2 and #3 from the first chart of this section.

The way in which the HTTP Server passes a request over to the Plugin is by invoking the "Service Handler" function. The URL received invokes this function when it matches the mask on a `Service` statement. If a match occurs, like what's illustrated above, the webserver will pass the request to the program specified on the right side of the `Service` directive. For the WebSphere V4 for zOS Plugin, that program is:

```
 /usr/lpp/WebSphere/WebServerPlugin/bin/was400plugin.so:service_exit
```

If the URL doesn't match any `Service` statements, the webserver will simply fall through and evaluate the URL against other directives in the `httpd.conf` file. The chances are good that what will result is an error when the webserver fails to match to *any* directive, or when it matches to the `Pass /*` directive.

The key points here are this:

- If you don't have a `Service` directive coded to catch the URL, the URL will never make it over to the Plugin. This is a common mistake. But the error page generated by this clearly tells the story of what went wrong, provided you understand the process illustrated above. If you get the classic "404" error when you though you were invoking a servlet, the chances are good you never caught on a Service statement.
- Just because a URL matches a `Service`, it does not mean the request will get over to the Plugin. The webserver might not have anything on the other side to pass the request. If the Plugin isn't

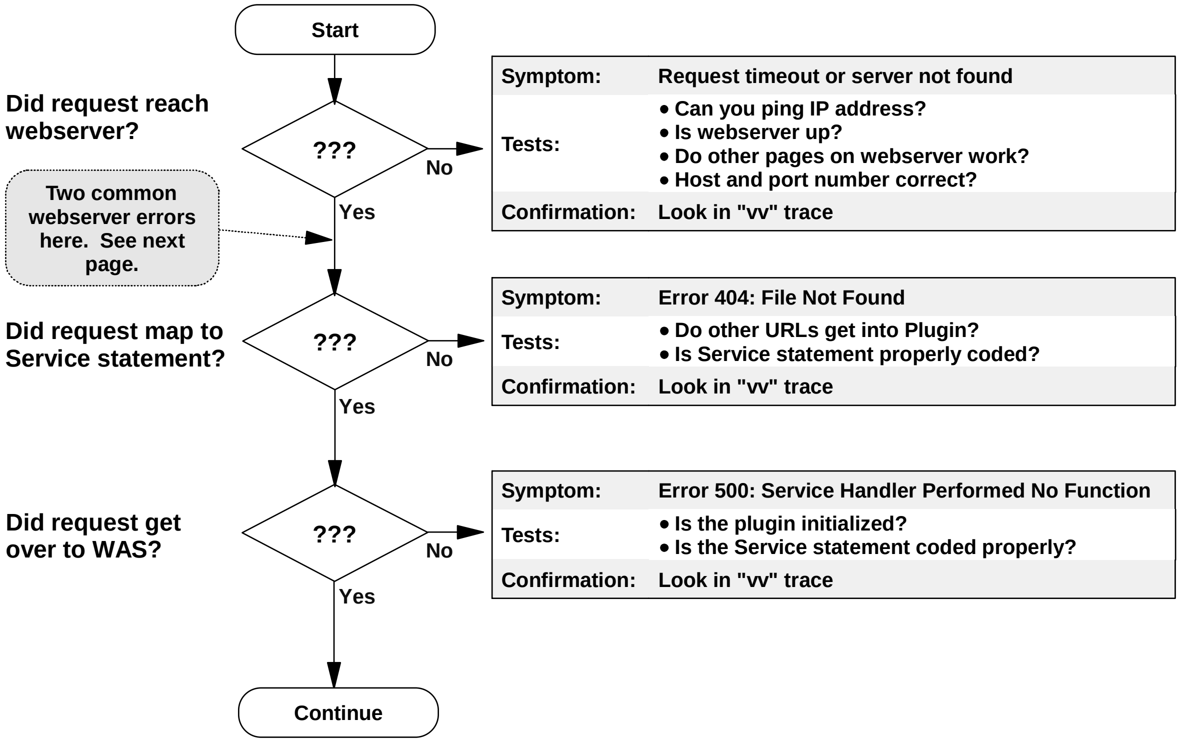
initialized, the attempt to "throw it over the wall" into the Plugin will fail. There are a lot of things that can keep the Plugin from initializing (we'll show you those in a bit). Also, if you mistype the directory or program name on the right side of the `Service` statement, the webserver will try to invoke the program specified, and then fail (a typo error will point to a directory that doesn't exist; the webserver has no choice but to fail).

This process of passing the request from the HTTP Server over to the Plugin is a key piece of the puzzle anytime the Plugin is part of the equation. If it doesn't get "over the wall" into the Plugin, *nothing* else will work.

Thankfully, it's fairly easy to tell if the request got over the wall based just on the browser error message.

Flowchart: Did Request Get Into the Plugin?

Did Request Get Into Plugin?



Did request reach webserver?

Two common webserver errors here. See next page.

Symptom:	Request timeout or server not found
Tests:	• Can you ping IP address? • Is webserver up? • Do other pages on webserver work? • Host and port number correct?
Confirmation:	Look in "vv" trace

Did request map to Service statement?

Symptom:	Error 404: File Not Found
Tests:	• Do other URLs get into Plugin? • Is Service statement properly coded?
Confirmation:	Look in "vv" trace

Did request get over to WAS?

Symptom:	Error 500: Service Handler Performed No Function
Tests:	• Is the plugin initialized? • Is the Service statement coded properly?
Confirmation:	Look in "vv" trace

Let's now introduce the logic flowchart that illustrates the process of getting to the webserver and then into the Plugin itself.

- The very first question is whether or not the request even got to the webserver. The typical symptom is some kind of request timeout message or a "server not found" error message. Lots of things can go wrong between a user's browser and the webserver itself. How can you tell? The easiest test is to ping the webserver's host name. (You could ping the IP address itself, but it's better to ping the host name so the DNS system is validated). But all that does is test connectivity between the user's workstation and the adapter of the webserver's system.

  So the next test is to make sure the webserver is up. You could do that by going to the system console and listing the active MVS tasks, but it's possible the webserver is showing "active" when it is in fact locked and not accepting traffic. So another test is to have the user point their browser at a web page *you know to be on the server* and accessible. If they can receive that page, then you know the webserver is up. So if the original URL failed, then perhaps there's something wrong with that URL, such as a mis-typed host name, or a bad port number.

  Note: Beware of browser cache when doing this simple connectivity test. Force a re-load of the page to insure you get the copy from the server and not from cache on your browser workstation.

- Once you've confirmed that the user can in fact reach the webserver, the next question is whether or not the request mapped to a `Service` statement. Failure to map to a `Service` statement will most often result in an "Error 404: File Not Found" error message.

  Note: Unless one of the two common HTTP Server errors occur. Those are described on the next chart.

  What the webserver is telling you is that no `Service` was evaluated, and the request fell through to a `Pass` directive, which tried to resolve the request to a static file ... and failed to find that file. So if you get an HTTP Server 404 error when you expected to invoke a servlet, you know the request never made it over the wall.

  The chances are good the `Service` statement is simply not coded at all (a common mistake), or the mask on the `Service` statement didn't "catch" the URL. The best way to check this quickly is to visually inspect the `Service` statement against the URL received. With practice you'll get really good at spotting these errors. For 100% validation, you can go into the "vv" trace and see what the webserver is doing with the request it received. If you see it evaluated against a `Pass` directive, you know no `Service` statement was invoked.

  Note: It's important to note that a "404" error means no `Service` was invoked. If your URL catches on a `Service` statement, but something else fails, you'll get something other than a 404 error message.

  If you have a test servlet deployed that you can have the user point to, that'll help verify if servlets in general are accessible. That will help isolate the problem back to a problem with their servlet request.

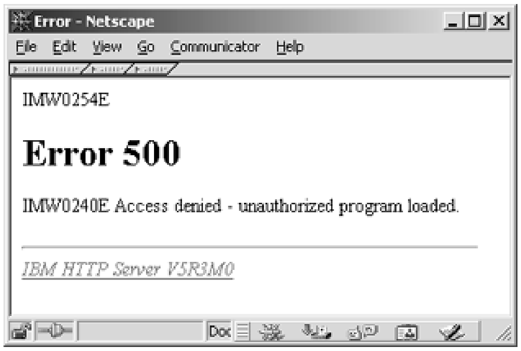
- If you get the error message "500: Service Handler Performed No Function," that means the request go to the webserver, caught on a `Service` statement, but for some reason couldn't get passed over to the Plugin. Here two questions come to mind: "Is the Plugin initialized?" and "Is the `Service` statement coded properly?"

  You can check whether the Plugin is initialized by looking in the webserver's trace for the "smiley face" message, or by pointing your browser at a test servlet you have deployed. If that test servlet works, you know the Plugin is up and working. If you've confirmed the Plugin is initialized, then the problem is probably that the `Service` statement for the failing URL is miscoded. The typical problem here is a typo error in the directory pointer to the `was400plugin.so` program. One small typo in that and the webserver will fail trying to pass the request.

If you the error symptom you get indicates that the request got into the Plugin (we'll look at those in a bit), you don't need to worry about all the stuff that comes prior to the Plugin. That's one of the first "rough-cut" steps: did the request get into the plugin?

Two Common HTTP Server Problems: Program Control and Surrogate Setup

Common HTTP Server Problems



IMW0254E

Error 500

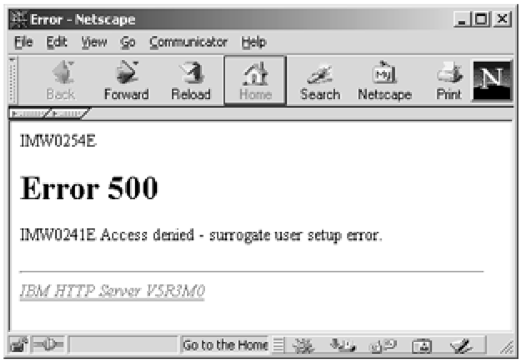
IMW0240E Access denied - unauthorized program loaded.

IBM HTTP Server V5R3M0

500 - "Unauthorized Program Loaded"

HTTP Server loaded module that wasn't program controlled:

```
SYS1.LINKLIB
hlq.SCEERUN
hlq.SCLBDLL
TCPIP.SEZALINK
hlq.DSNLOAD and hlq.DSNEXIT (DB2)
hlq.CSSLIB (JDBC)
All Plugin DLLs
All Java DLLs
```


Error - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print
Go to the Home

IMW0254E

Error 500

IMW0241E Access denied - surrogate user setup error.

IBM HTTP Server V5R3M0

500 - "Surrogate User Setup Error"

HTTP Server attempted to switch task to userid on which is did <u>not</u> have surrogate authority

```
httpd.conf
  :
Userid  PUBLIC
  :
```

■

There are two very common errors that can occur when a URL hits your webserver, but before it gets into the Plugin itself. They are:

- **Error 500: Access Denied - Unauthorized Program Loaded**

  This error occurs whenever the HTTP Server attempts to load a module into its address space that is not program controlled. The chart shows the typical MVS data sets that need to be program controlled to be used by the HTTP Server. Those are ones that would have been hit long ago when the webserver was first being installed.

  The last two -- the highlighted ones -- are the ones that will crop up if you're installing the Plugin for the first time. Because the Plugin runs inside the webserver's address space, it will load the Plugin's DLLs into the address space. Therefore, all the DLLs in the Plugin's `/bin` directory need to be program controlled. Similiarly, since the Plugin will load a Java Virtual Machine (JVM), the Java `/bin` directory files also need to be program controlled.

- **Error 500: Access Denied - Surrogate User Setup Error**

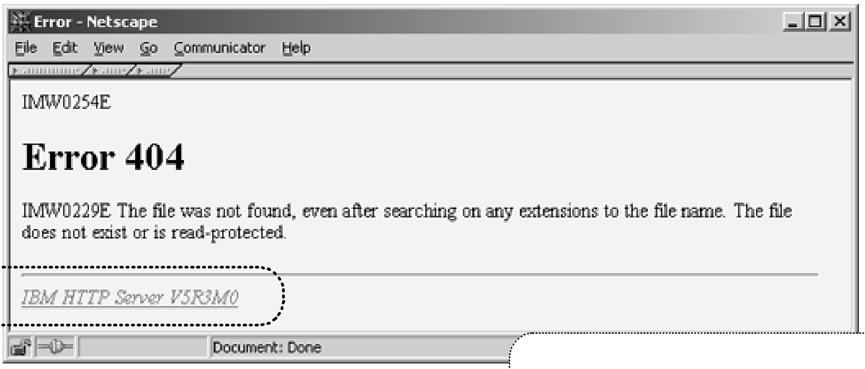
  This error occurs when the HTTP Server attempts to switch the task to a userid over which it does not have RACF surrogate authority. The most common reason for this is the use of an ID on the `Userid` directive in the `httpd.conf` file, and the webserver's ID (the one under which the started task is running) doesn't have surrogate authority.

  In truth, this problem will probably have been discovered when the HTTP Server was first being installed. But if for any reason the `Userid` directive value is changed, and the RACF work to provide surrogate authority wasn't done, you'll hit this problem.

Two Different 404 Messages: Who Issues the Message is Important

Who Issued the Error Message?

Webserver:



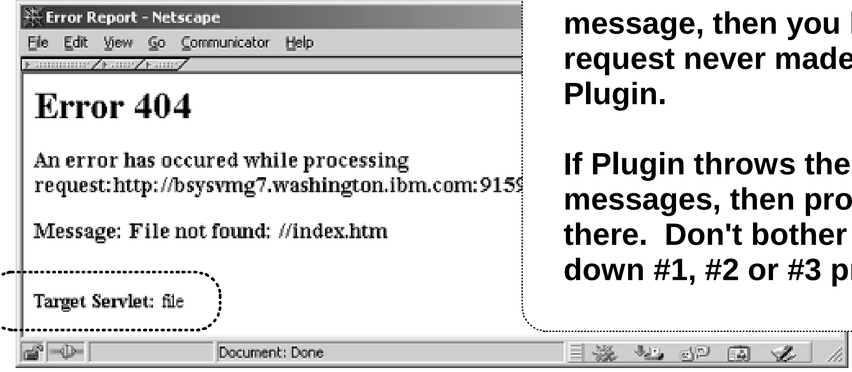
IMW0254E

Error 404

IMW0229E The file was not found, even after searching on any extensions to the file name. The file does not exist or is read-protected.

IBM HTTP Server V5R3M0

Plugin



Error 404

An error has occured while processing request:http://bsysvmg7.washington.ibm.com:915

Message: File not found: //index.htm

Target Servlet: file

If webserver issues error message, then you know the request never made it to the Plugin.

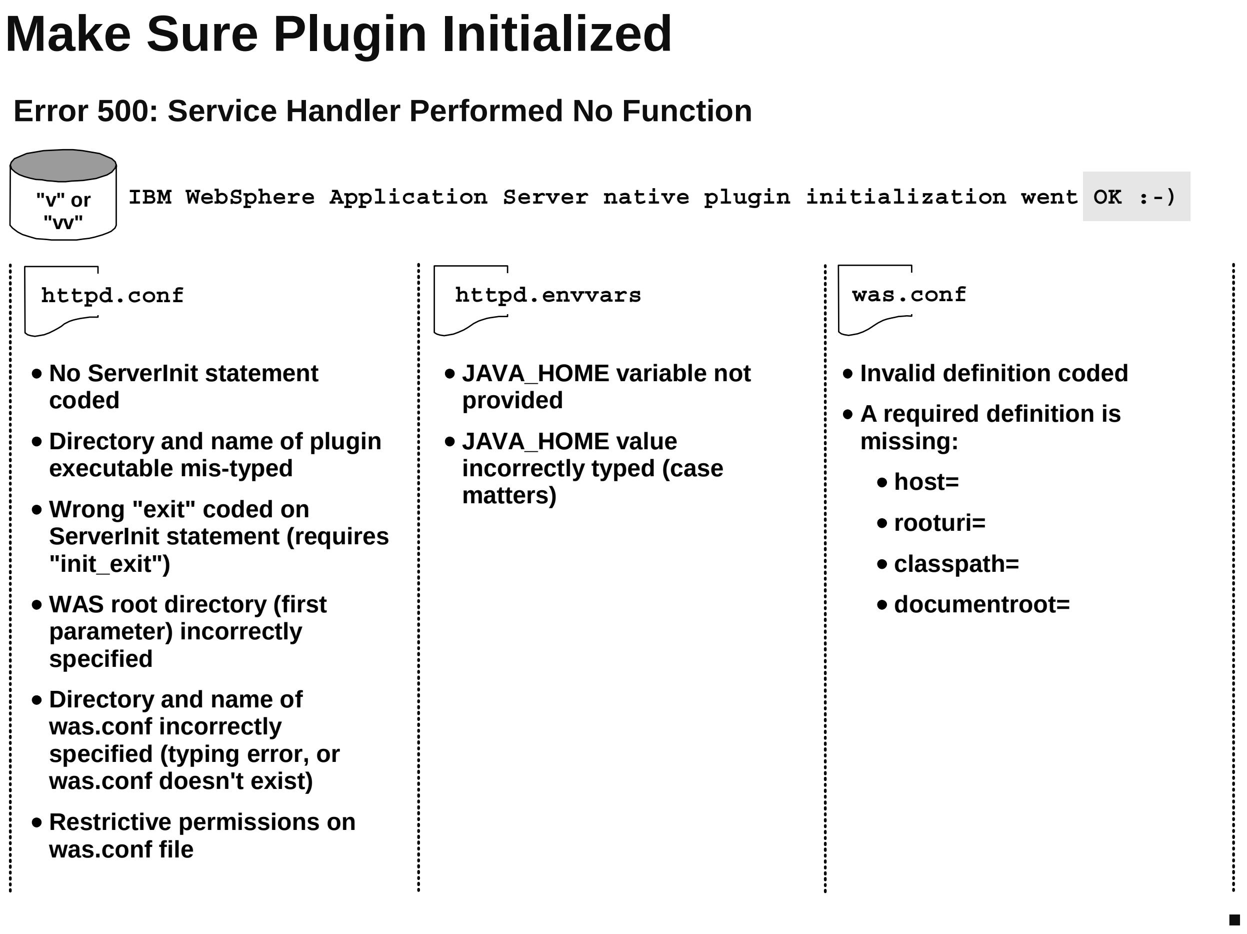
If Plugin throws the messages, then problem is there. Don't bother hunting down #1, #2 or #3 problems.

For most people, the common "404" message signals that the webserver couldn't find the page requested. But it turns out the Webserver and the Plugin *both* issue 404 error messages. If you're not using custom error pages, the sample above shows what the two versions look like. The HTTP Server message will say it's from the "IBM HTTP Server," and the Plugin message will indicate that the target servlet was "file." URLs that get thrown over to the Plugin but end up not matching any definitions in the `was.conf` file will be considered by the Plugin to be, by default, a request to issue a file. The Plugin has a built-in servlet called the "SimpleFileServlet," and its role is to simply serve files to the user. If that function gets invoked because the Plugin couldn't resolve the request to anything else (no definitions in `was.conf`), and that function couldn't find the file implied, then this 404 comes out.

The reason I bring this up is for this reason: you can use this to quickly determine in what area the problem lies:

- **HTTP Server 404** -- you know the request never got "over the wall" into the Plugin. That means the problem has something to do with a Service statement, since that's the mechanism the webserver uses to throw requests over to the Plugin. As we saw on the previous chart, if a URL matches a Service statement but the Plugin isn't there to catch it, you get a 500 error message. That means that a webserver 404 message means no Service statement was matched at all. Either you didn't code one, or the pattern mask on the directive didn't match up with the URL received.
- **Plugin 404** -- you know the request got over the wall. Therefore, don't bother with issues related to the HTTP Server ... it's done its job and thrown the request over to the plugin. Your problem lies in the coding of the `was.conf`.

Reasons Why the Plugin May Not Initialize



Make Sure Plugin Initialized

Error 500: Service Handler Performed No Function

"v" or "vv"

```
IBM WebSphere Application Server native plugin initialization went OK :-)
```

```
httpd.conf
```

- **No ServerInit statement coded**
- **Directory and name of plugin executable mis-typed**
- **Wrong "exit" coded on ServerInit statement (requires "init_exit")**
- **WAS root directory (first parameter) incorrectly specified**
- **Directory and name of was.conf incorrectly specified (typing error, or was.conf doesn't exist)**
- **Restrictive permissions on was.conf file**

```
httpd.envvars
```

- **JAVA_HOME variable not provided**
- **JAVA_HOME value incorrectly typed (case matters)**

```
was.conf
```

- **Invalid definition coded**
- **A required definition is missing:**
  - **host=**
  - **rooturi=**
  - **classpath=**
  - **documentroot=**

The error message "Error 500: Service Handler Performed No Function" comes up whenever a `Service` statement is met, but the request can't be passed to the Plugin. There are two primary causes of this failure:

1. The Plugin itself isn't initialized (as indicated by "smiley face" in "vv" trace)
2. The Service statement has a typo error and is pointing off to an invalid HFS location or file

Note: Read the text of the 500 error message carefully ... there are other 500 error messages related to RACF surrogate setup errors, or problems associated with program control not set for some modules of the Plugin code. What's being discussed here is the "Service Handler Performed No Function" message.

The HTTP Server will happily invoke the "Service Handler" (The `Service` statement) even if the right-hand side of the directive is pointing off into never-never land. Or, the `Service` statement could be coded perfectly, but the Plugin didn't come up for some other reason.

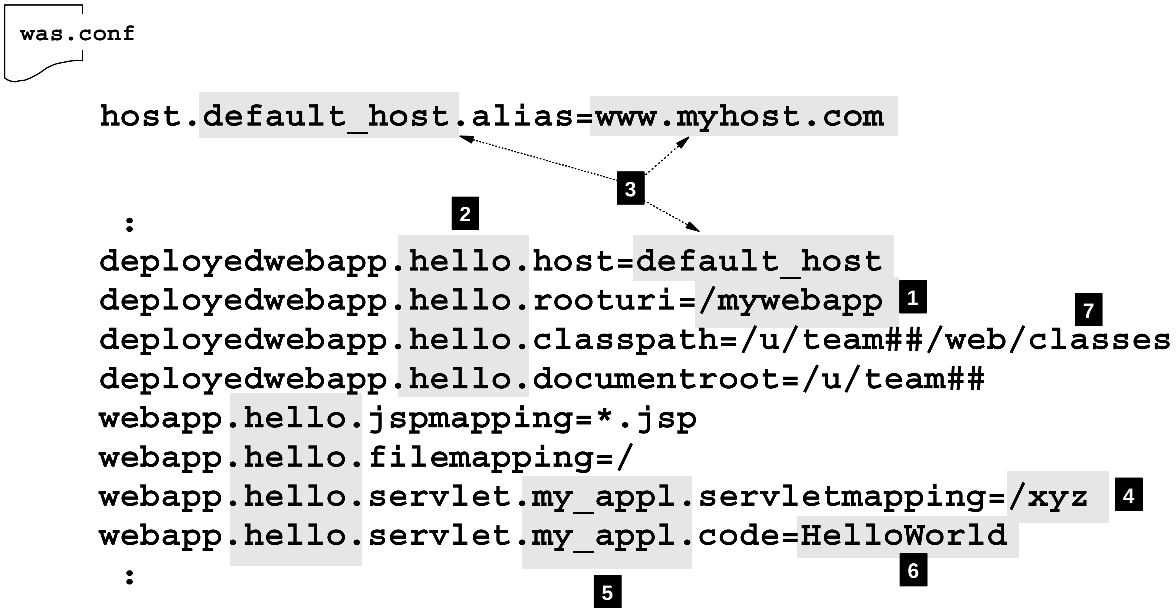
So if you get a 500 error message, the first thing to do is check for the smiley face. If you have no smiley face, you know the Plugin isn't initialized. Then you can start investigating the cause of that, and the chart above summarizes the things that can cause the Plugin to fail to initialize. If the Plugin did initialize, and you still got a 500 error, that means the Service statement pointer to the Plugin has a typo in it somewhere.

Unfortunately, the traces aren't terrible helpful in debugging these problems. The best solution is to visually inspect your definitions and look for mistakes.

Connecting all the Pieces of a Webapp Configuration in was.conf

To Invoke a Servlet ...

http://www.myhost.com/mywebapp/xyz



```
was.conf

host.default_host.alias=www.myhost.com

  :
deployedwebapp.hello.host=default_host
deployedwebapp.hello.rooturi=/mywebapp
deployedwebapp.hello.classpath=/u/team##/web/classes
deployedwebapp.hello.documentroot=/u/team##
webapp.hello.jspmapping=*.jsp
webapp.hello.filemapping=/
webapp.hello.servlet.my_appl.servletmapping=/xyz
webapp.hello.servlet.my_appl.code=HelloWorld
  :
```

Use this process when you visually inspect `was.conf` looking for errors

By now you've determined that you've made it into the Plugin. Now comes the process of identifying and invoking the servlet in question. Much can go wrong here as well. Here is the process that takes place:

1. The entire URL is passed over to the Plugin, including the host portion. The Plugin then looks for a `rooturi=` definition that matches the URL received.
2. The Plugin next goes to the "application name" portion of the statement (which is a string of your choosing -- `hello` in this example), and then "bundles up" all the other statements in the `was.conf` with the same "application name" string. This is how the Plugin relates one definition in the `was.conf` to another, so it's important that the value used for all the definitions for an application block *be identical* (including case).

> ***Suggestion:*** Keep all the statements for a given application grouped together in a contiguous block in the `was.conf` file. The sample `was.conf` file provided with the Plugin splits the `deployedwebapp` from the `webapp`, which I don't think is a good idea. Having all the statements in a grouped block (like what's shown above) allows you to visually inspect the definitions easier. If you had the definitions for an application scattered all over the `was.conf`, you would have to scroll up and down to visually inspect. Keep them together.

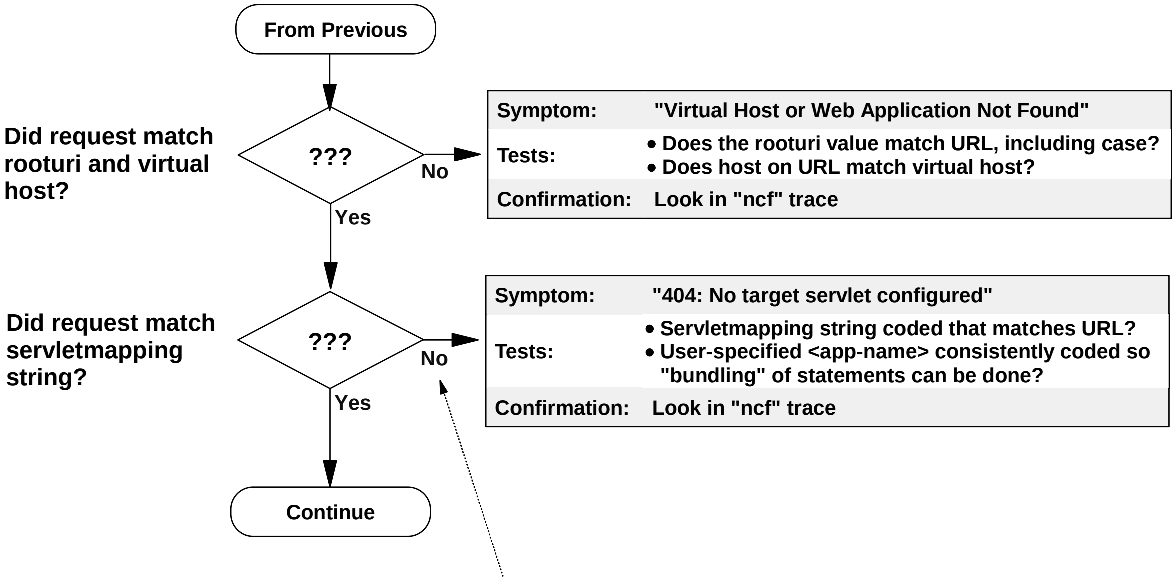
3. Next comes the check for the virtual host. With the definitions "bundled up," the Plugin can now locate the `host=` statement, which has the pointer to the `alias=` statement that names the virtual host for this application. In this example, **`default_host`** points to the `host.`**`default_host`**`.alias=` statement with a value of `www.myhost.com`. The Plugin will see if the URL received has that same host value. In this example it does.

4. Assuming the virtual host matches, the Plugin next goes in search of a servletmapping string that maps to the URL received. The Plugin will takes the servletmapping string it finds in the "bundle" of definitions and concatenate it on the end of the rooturi string and see if it matches. In this example `/xyz` is concatenated on the end of `/mywebapp` to form `/mywebapp/xyz`. That matches character-for-character with the URL received. With a match, the Plugin can now go in search of the actual servlet to invoke.
5. The "servlet name" is another string that you get to make up, and it's purpose is to allow the Plugin to do a "sub-bundle" of definitions within the bigger bundle performed in step 2. This sub-bundle allows the Plugin to link the servletmapping string (`/xyz` in this case) with the `code=` definition that names the actual class file to invoke.
6. In this example, the class file is `Helloworld.class` (the `.class` extension is never specified).
7. The Plugin now goes to the `classpath=` definition and looks for the classfile in the directories specified.

As you can imagine, there are lots of places where things can go wrong in this system. This chart was presented because it shows you a systematic method you can use to inspect your `was.conf` for errors. Many of the problems that you may encounter will not be flagged in the "ncf" trace with anything so obvious as "Hey! You forgot to code the statement properly!" So frequently you'll simply have to go into the `was.conf` and look for the problem.

Flowchart: Resolving URL to Plugin Action

Resolving Request to Plugin Action



Did request match rooturi and virtual host?

Symptom:	"Virtual Host or Web Application Not Found"
Tests:	• Does the rooturi value match URL, including case? • Does host on URL match virtual host?
Confirmation:	Look in "ncf" trace

Did request match servletmapping string?

Symptom:	"404: No target servlet configured"
Tests:	• Servletmapping string coded that matches URL? • User-specified <app-name> consistently coded so "bundling" of statements can be done?
Confirmation:	Look in "ncf" trace

Error message may be different if you have "jspmapping" or "filemapping" coded ...

So let's walk through the logic of that process and see what kinds of error conditions you can expect.

- The first question is whether the "virtual host + rooturi" portion of the URL matched any definitions. If not, you'll get an error message that says "Virtual Host or Web Application Not found." What this means is the Plugin searched through the entire `was.conf` and found nothing it could match against. The "ncf" trace will simply confirm that it couldn't find anything, but offer no other assistance with what may be the problem. You are left to visually inspect the `was.conf` and determine the reason why the URL didn't match.
- If a hit on "virtual host + rooturi" was successful, the Plugin will now go looking for a servletmapping match. If it fails to find a servletmapping match, you will get a "404: No target servlet configured" error message.

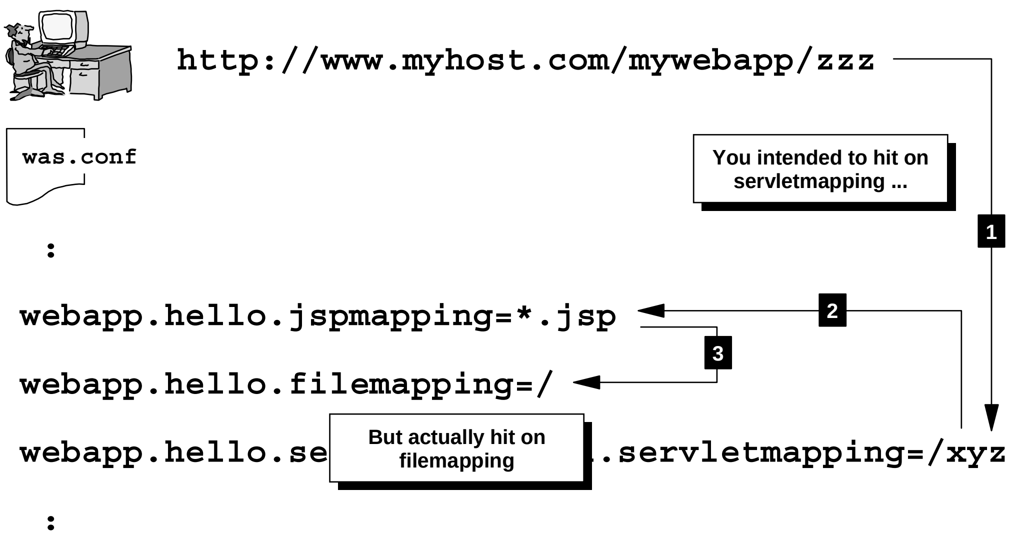
  Note: Or you may get a "404: File Not Found" message. It depends on whether you have a `filemapping=` definition in the `was.conf`. That's covered on the next chart.

  If you get the "404: No target servlet configured" message, you know that you got a hit on the rooturi portion. So you can go into the `was.conf`, visually inspect the block of definitions and see why you didn't hit on the servletmapping portion.

Let's now turn to how the Plugin handles JSP and static file requests.

Order of Process of Servlets, JSPs and Files in Plugin

Order of Processing



```
http://www.myhost.com/mywebapp/zzz
```

was.conf
You intended to hit on servletmapping ...
1
2
3
But actually hit on filemapping

```
  :
webapp.hello.jspmapping=*.jsp
webapp.hello.filemapping=/
webapp.hello.se .servletmapping=/xyz
  :
```

Error Message Received:

- **If only servletmapping coded:** ***Error 404: no target servlet configured***
- **If coded as shown above:** ***Error 404: file not found. Target servlet: file***

This chart illustrates how the Plugin handles a URL when three different "mapping" statements exist in the block of definitions for a web application. The example above shows a URL with a `/zzz` following the rooturi of `/mywebapp`. The Plugin will then try to match against a servletmapping string first. But in this example, the servletmapping is `/xyz`, not `/zzz`, so no match occurs.

If the servletmapping definition was the only "mapping" statement for the application block, the failure to match on it would result in a "404: no target servlet configured." But because the definition block has `jspmapping=` and `filemapping=`, the Plugin will continue searching those:

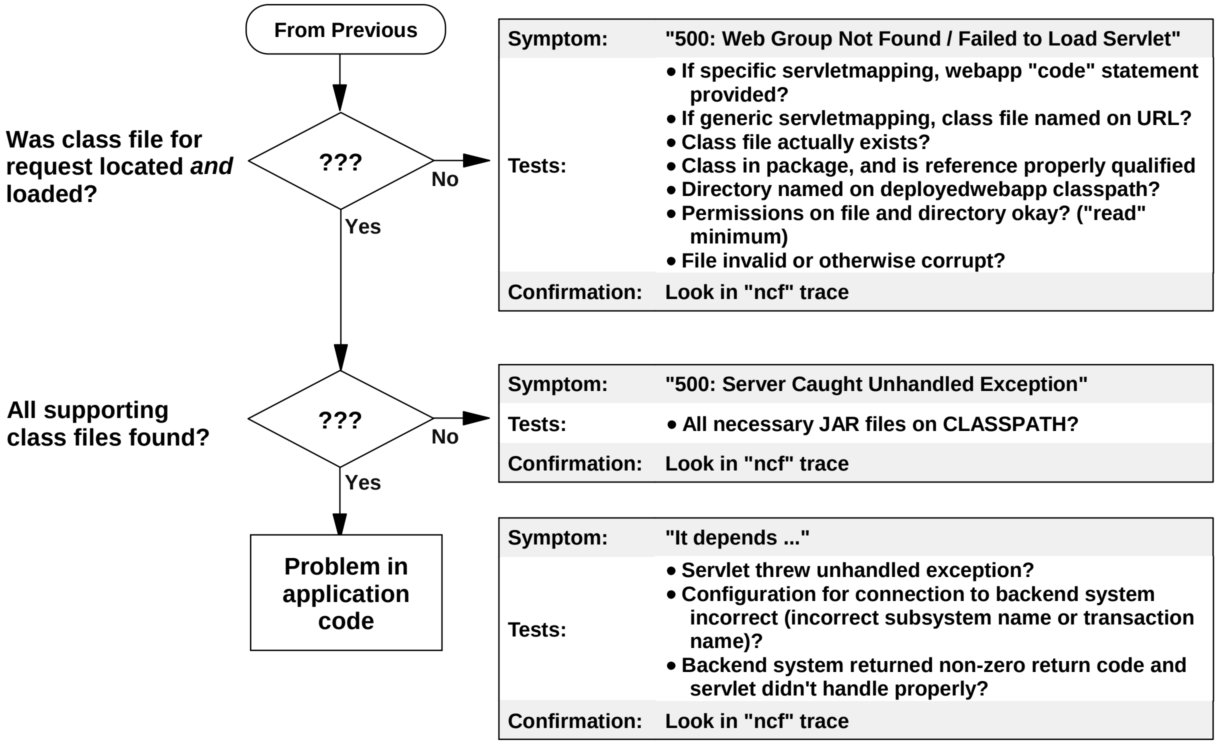
- It'll try to match against the `jspmapping=*.jsp` value, but since the URL doesn't end "jsp," that match won't succeed either.
- It'll then try to match against the `filemapping=/` value. Since "/" is a universal catch-all, it will in fact match against this. The Plugin will then go look for a file called `zzz` in the `documentroot=` directory, but will most likely fail with a "404: file not found. Target servlet: file" error message. This is because the Plugin invoked its "SimpleFileServlet" to issue the file, but when it couldn't find the file it had to issue it's default "file not found" error message.

If you get a "Error 404: file not found. Target servlet: file" error message when you had hoped a servlet would be invoked, then from this message you can determine a few things:

- You're into the Plugin. Don't worry about Service statement issues and the like
- You matched on the "virtual host + rooturi" portion
- You didn't match on a servletmapping, which is what you intended, so focus on that.

Flowchart: Plugin Action Once Servlet Identified

Servlet Identified, Now Load It


From Previous
Was class file for request located *and* loaded?
???
No
Yes
All supporting class files found?
???
No
Yes
Problem in application code

Symptom:	"500: Web Group Not Found / Failed to Load Servlet"
Tests:	• If specific servletmapping, webapp "code" statement provided? • If generic servletmapping, class file named on URL? • Class file actually exists? • Class in package, and is reference properly qualified • Directory named on deployedwebapp classpath? • Permissions on file and directory okay? ("read" minimum) • File invalid or otherwise corrupt?
Confirmation:	Look in "ncf" trace

Symptom:	"500: Server Caught Unhandled Exception"
Tests:	• All necessary JAR files on CLASSPATH?
Confirmation:	Look in "ncf" trace

Symptom:	"It depends ..."
Tests:	• Servlet threw unhandled exception? • Configuration for connection to backend system incorrect (incorrect subsystem name or transaction name)? • Backend system returned non-zero return code and servlet didn't handle properly?
Confirmation:	Look in "ncf" trace

Let's assume you got a hit on the `servletmapping=` string, and the servlet class file has been *identified*. Now the Plugin will go looking for the class file and attempt to load it. At this point there are two *basic* things can go wrong:

- Class file can't be found on any of the `CLASSPATH` entries
- Class file was found on `CLASSPATH`, but can't be loaded

What you will see in this case is a "500: Web Group Not Found / Failed to Load Servlet" message. There's a whole host of things that can cause this error message, as the chart shows. Thankfull, the "ncf" trace will give a fairly good indication of what's going wrong. (For examples of ncf trace listing of some of these problems, see "Trace Examples" on page 45.)

The servlet class file might have loaded perfectly well, but then call for another class file which couldn't be found. This is common when an application requires supporting Java JAR files and you forget to include the JAR file on the CLASSPATH. Depending on how the servlet is coded, it might just throw an unhandled exception, which will result in "500: Server Caught Unhandled Exception."

If you get past all of that, then any problems that result are likely the result of an application problem, or some problem executing against a backend system. The error symptoms you can get vary greatly, but will often be "500: Server Caught Unhandled Exception" if the servlet isn't coded very well to handle error conditions it encounters.There is almost no way to debug these problems without tracing, and perhaps even running debugging at the application layer.

But if you get the "500: Server Caught Unhandled Exception," you can be fairly well assured that your servlet was identified and loaded, but that something else went wrong. So focus from that point forward. Don't worry about rooturi or servletmapping problems.

Some Common Reasons For "Class Not Found"

Some "Class Not Found" Issues

`deployedwebapp.hello.classpath=/u/team##/dir1`	**Directory doesn't exist, or the class file doesn't reside in this directory**
`appserver.classpath=/u/team##/Cics/ctgclient.jar`	**Improper case on directory name**
`appserver.classpath=/u/team##/cics/ctgclient.jar` Permissions: `700`	**Permissions on directory too restrictive**
`appserver.classpath=/u/team##/cics/ctgclient.jar` `TEAM##.CTG.HFS` not mounted	**HFS file system not mounted**
`deployedwebapp.hello.classpath=u/team##/dir1`	**Leading slash missing**

Some other causes:

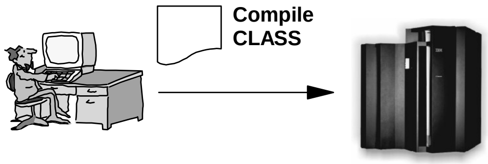
- **Failed to code common support JAR on appserver.classpath**
- **Class in package and fully-qualified name isn't correct**
- **Permissions on class file itself too restrictive**

It would be easy to think that "class not found" problems are simply the case where you forget to put the servlet class in the classpath. But there's a whole range of things that can prevent the Plugin from getting to where the class is located. This chart summarizes a few of them.

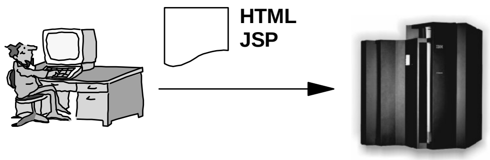
Keep your mind open to other causes of "class not found" other than simply that the file doesn't exist.

Two Common Problems Related to State of Servlet Class File

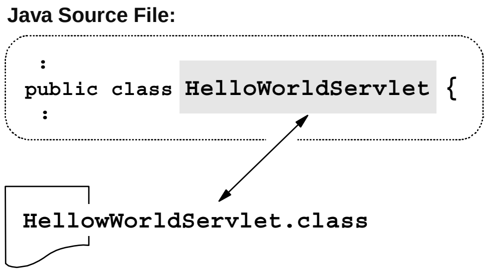
Other Common Problems



FTP compiled class files and JAR files in `binary`. Default FTP to S/390 is `ascii`. Results in file that cannot be read by JVM.



FTP HTML or JSP files `ascii` mode, which will convert to `EBCDIC`. Plugin expects them to be in `ascii` in the HFS.



Java Source File:

```
  :
public class HelloWorldServlet {
  :
```

```
HellowWorldServlet.class
```

Compiled class file must have the same name as name of class in Java source. Case matters. Never rename a class file.

This chart illustrates some other common problems that can occur:

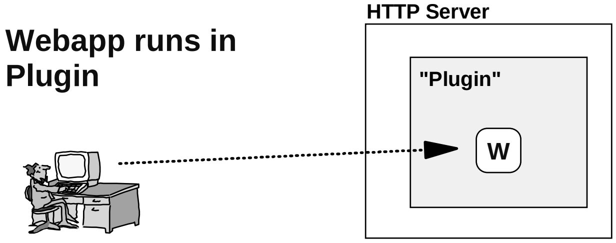
- **Class file transferred from workstation in ASCII format**-- this results in the class file contents being translated into EBCDIC. The JVM will not understand the format of the class file, and will throw a "500: Failed to Load Servlet" error message. *Always FTP class files, JAR files, ZIP files and WAR files in binary mode.*
- **HTML, JSP or Image files transferred from workstation in ASCII format** -- this results in the files being translated into EBCDIC. It's important to understand that the Plugin will assume that HTML pages and JSPs are stored in ASCII in the HFS. The Plugin will *not* do an "on the fly" translation to ASCII as it serves it out. This is exactly opposite of the default behavior of the HTTP Server, which *will* do an EBCDIC-to-ASCII translation on the fly. So if you plan on having the Plugin serve the HTML or JSP pages, then FTP them from the workstation to the host in *binary* mode so that they're stored in the HFS in ASCII codeset.

  GIF/JPG files should always be transferred in binary. But if you accidentally FTP in ASCII, the file will garbled and your images won't show properly.
- **Renaming a class file** -- the name of a class file is directly linked to the the name on the class statement in the Java source code. Renaming a compiled class file after compilation will result in "500: Failed to Load Servlet" error.

Note: It's okay to rename a JAR file. The name of a JAR file is independent of the contents of the JAR file. Of course, you can't simply go around renaming JAR files if that JAR is referenced on a CLASSPATH somewhere else. So you have to be a *little* careful.

Case 1: Conclusion

Case 1: Conclusion



Webapp runs in Plugin

Understand structure of request URL

Determine if request "got over the wall"

- **HTTP Server 404 -- No**
- **HTTP Server 500 -- No**
- **Plugin 404 -- Yes**
- **Plugin 500 -- Yes**

If in Plugin, does message indicate problem finding servlet, or running servlet?

- **"Virtual Host or Web Application Not Found" -- Finding**
- **"404: No target servlet configured" -- Finding**
- **"500: Web group not found / failed to load servlet" -- Running**
- **"500: Server caught unhandled exception" -- Running**

Eliminate possibilities based on external symptoms, then go into traces

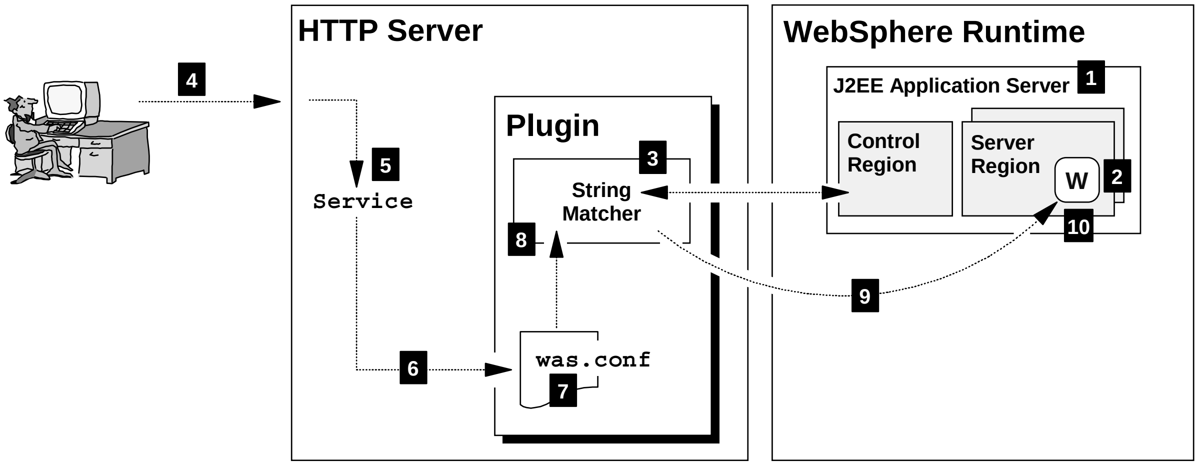
We finish the case where the web application is running within the Plugin. The main points are stressed on the chart:

- **Understand URL** -- if you think the URL is "x" when in fact it's "y", you'll have a heck of a time figuring out why things aren't working when you visually look at the configuration settings. Always validate the URL before you start.
- **Determine if the URL made it into the Plugin** -- and you can do this from the error condition you get. If you can tell that it made it into the Plugin, you don't need to mess with issues of Service statements or whether the Plugin's initialization parameters are correct.
- **If in the Plugin, is the trouble *finding* a servlet, or *running* the servlet** -- here again, the error message you get will tell you this.

It's possible to figure out a good many different problems without going into a trace.

Case 2: Plugin Acts as Router to Web Container in Runtime

Case 2: Plugin Acts as Router



For things to work properly ...

1. **J2EE Application Server started**
2. **WebApp bound to virtual host**
3. **Plugin and Runtime "shake hands"**
4. **Request able to get to server**
5. **Request matches to Service**
6. **Request gets into Plugin**
7. **No "Rooturi / Virtual Host" match in was.conf**
8. **Request matches to string in "string matcher table"**
9. **Flow back to runtime open (low odds of failure here)**
10. **Request servletmapping matches deployed webapp, class file found and loadable**

We're going to shift gears now and focus on the case where the web application runs in the web container of the J2EE application server, back in the WebSphere V4 runtime environment. In this case, the Plugin acts simply as a "router" of sorts, catching the URL request and sending it back to the web container for servlet execution.

This is a bit more complicated environment, but shares a good many of the same debugging concepts as "Case 1" presented earlier. For this to work, the following must happen:

1. The J2EE application server in which the web application is deployed must be started. If it's not started, the WebSphere V4 runtime will never communicate to the Plugin the information about the deployed webapps in that J2EE server.
2. The web application's "contextroot" (exactly like the "rooturi" from earlier) must be "bound" to a virtual host. All that means is that the contextroot value must be associated with a virtual host value. That binding takes place at J2EE server startup, and is based on the coding in the `webcontainer.conf` file.
3. The Plugin and the WebSphere V4 need to "shake hands," which means the Plugin needs to establish a communication link with the Systems Management Server (SMS) and then receive from the SMS a list of all the available servers and deployed web applications.
4. The request must be able to get to the HTTP Server. This is just like in Case 1.
5. The request must match a `Service` statement, just like in Case 1.
6. The request must get into the Plugin, just like in Case 1.

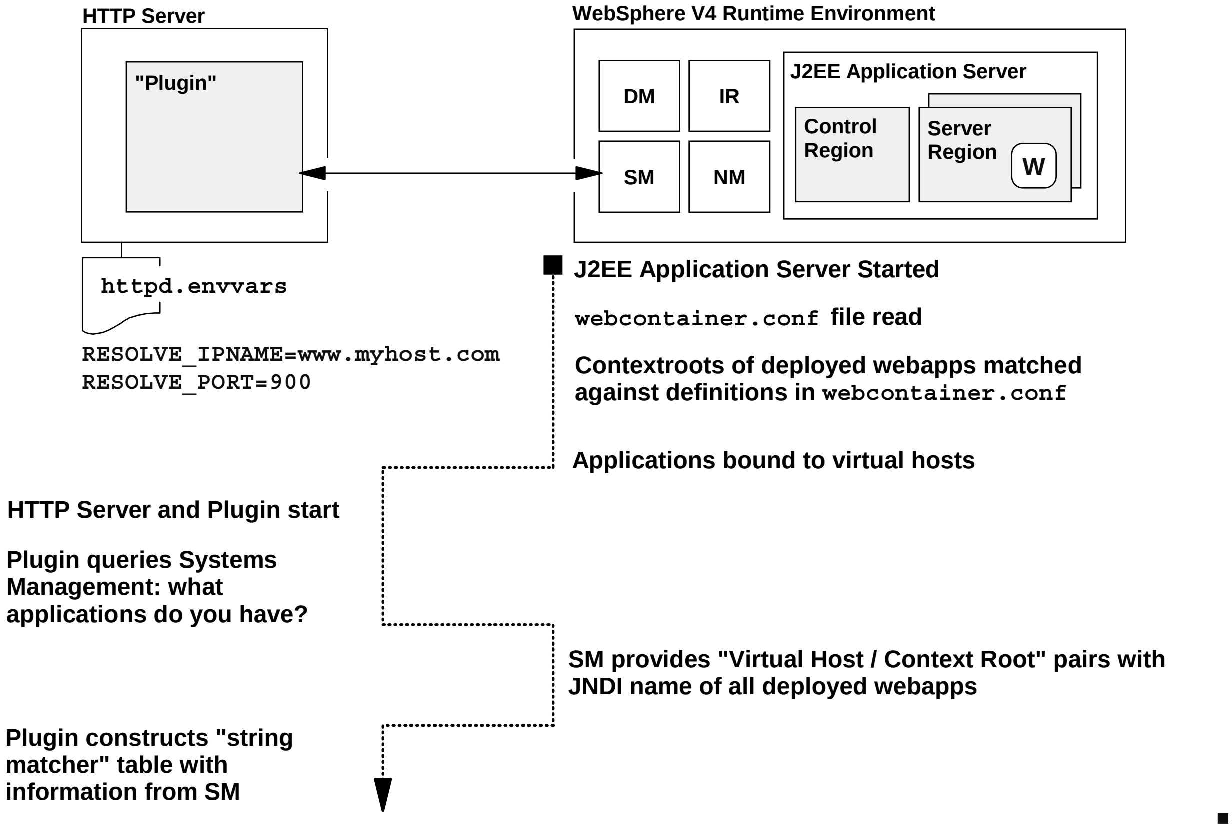
7. The Plugin has to *fail to find* in its local `was.conf` file (that's right, *fail* to find) any "rooturi / virtual host" match. If it gets a hit on a local rooturi value, it'll think it's supposed to run a webapp locally like we explored in "Case 1." But that's not what we want to do here. So we need to make sure the Plugin sees *no local definitions*. That'll allow the Plugin to consider passing the request back to the runtime.
8. The Plugin will next match the request against an internal table known as the "String Matcher Table." That's just a summarized list of the available web applications back in the WebSphere V4 runtime. That list came from the runtime when the two of them "shook hands" earlier. The "String Matcher Table" consists of "virtual host + contextroot" pairs, and the JNDI name of the home interface of that web application. The Plugin uses a match against the "String Matcher Table" to know to pass the request back to the runtime. If a match on the "virtual host + context root" occurs, then the request is passed to the runtime. If not, you get an error.
9. The request needs to get back to the runtime. There's little chance this will be a problem. If the handshake took place, then the communication link must be open.
10. Once in the web container, the servletmapping portion of the URL must match against the servletmapping string defined on a deployed application. Recall that the "string matcher table" in the the Plugin matches only on virtual host and context root, so it's possible that the request will be passed to the runtime and yet the servletmapping portion on the URL not match an application. If it does match, then WebSphere will locate and attempt to load the servlet.

    Note: Because the installation of the code into the HFS is done by the Systems Management Server (SMS), you likely won't see many of the typo problems seen in WebSphere V3.5. So if you get a hit on the servletmapping string, the chances of not finding the servlet class file are small.

Now let's focus on the "handshake" that takes place between the runtime and the Plugin.

Background: Interaction Between Plugin and WebSphere V4 Runtime

What's Behind Steps #1, #2 and #3



RESOLVE_IPNAME=www.myhost.com
RESOLVE_PORT=900

J2EE Application Server Started

`webcontainer.conf` file read

Contextroots of deployed webapps matched against definitions in `webcontainer.conf`

Applications bound to virtual hosts

HTTP Server and Plugin start

Plugin queries Systems Management: what applications do you have?

SM provides "Virtual Host / Context Root" pairs with JNDI name of all deployed webapps

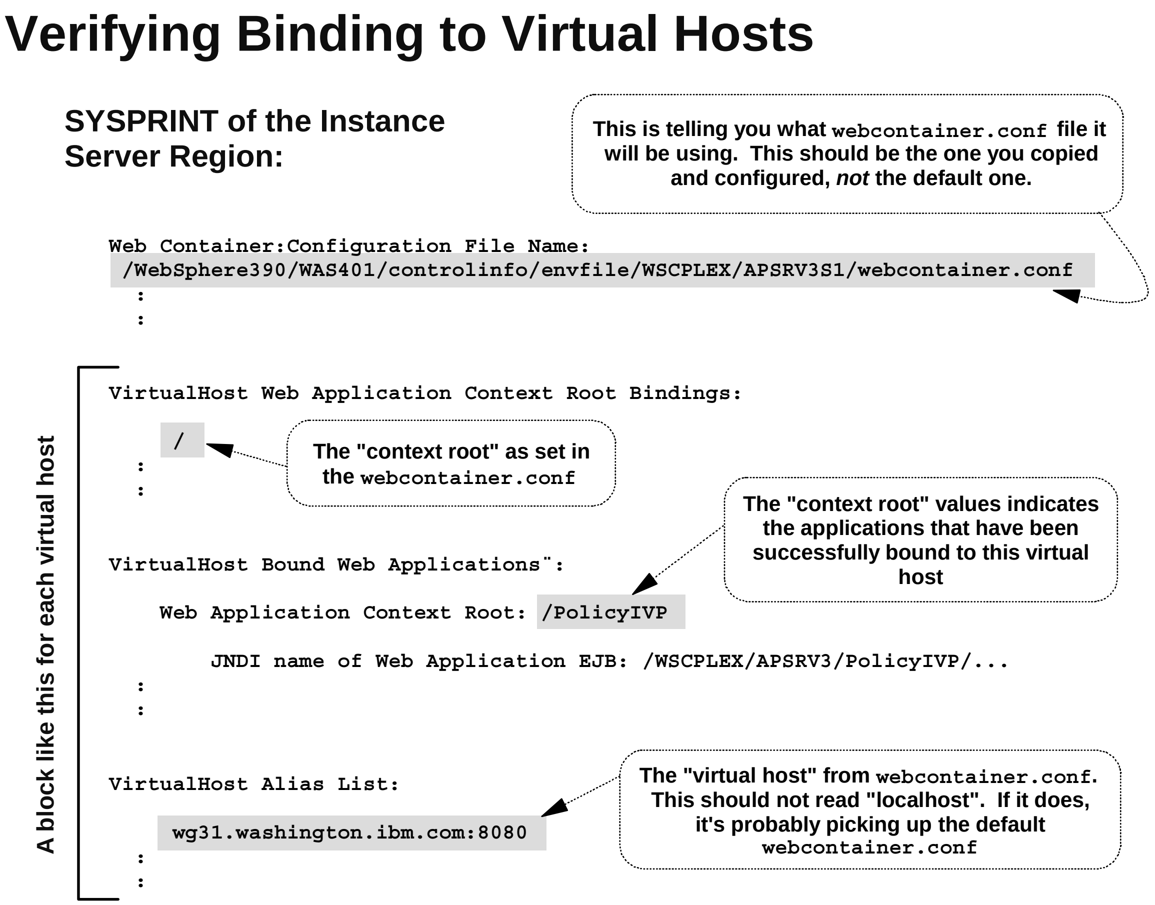
Plugin constructs "string matcher" table with information from SM

Here's a high-level of the process that takes place when the runtime and the Plugin shake hands and exchange information:

- When the J2EE application server in which an application is deployed is started, it reads the `webcontainer.conf` file for that server.
- The server will now "bind" the web application's context root to a virtual host. This is done based on how the `webcontainer.conf` is coded.
- At some point the HTTP Server is started, and the Plugin initializes.
- Based on two updates to the `httpd.envvars` file, the Plugin will send a message to the Systems Management Server and ask what applications it knows about for all started J2EE application servers in the WebSphere node.
- The SMS will return all the "Virtual Host + Context Root" pairs it knows about, based on the binding of context roots to virtual hosts that takes place when the J2EE servers start up. In addition to the virtual host and context root, the SMS will return the JNDI home interface
- The Plugin maintains all this "Virtual Host + Context Root + JNDI name" in an internal table called the "String Matcher" table. It is against that internal table that a new request is compared to see whether it is to be sent to the runtime.

A common question is, "Does the order of startup matter?" In fact, no. You can start the HTTP Server first and then the J2EE application server. The "hand-shake" is not a one-time thing. There's a polling cycle the Plugin goes through to synchronize itself with the runtime. So if you start your J2EE application server after the Plugin is already up, the Plugin may not know about the J2EE server's applications right away, but within a minute or two it will.

Verification that WebApps Successfully Bound to Virtual Host



Verifying Binding to Virtual Hosts

SYSPRINT of the Instance Server Region:

This is telling you what `webcontainer.conf` file it will be using. This should be the one you copied and configured, *not* the default one.

```
Web Container:Configuration File Name:
/WebSphere390/WAS401/controlinfo/envfile/WSCPLEX/APSRV3S1/webcontainer.conf
  :
  :


VirtualHost Web Application Context Root Bindings:

    /
  :
  :


VirtualHost Bound Web Applications¨:

    Web Application Context Root: /PolicyIVP

        JNDI name of Web Application EJB: /WSCPLEX/APSRV3/PolicyIVP/...
  :
  :


VirtualHost Alias List:

    wg31.washington.ibm.com:8080
  :
  :
```

The "context root" as set in the `webcontainer.conf`

The "context root" values indicates the applications that have been successfully bound to this virtual host

The "virtual host" from `webcontainer.conf`. This should not read "localhost". If it does, it's probably picking up the default `webcontainer.conf`

This document will not go into how to code up your webcontainer.conf so that applications bind to virtual hosts. You can get that information from white paper WP100238, which is out on the IBM support webpage at:
`http://www.ibm.com/support/techdocs/atsmastr.nsf/PubAllNum/WP100238`
However, I do want to cover the topic of *validating* that the binding did in fact take place. There are two ways to do this, and the most direct way is to take a look at the SYSPRINT of the J2EE application server's server region (not the control region). That SYSPRINT will show two key things:

- The `webcontainer.conf` file in use by the server. The file to be used by a server is pointed to with a property you add to the `jvm.properties` file. If you accidentally mis-type the property in jvm.properties, WebSphere will fail to get the `webcontainer.conf` file you want it to get. If it can't get the file you intended, it'll fall back and use the *default* `webcontainer.conf`. That's a problem, because the default `webcontainer.conf` won't have any of your virtual hosts defined. So it's important to make certain you successfully get the `webcontainer.conf` you intended.

  Note: The default `webcontainer.conf` can be found at the following HFS location:

  `/usr/lpp/WebSphere/bin/webcontainer.conf`

  If in the SYSPRINT you see your server making use of this default `webcontainer.conf` file, you know you have a problem with your pointer in the `jvm.properties` file.

- A listing of the what applications bound to what virtual hosts. If an application successfully binds to a virtual host, then that information will be passed over to the Plugin for the "string matcher" table. If the application doesn't bind to a virtual host, then there's no way to run that application. Therefore, it's critical to verify that the binding took place successfully.

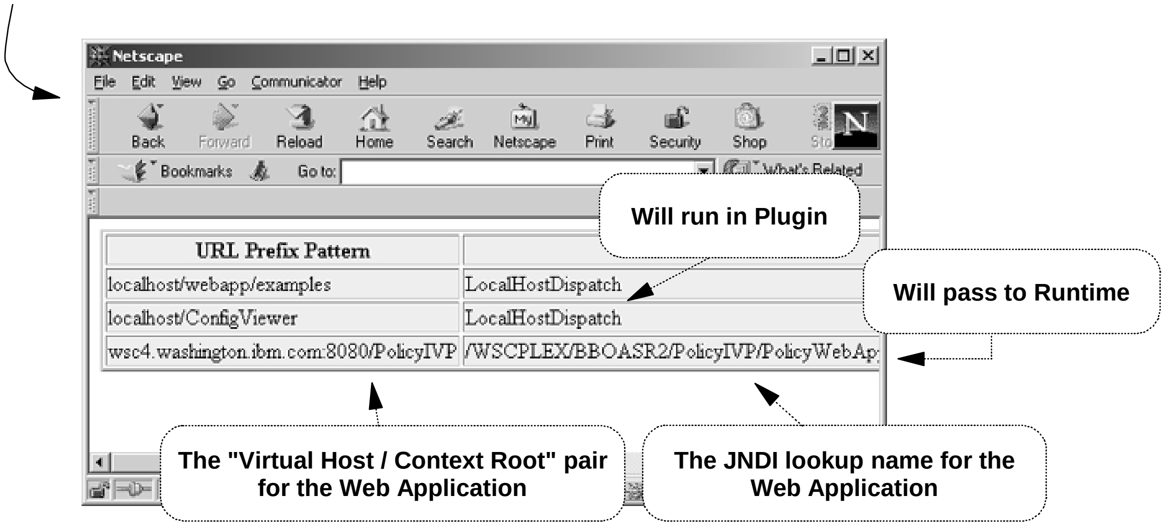
Verification that the Plugin-Runtime Handshake Took Place

Verifying the Handshake Took Place

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

1. Select **"Show server configuration"**
2. Scroll one page and select **"Application Dispatching Information"**

This is a function of the Plugin. It reports back what the Web Container says it has bound


Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Sto
Bookmarks Go to: What's Related

URL Prefix Pattern	
localhost/webapp/examples	LocalHostDispatch
localhost/ConfigViewer	LocalHostDispatch
wsc4.washington.ibm.com:8080/PolicyIVP	/WSCPLEX/BBOASR2/PolicyIVP/PolicyWebAp

Will run in Plugin

Will pass to Runtime

The "Virtual Host / Context Root" pair for the Web Application

The JNDI lookup name for the Web Application

The presence of your webapp in this table means the plugin has communicated with the web container and has knowledge of your application

The second way you can verify if the application bound to a virtual host -- and also to see the contents of the Plugin's "string matcher table" -- is to take a look at the "Application Dispatching Information." This is a servlet that comes with the Plugin, and what it does is display on a browser screen the string matcher table the Plugin is maintaining in its memory. The URL used to get at the Application Dispatching servlet is:

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

Once at that front page, you simply click on the "Show server configuration" link, then on the "Application Dispatching" link. That will run the servlet, which will format and display the contents of the string matcher table.

Taking a look at this page provides two key pieces of validation:

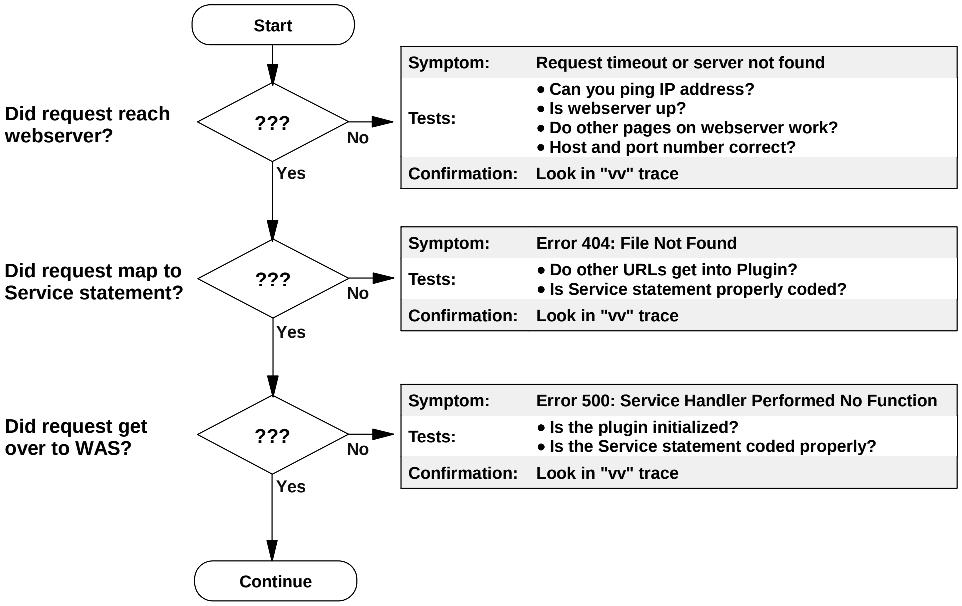
- If your application appears in the table, it means the Plugin and the runtime shook hands and exchanged the information. That means if a request is received by the Plugin and matches your application's virtual host and context root string, the Plugin will pass the request back to the runtime.
- If your application appears in the table, that means your application successfully bound to a virtual host. This is the less direct way of verifying binding (the other being looking in the SYSPRINT).

It is a good practice to make use of this feature of the plugin as a quick way of validating that the application bound to a virtual host and that the runtime has exchanged information with the Plugin.

Case 2 Process for Getting into the Plugin the Same as in Case 1

Did Request Get Into Plugin?

Steps #4, #5 and #6


Start
Did request reach webserver?
???
No
Yes
Did request map to Service statement?
???
No
Yes
Did request get over to WAS?
???
No
Yes
Continue

Symptom:	Request timeout or server not found
Tests:	● Can you ping IP address? ● Is webserver up? ● Do other pages on webserver work? ● Host and port number correct?
Confirmation:	Look in "vv" trace

Symptom:	Error 404: File Not Found
Tests:	● Do other URLs get into Plugin? ● Is Service statement properly coded?
Confirmation:	Look in "vv" trace

Symptom:	Error 500: Service Handler Performed No Function
Tests:	● Is the plugin initialized? ● Is the Service statement coded properly?
Confirmation:	Look in "vv" trace

This is *exactly* like it was for "Case 1" ... the exact same debugging methodology applies

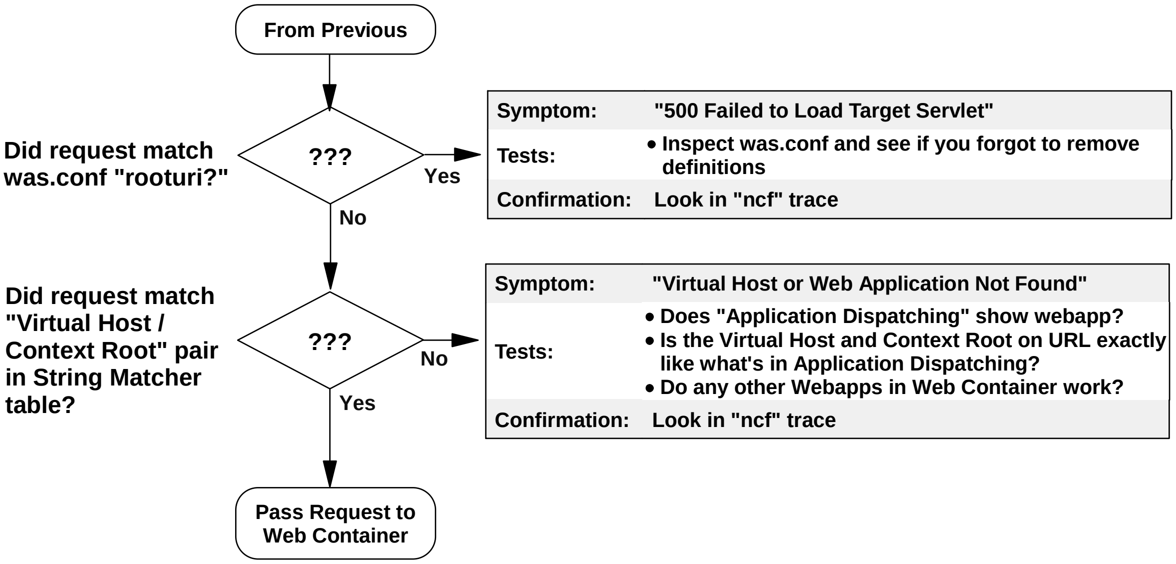
Back in "Case 1" we talked about how to dbug and validate whether the request was received by the HTTP Server, mapped to a `Service` statement, and actually got into a running copy of the Plugin.

In this case -- "Case 2" where the Plugin is acting as a router, passing requests back to the runtime -- the need to map to a `Service` and get into the Plugin is exactly like it is if the webapp was running in the Plugin. Therefore, the process by which you can debug problems getting into the Plugin is exactly the same as we discussed earlier.

Key point: use the external symptoms to determine if the request got into the Plugin. HTTP Server 404 errors or "500: Service Handler Performed No Function" errors mean the request didn't get into the Plugin.

Flowchart: Does URL Match "String Matcher" Table in V4 Plugin

Did Request Map to "String Matcher?"


From Previous
Did request match was.conf "rooturi?"
???
Yes
No
Did request match "Virtual Host / Context Root" pair in String Matcher table?
???
No
Yes
Pass Request to Web Container

Symptom:	"500 Failed to Load Target Servlet"
Tests:	• Inspect was.conf and see if you forgot to remove definitions
Confirmation:	Look in "ncf" trace

Symptom:	"Virtual Host or Web Application Not Found"
Tests:	• Does "Application Dispatching" show webapp? • Is the Virtual Host and Context Root on URL exactly like what's in Application Dispatching? • Do any other Webapps in Web Container work?
Confirmation:	Look in "ncf" trace

Key to mapping URL over to Web Container:

- **Don't get caught on old definitions left over in the was.conf ... clean that file**
- **Make sure Virtual Host and Context Root are exact matches (case and port number)**

Once we know the request made its way into the Plugin, then two questions present themselves:

- Did the request match a "rooturi + virtual host" setting in the local `was.conf` file? *To run a web application back in the runtime, you don't want it to catch on a local definition*. However, if the answer is "yes," then what that means is you probably forgot to clean an old definition out of `was.conf`. That'll probably result in a "500: Failed to Load Target Servlet" because no servlet is actually available to the Plugin to run.
- If no `was.conf` definition matched (which would be a good thing), then the next question is whether the request matched one of the entries in the "string matcher table." For the Plugin to pass the request back to the runtime, the request must match the "Virtual Host + Context Root" in the matcher table. If you fail to get a match, you'll see the "Virtual Host or Web Application Not Found" error message.

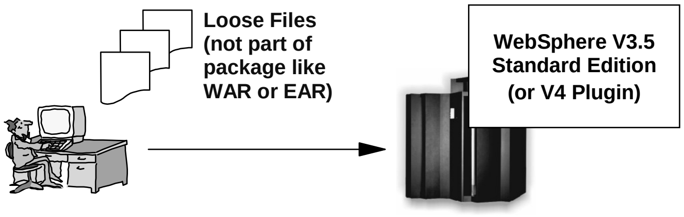
  If you see this error, you can use the "Application Dispatching" to see what the string matcher for the Plugin shows. It could be that the "handshake" between the Plugin and the runtime hasn't yet taken place, and the string matcher hasn't yet been updated. It could be that the request is just slightly different -- a case difference perhaps, or a port difference -- and therefore doesn't really match the string matcher table. The point is, you can use the "Application Dispatching" information to visually compare the request against what the Plugin will pass back to the runtime.

  Another quick test is to point your browser at another web application back in the web container to see if any webapps work. If you have a simple test webapp deployed back there for just such a test, you can validate that the J2EE server can be reached.

If you get past these two checkpoints, your request will be passed back to the runtime. Now let's look at what happens once you've over there.

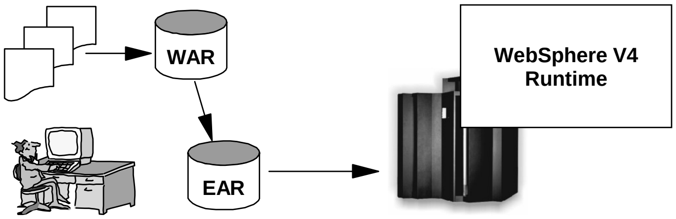
WebSphere V4 Results in Fewer Human Errors

Packaging Tools and Human Errors


Loose Files (not part of package like WAR or EAR)
WebSphere V3.5 Standard Edition (or V4 Plugin)

Lots of room for human error:

- **Improper FTP mode (ASCII/EBCDIC)**
- **Set permissions on files improperly**
- **Mis-code `was.conf` definitions**



Tools help reduce human error

- **FTP done automatically**
- **Permissions set automatically**
- **CLASSPATH handled by Runtime**

Message:

- Problems such as mis-coded CLASSPATH, class file dropped into wrong directory, typo in the **`was.conf`** definition have been reduced
- Mis-typed contextroot or servletmapping strings still possible

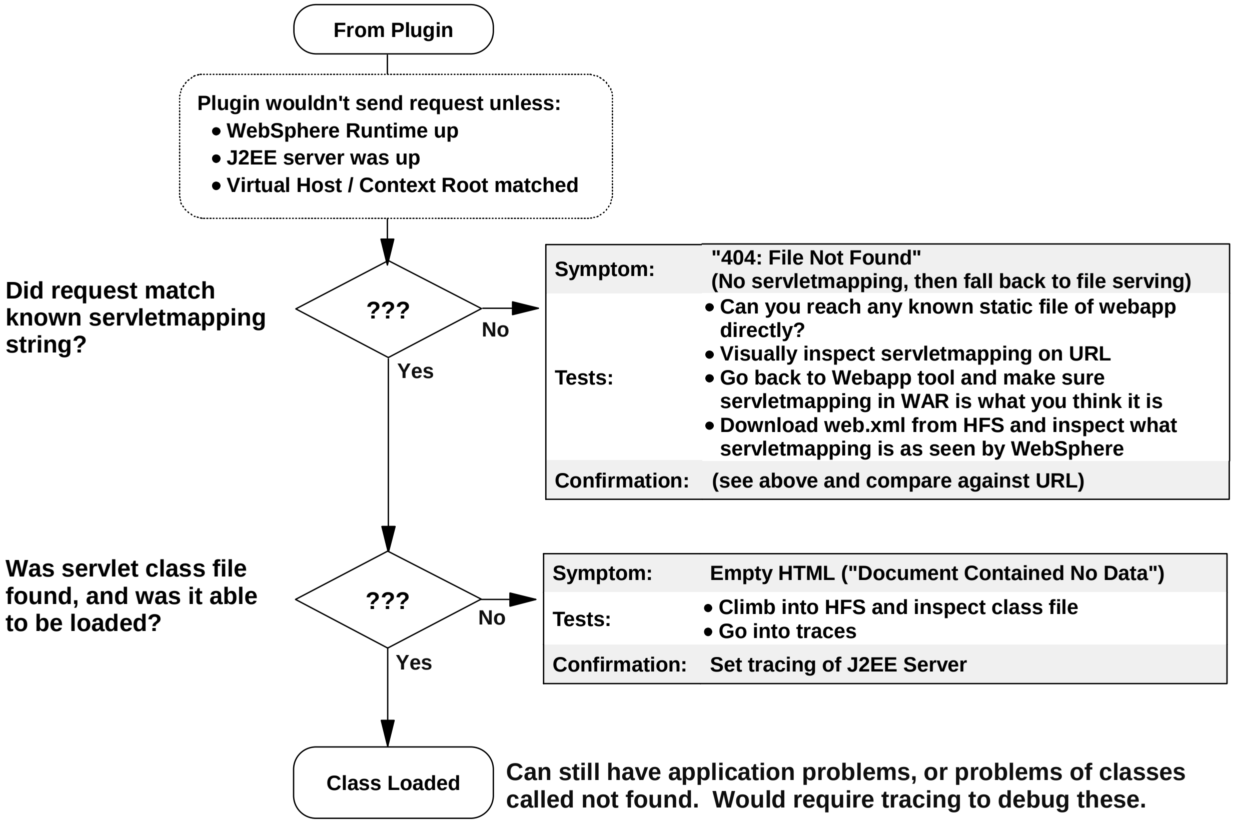
A quick point: in the WebSphere V3.5 environment (or, for that matter, the WebSphere V4 Plugin environment -- Case 1 in this presentation), there are lots of chances to make human errors when deploying an application. The individual files are hand-FTPed up to the servers, which means there's a chance they'll be transferred using the wrong mode. Any directories you create, or files you copy, may not have the right permissions. And there's always a chance to simply mis-type a directory or a string value into the `was.conf` file.

With the WebSphere V4 environment, the tools do the deploying for you. Therefore, all the issues of FTP modes, permissions and CLASSPATH updates are automatically handled. There's still a chance that you'll provide a garbled contextroot or servletmapping string for a webapp. But the chances for error have been reduced.

Now let's look at the logic flowchart once we're inside the web container.

Flowchart: Once Request Gets into V4 Web Container

Once Inside the Web Container



From Plugin

Plugin wouldn't send request unless:
- WebSphere Runtime up
- J2EE server was up
- Virtual Host / Context Root matched

Did request match known servletmapping string?

???

No

Yes

Symptom:	"404: File Not Found" (No servletmapping, then fall back to file serving)
Tests:	• Can you reach any known static file of webapp directly? • Visually inspect servletmapping on URL • Go back to Webapp tool and make sure servletmapping in WAR is what you think it is • Download web.xml from HFS and inspect what servletmapping is as seen by WebSphere
Confirmation:	(see above and compare against URL)

Was servlet class file found, and was it able to be loaded?

???

No

Yes

Symptom:	Empty HTML ("Document Contained No Data")
Tests:	• Climb into HFS and inspect class file • Go into traces
Confirmation:	Set tracing of J2EE Server

Class Loaded

Can still have application problems, or problems of classes called not found. Would require tracing to debug these.

Before a request will be passed from the Plugin over to the runtime, several things must be true, as shown in the dotted-line box at the top of this chart. If you avoided the error conditions shown on the previous flowchart, that means the request is over in the runtime. There, two questions come up:

- Did the request match a known servletmapping string? Having an error pop up on this is quite likely, particularly if the user hand-types the URL into the location window of their browser. The error you are likely to see is "404: File Not Found."

  Note: The web container has a sequence of processing much like the Plugin: it looks for a servletmapping first, then a JSP mapping second, and finally a file mapping. I'm showing the "404" error because that's the most likely error when you fail to hit on a servletmapping string. The web container will then look for a JSP mapping (default of `*.jsp` as the mask on that), and failing that, will default to assuming that the request is for a file. The contents of your URL will then be assumed to be a file name, and that will mostly likely not be found. Hence, a 404 error.

  Where does the servletmapping string for a webapp come from? It is given to the webapp at the time the WAR file is generated (by a tool such as WSAD), and the value is written in the `web.xml` file inside the WAR file. When the J2EE application server comes up, it looks into all the `web.xml` files of all the deployed applications and grabs the servletmapping information for each and caches that information. Then, when a URL comes in, WebSphere compares it against its known servletmapping strings.

  Note: If the servlet is resolved (in other words, the servletmapping string matched), but there was a problem with the servlet, you get a different error -- which is the next in this flowchart example.

Debugging this is tricky. The trace for this would be in the SYSPRINT, and can be difficult to wade through (how to get this trace will be discussed next). So if you know you've been passed to the web container, but there appears to be a problem driving the servlet, there are a few tests you can do:

- If the web application you're trying to drive has a static file (HTML file, or a GIF/JPG image), try getting that file directly. The "Virtual host + Context root" for the static file would be the same as that for the servlet, but you would simply name the file rather than provide a servletmapping string. If you can get the static file, then you've narrowed the problem to something having to do with the servletmapping string.
- Take a good close look at the structure of the URL and compare it against what the servletmapping string *should* be.
- If you're not sure what the servletmapping string *should* be, you can go to the webapp development tool you used and see what the servletmapping was set to there.
- To double-check that servletmapping string, you could as a last resort download the `web.xml` file from the HFS and inspect the value in that file.

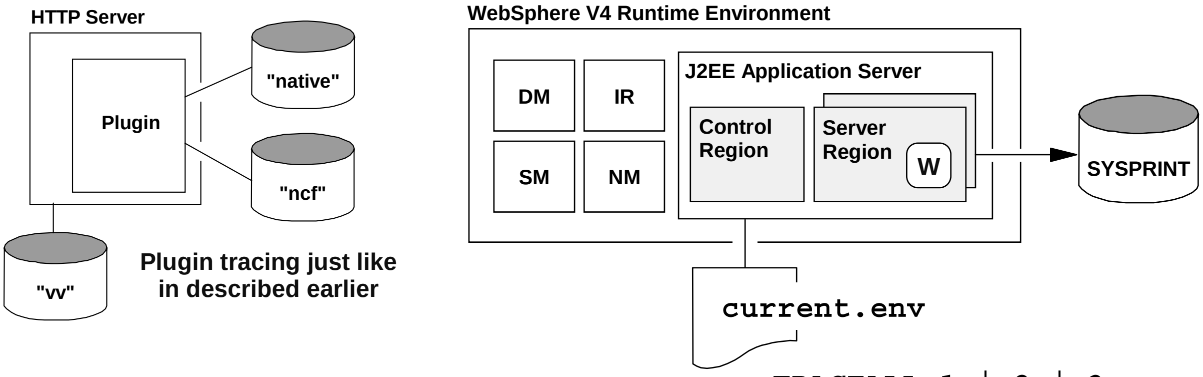
- Assuming the servletmapping string was resolved, WebSphere will then get the servlet class file name from the `web.xml` and go and try to load that. At this point, WebSphere will either a) fail to find the class file, b) find it, but not be able to read it (permissions or file bad or corrupt), or c) find it, read it and invoke it.

  Because the deployment tool stores the files and sets the classpath information, it's not likely that the file won't be found or be corrupt. But just in case it is, the error message you receive is an HTML file, to which the browser will throw the "Document Contained No Data" message.

  There's not much you can do to debug this further. You can go into the HFS and visuallly inspect the class file to make sure that it's there and the permissions are okay. But as mentioned earlier, the deployment tool takes care of this, and the chances of the file not being there or having bad permissions is small. Beyond that, to get deeper into this problem requires the tracing function, which is discussed next.

Sources of Trace Information for WebApp in Web Container

Sources of Trace Information



```
TRACEALL=1 | 2 | 3
TRACEBUFFLOC=SYSPRINT
```

Application runs in Server Region, which is controlled by settings in `current.env`

Set *amount* of tracing with TRACEALL

Set *location* of tracing output with TRACEBUFFLOC

A lot of information; use other symptoms to debug before this

Tracing of the J2EE application server is controlled by a few environment variables in the `current.env` file, which each server instance has. Two variables come into play:

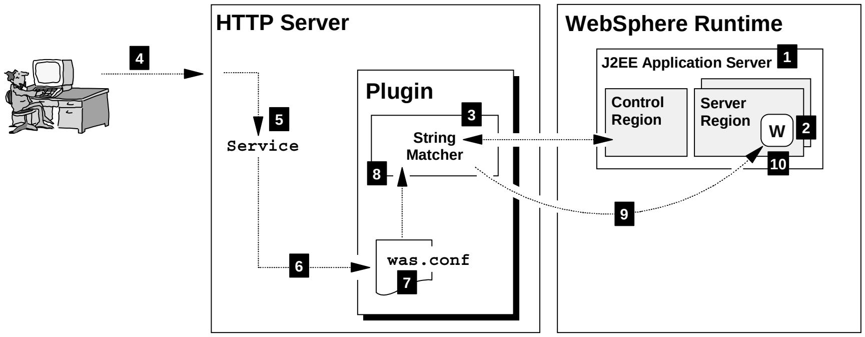
- TRACEALL -- this sets the amount of tracing; 1=least, 3=most
- TRACEBUFFLOC -- this sets the location of the tracing output. It defaults to BUFFER, which means you need to run a trace formatting job. A value of SYSPRINT puts it to that location.

> **Note:** There are other tracing capabilities available, and are intended to be used only at the direction of IBM support. This tracing capability is the basic set, and can be used for first-cut debugging.

It is not the intent of this presentation to show you how to read this trace file. The output is very long and detailed. The purpose of showing you this is to tell you how to invoke the traces, and to suggest that you use other sources of debugging information prior to digging into traces.

Case 2: Conclusion

Case 2: Conclusion



Four key things:

- Did webapp bind to virtual host? -- Review SYSPRINT of server region
- Does Plugin "shake hands" with Runtime? -- Review "Application Dispatching"
- Is the Plugin initialized? -- Check for "smiley face"
- Can URL "get over the wall?" -- Provide Service statement in httpd.conf

Understand structure of request URL

- Service statement dependent on this
- Routing to Runtime based on match with "string matcher table"

Use external symptoms to figure out how far request got

- HTTP Server Error 404? -- still in webserver
- "Virtual Host or Web Application Not found" -- into Plugin, but not routed to runtime
- Recursive Error with "404 File Not Found" -- into Runtime, but no servletmapping match

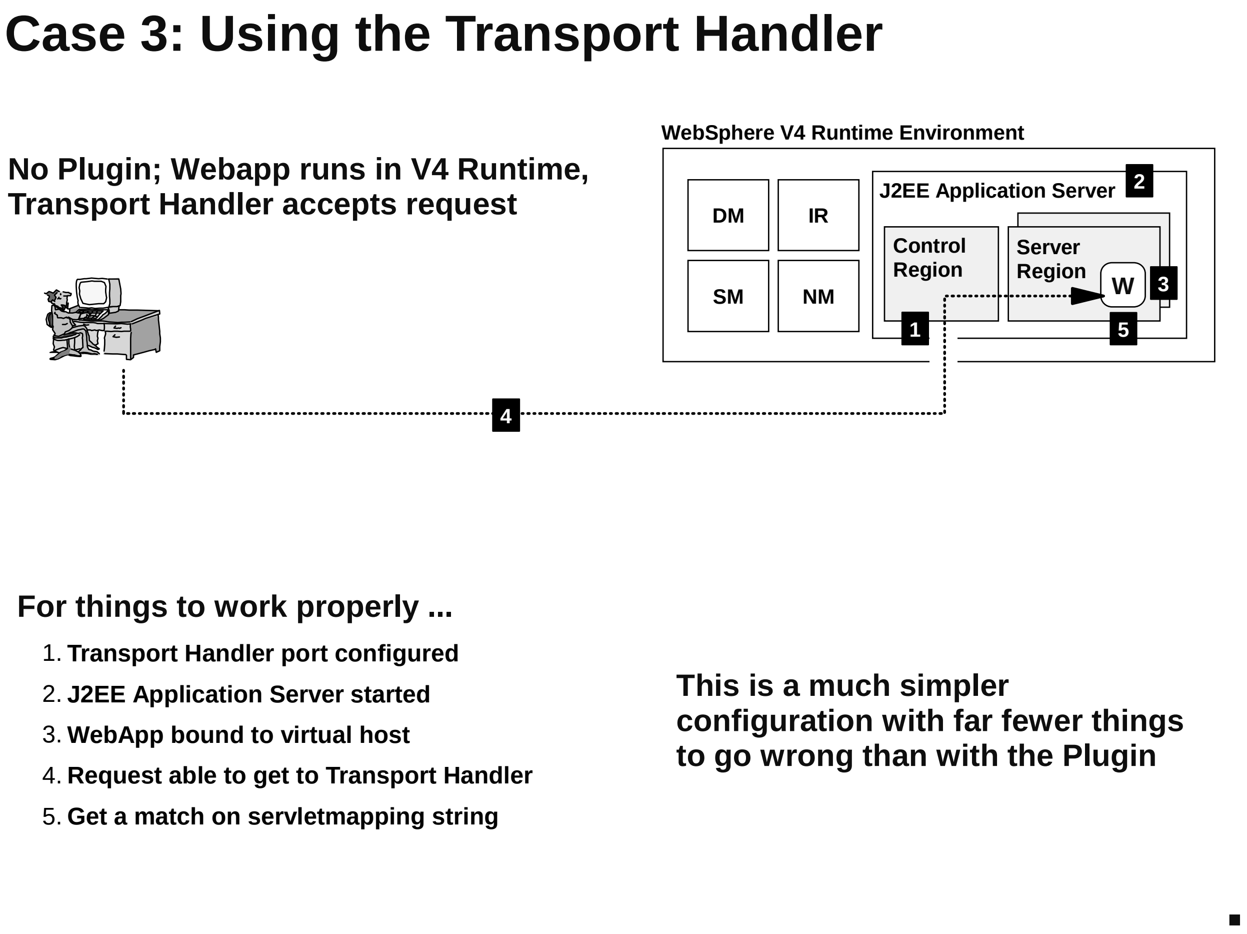
To debug this setting, you need to focus on four key things, as shown in the chart above. Earlier in this section we talked about things you can check to insure that four key things were accomplished. You can determine these without ever going into any of the traces.

Knowing the precise structure of the URL becomes important because the proper operation of the Service statement is dependent on it, as is the matching to the string matcher table, as is the matching to the servletmapping string.

Finally, you can use the external symptoms to determine how far the request got in this system. The chart shows what error messages come from what sources.

Now let's turn to the issue of the Transport Handler. That's Case #3 ...

Case 3: Handling HTTP Requests Through the Transport Handler



Case 3: Using the Transport Handler

No Plugin; Webapp runs in V4 Runtime, Transport Handler accepts request

For things to work properly ...

1. **Transport Handler port configured**
2. **J2EE Application Server started**
3. **WebApp bound to virtual host**
4. **Request able to get to Transport Handler**
5. **Get a match on servletmapping string**

This is a much simpler configuration with far fewer things to go wrong than with the Plugin

This is the simplest scenario ... by using the Transport Handler you can eliminate the Plugin entirely from the picture, and by doing so, eliminate all the complexity of the Service statements, Plugin initialization, and the "handshake" between the Plugin and the runtime. The request from the user goes directly to the control region of the J2EE application server, and from there directly to the web container.

Note: *"...you can eliminate the Plugin entirely from the picture."* This doesn't mean you *have* to eliminate it. It is quite possible to have the Plugin *and* the Transport Handler in the picture together; the Plugin handling some requests and the Transport Handler others.

For this to work, only a few things need to be in place:

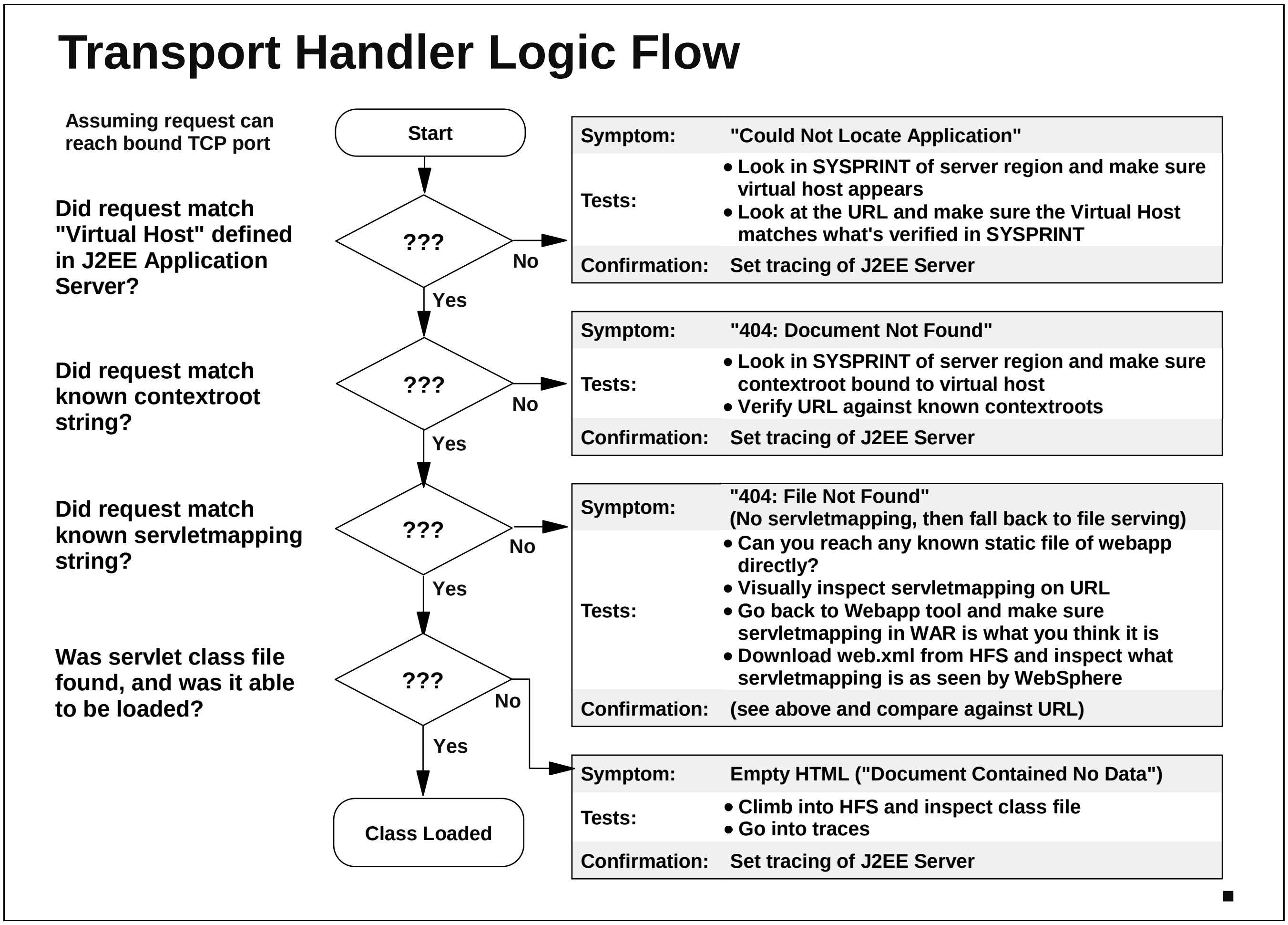
1. The Transport Handler port needs to be configured (which is nothing more than a simple environment variable update to the `current.env` file for the J2EE server instance)
2. The J2EE application server needs to be started (just like it did in Case #2)
3. The web application needs to be bound to a virtual host (just like it did in Case #2)

   Note: This involves having a valid `webcontainer.conf` file, properly pointed to from the `jvm.properties` file, with valid `contextroots=` and `alias=` coding to allow the webapp to bind to the virtual host. See **`WP100238`** on `www.ibm.com/support/techdocs` for more details.

4. The request must be able to get across the network to the port on which the Transport Handler is listening (this has to do with whether the network can get the request to the TCP port)
5. The request must match a servletmapping string known by the web container (just like in Case #2)

As you can see, the process is very much similar to Case #2, just without all the Plugin processing. Let's now look at the logic flow for debugging purposes.

Flowchart: Transport Handler



Transport Handler Logic Flow

Assuming request can reach bound TCP port

Did request match "Virtual Host" defined in J2EE Application Server?

Symptom:	"Could Not Locate Application"
Tests:	• Look in SYSPRINT of server region and make sure virtual host appears • Look at the URL and make sure the Virtual Host matches what's verified in SYSPRINT
Confirmation:	Set tracing of J2EE Server

Did request match known contextroot string?

Symptom:	"404: Document Not Found"
Tests:	• Look in SYSPRINT of server region and make sure contextroot bound to virtual host • Verify URL against known contextroots
Confirmation:	Set tracing of J2EE Server

Did request match known servletmapping string?

Symptom:	"404: File Not Found" (No servletmapping, then fall back to file serving)
Tests:	• Can you reach any known static file of webapp directly? • Visually inspect servletmapping on URL • Go back to Webapp tool and make sure servletmapping in WAR is what you think it is • Download web.xml from HFS and inspect what servletmapping is as seen by WebSphere
Confirmation:	(see above and compare against URL)

Was servlet class file found, and was it able to be loaded?

Symptom:	Empty HTML ("Document Contained No Data")
Tests:	• Climb into HFS and inspect class file • Go into traces
Confirmation:	Set tracing of J2EE Server

■

Assuming that the request is able to reach the TCP port bound to the Transport Handler (you can verify that the control region is listening on the port by issuing a `TSO NETSTAT` command), then there's a series of questions:

- Does the "Virtual Host" of the request match one known by the web container? This will be based on the applications bound to the virtual hosts based on the coding of the `webcontainer.conf` file.

  Note: What's interesting is that the Transport Handler, unlike the Plugin, does a two-stage check here. First it checks the virtual host value, and then it checks the contextroot value. There are two different error symptoms.

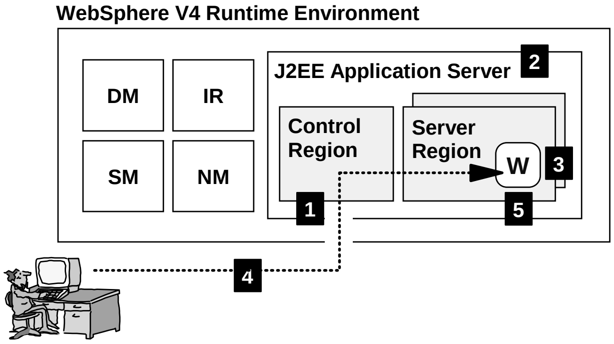
  If the request doesn't match a "Virtual Host" the Transport Handler will reject the request with a rather bland "Could Not Locate Application" message. If you see this, you know there's either a problem with your URL not being correct, or the application didn't bind to a virtual host. You can verify the binding by looking in the SYSPRINT of the server region.

- The next question is whether or not the "contextroot" on the URL matches a known and bound contextroot in the web container. Here the Transport Handler will look for a match on the "virtual host + contextroot" pair. If it can't find a hit, it will throw a "404: No Document Found" error. To the best of my knowledge, this is the only place this particular error is issued.

- The next question is whether or not the servletmapping string will match one known by WebSphere. This is just like was described for Case #2. You will likely see a "404: File Not Found," which means WebSphere defaulted back to considering the request to be that for a file, and not finding the file. As described in the Case #2 example, the method for checking this is to check the URL against the servletmapping value set for the web application.

- If you resolve to a servlet, but the servlet itself can't load, by default you'll get the mysterious "Document Contained No Data" on the browser screen. This is not a pleasant message to debug. The web container tried to load the specified class file, but could not for some reason or another. The only way to get anywhere near what's going wrong is to climb into the J2EE server trace, and that can be a big effort to make heads or tails of what you see in there

Case 3: Conclusion



Very simple to configure; fewer things to go wrong

Use SYSPRINT to verify application bound to Virtual Host

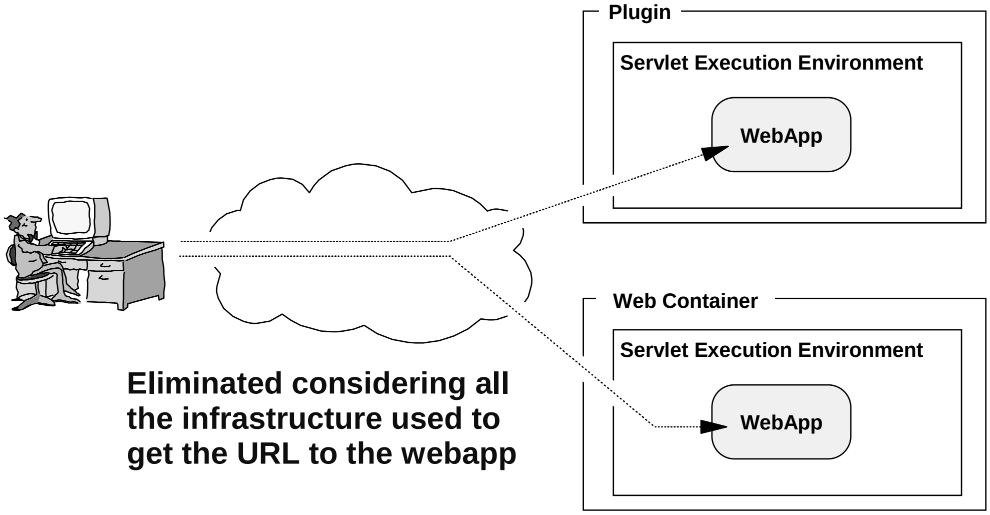
Use TSO NETSTAT to insure Control Region listening on port

Problems will most likely be result of mistyped Virtual Host, Context Root or Servletmapping string

When the scenario involves just the Transport Handler, the environment is simple enough that many of the other errors will never occur. The key is to make sure the application is bound to a virtual host, and that the Transport Handler is listening on the designated port. Most problems with this environment will involve mistyped virtual host, context root or servletmapping strings.

Other Problems Might Still Exist

Problems Still in the "Problem Space"



Eliminated considering all the infrastructure used to get the URL to the webapp

Webapp resolved, class file found and loaded

- **Dependent class files not found**
- **Data subsystems not available or not properly coded to**
- **Unhandled exceptions in application**
- **Other application-related problems**

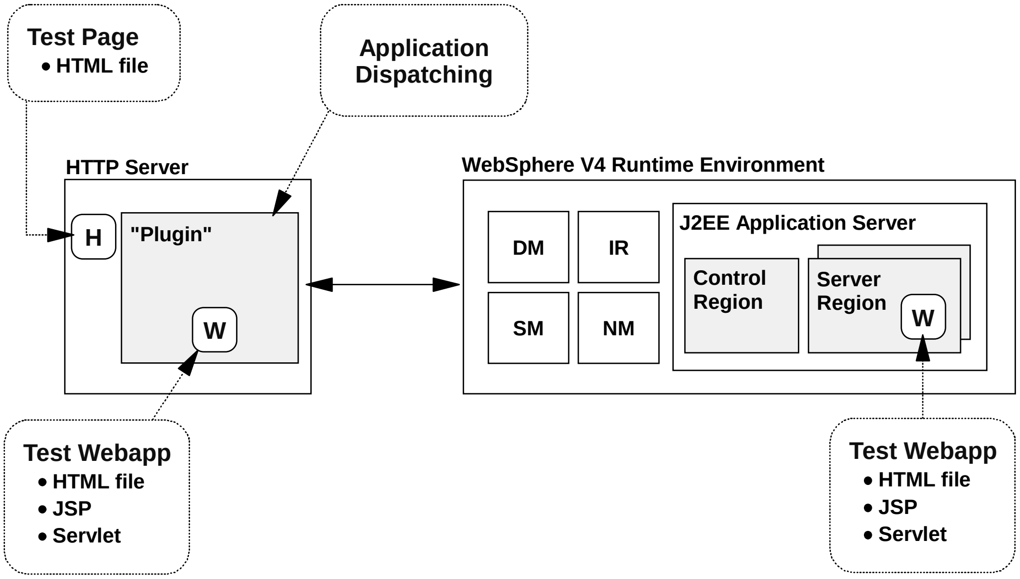
This is "advanced" debugging activity

Throughout the course of this presentation, the focus has been on eliminating from consideration those problems related to getting the URL to the servlet execution environment and actually loading the webapp. Once the servlet class file has been located and loaded, there's still more errors that can occur, unfortunately. Some of those are illustrated in the bullet list at the bottom of the chart.

Debugging those problems is a separate subject. Problems within the application itself may require tracing and trapping of the application itself, and will very likely require the involvement of the developer.

But the key here is this: before you bring the developer into the picture, or start turning on traces, it is best to eliminate from consideration those problems outlined in the this paper. Then you can go with confidence into the more time-consuming debugging activities.

Provide Yourself Some Monitoring Tools



Point is to provide yourself with things in your system you can go to when you need to quickly verify that basic connectivity and operation is working.

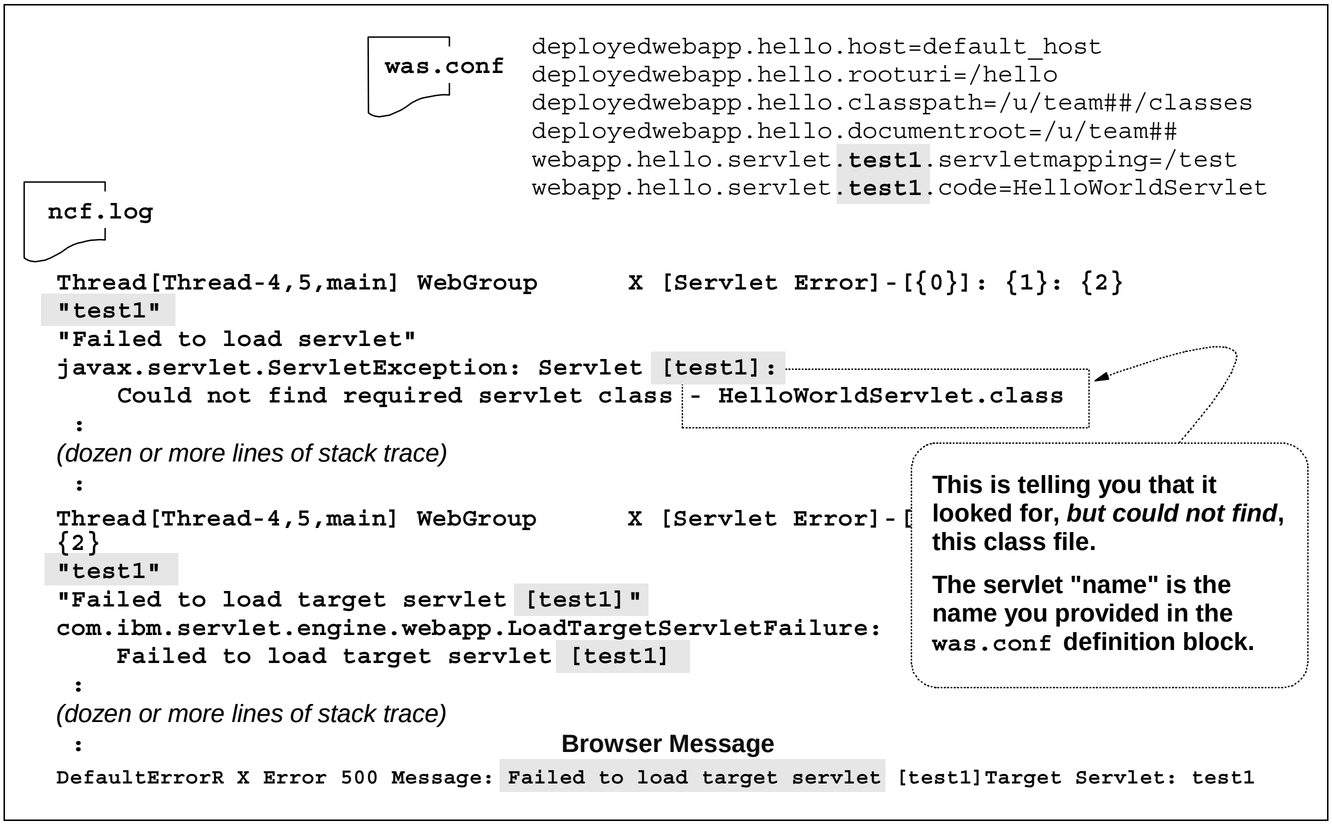
▪

As a final point, it is recommended that you provide yourself some tools in your system that you can use to verify the basic operation of key things.

- **Test HTML Page** -- have a static HTML page deployed in your HTTP Server that can serve as a test of access to the web server. You can point your browser at this page and confirm that the webserver is up and at least serving HTML pages.
- **Test Plugin Webapp** -- have a test webapp deployed in the Plugin so that you can verify that the Plugin is working and able to execute servlets and JSPs. The supplied `/webapp/examples` servlet can serve this function.
- **Application Dispatching** -- this function, which is provided with the WebSphere V4 Plugin, is a fine tool to use to verify the connectivity between the Plugin and the WebSphere V4 runtime. Make use of this to monitor what applications are available in the runtime and to verify that the Plugin and the runtime are talking to one another.
- **Test Webapp in Web Container** -- have a simple test web application in the WebSphere runtime environment. This simple web application should have a static file (HTML, JPG/GIF), a very simple JSP and a simple servlet. This will allow you to test connectivity to the runtime environment (either through the Plugin or the Transport Handler) and verify that things back there are working. If you can get to your test application, you don't need to worry about many of the pieces of the puzzle leading up to the web container.

Appendix A: Plugin ncf Trace Examples

The following illustrates what you will find in the "ncf" trace (one of the two traces put out by the Plugin) when a "class not found" condition is encountered:



```
deployedwebapp.hello.host=default_host
deployedwebapp.hello.rooturi=/hello
deployedwebapp.hello.classpath=/u/team##/classes
deployedwebapp.hello.documentroot=/u/team##
webapp.hello.servlet.test1.servletmapping=/test
webapp.hello.servlet.test1.code=HelloWorldServlet
```

```
Thread[Thread-4,5,main] WebGroup      X [Servlet Error]-[{0}]: {1}: {2}
"test1"
"Failed to load servlet"
javax.servlet.ServletException: Servlet [test1]:
    Could not find required servlet class - HelloWorldServlet.class
  :
(dozen or more lines of stack trace)
  :
Thread[Thread-4,5,main] WebGroup      X [Servlet Error]-[
{2}
"test1"
"Failed to load target servlet [test1]"
com.ibm.servlet.engine.webapp.LoadTargetServletFailure:
    Failed to load target servlet [test1]
  :
(dozen or more lines of stack trace)
  :
```

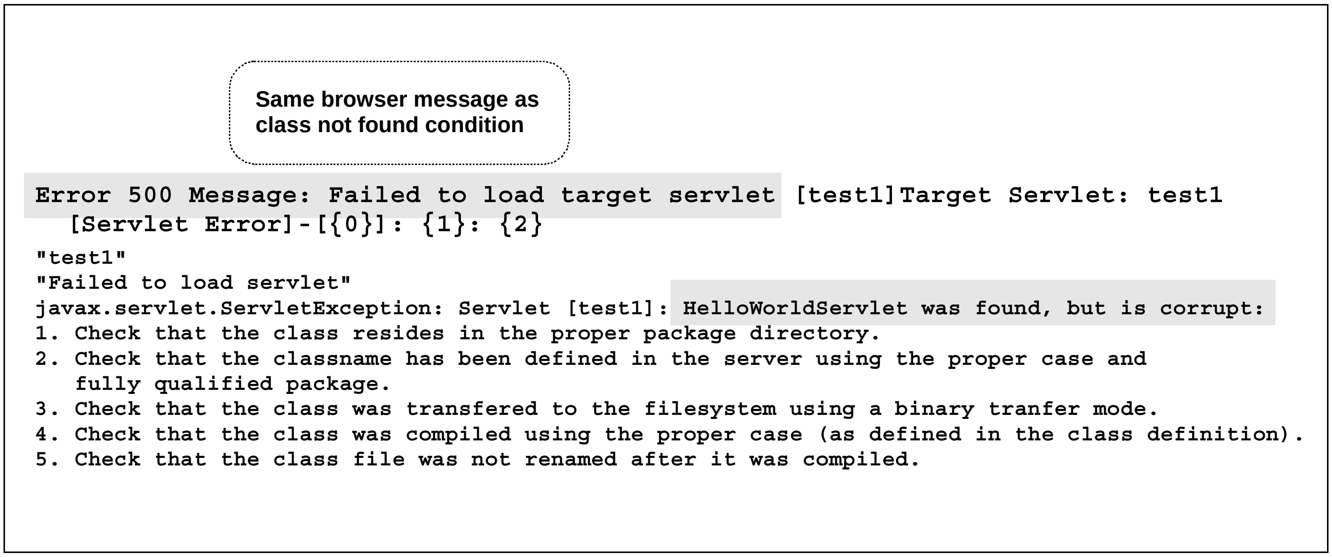
This is telling you that it looked for, *but could not find*, this class file.

The servlet "name" is the name you provided in the `was.conf` definition block.

Browser Message

```
DefaultErrorR X Error 500 Message: Failed to load target servlet [test1]Target Servlet: test1
```

And the following illustrates what you will see in the "ncf" trace when a servlet class file is found, but for whatever reason can't be loaded. As you see, the trace will provide you with a list of suggested causes for the problem:



Same browser message as class not found condition

```
Error 500 Message: Failed to load target servlet [test1]Target Servlet: test1
  [Servlet Error]-[{0}]: {1}: {2}
"test1"
"Failed to load servlet"
javax.servlet.ServletException: Servlet [test1]: HelloWorldServlet was found, but is corrupt:
1. Check that the class resides in the proper package directory.
2. Check that the classname has been defined in the server using the proper case and
   fully qualified package.
3. Check that the class was transfered to the filesystem using a binary tranfer mode.
4. Check that the class was compiled using the proper case (as defined in the class definition).
5. Check that the class file was not renamed after it was compiled.
```

End of Document