IBM WebSphere Software

**WebSphere Application Server for z/OS and OS/390**

# Configuring the WebSphere Plug-in with WebSphere V5 for z/OS

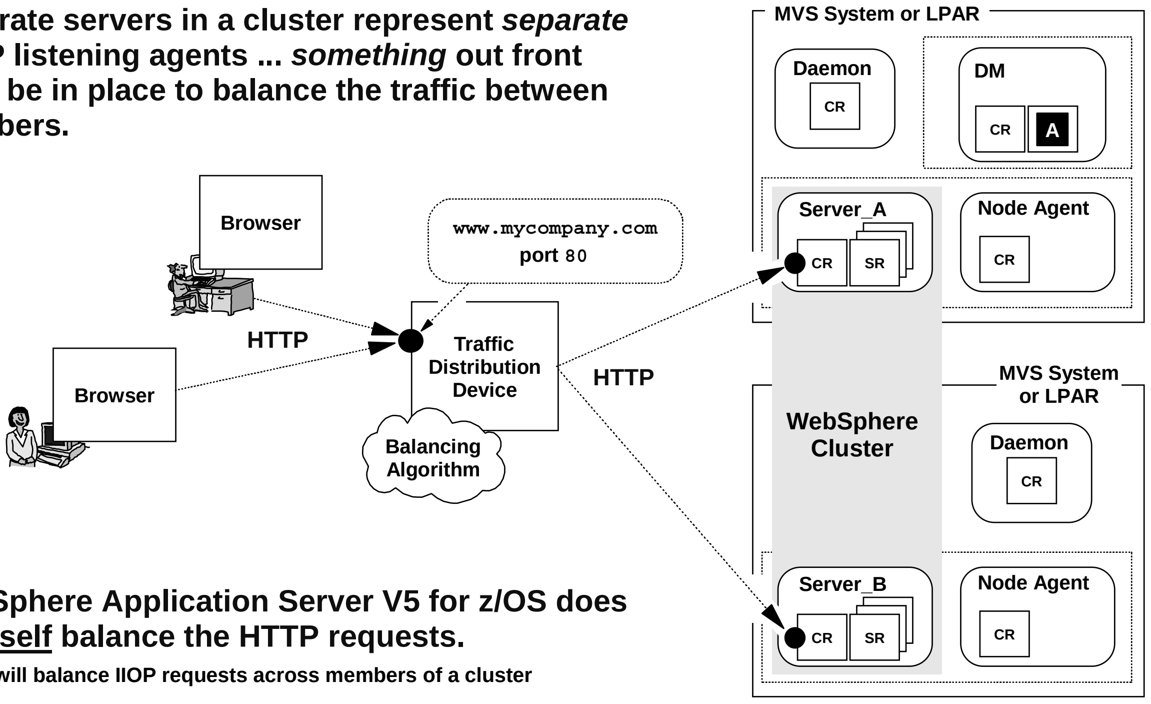
**IBM Americas Advanced Technical Support -- Washington Systems Center**
**Gaithersburg, MD, USA**

Donald C. Bagwell
Certified Consulting I/T Consultant
IBM Washington Systems Center
dbagwell@us.ibm.com
301-240-3016

(This page intentionally left blank)

## HTTP Requests

**Separate servers in a cluster represent *separate* HTTP listening agents ... *something* out front must be in place to balance the traffic between members.**



**WebSphere Application Server V5 for z/OS does not <u>itself</u> balance the HTTP requests.**

**But it will balance IIOP requests across members of a cluster**

**Lots of different solutions to balance traffic.**

**Topic here: "WebSphere HTTP Plugin for z/OS"**

In WebSphere Application Server Version 5 for z/OS, each application server acts as its own HTTP listening agent. The configuration of each server includes the designation of the HTTP (and HTTPS) ports that particular server will listen on. This is true even for servers that are part of a "cluster," WebSphere's mechanism for grouping two or more cloned servers together to form a logical "one."

It's important to understand that WebSphere Application Server itself will not intercede and balance the incoming HTTP request across cluster members. Some other device is needed to do that. Many different such devices exist: Sysplex Distributor, WebSphere Edge Server, various router vendors. For the purposes of this presentation we'll focus on something known as the "WebSphere HTTP Plugin for z/OS."

**Note:** WebSphere Application Server *will* act to intercede and balance IIOP flows. That's a topic unrelated to this presentation.

Let's take a look at a high-level view of the "WebSphere HTTP Plugin for z/OS."

## Agenda

- **Answer some up-front questions about the "WebSphere HTTP Plugin for z/OS"**
- **Briefly discuss what "Session Affinity" is**
- **Show how the "HTTP Plugin" is configured in the HTTP Server**
- **Take a look at the contents of the `plugin-cfg.xml` file**
- **Show how WebSphere Application Server for z/OS Version 5 can automatically generate the `plugin-cfg.xml` file**
- **Review some troubleshooting and problem determination tips**
- **Finish up with a quick illustration of a blended configuration: "HTTP Plugin" + Sysplex Distributor**

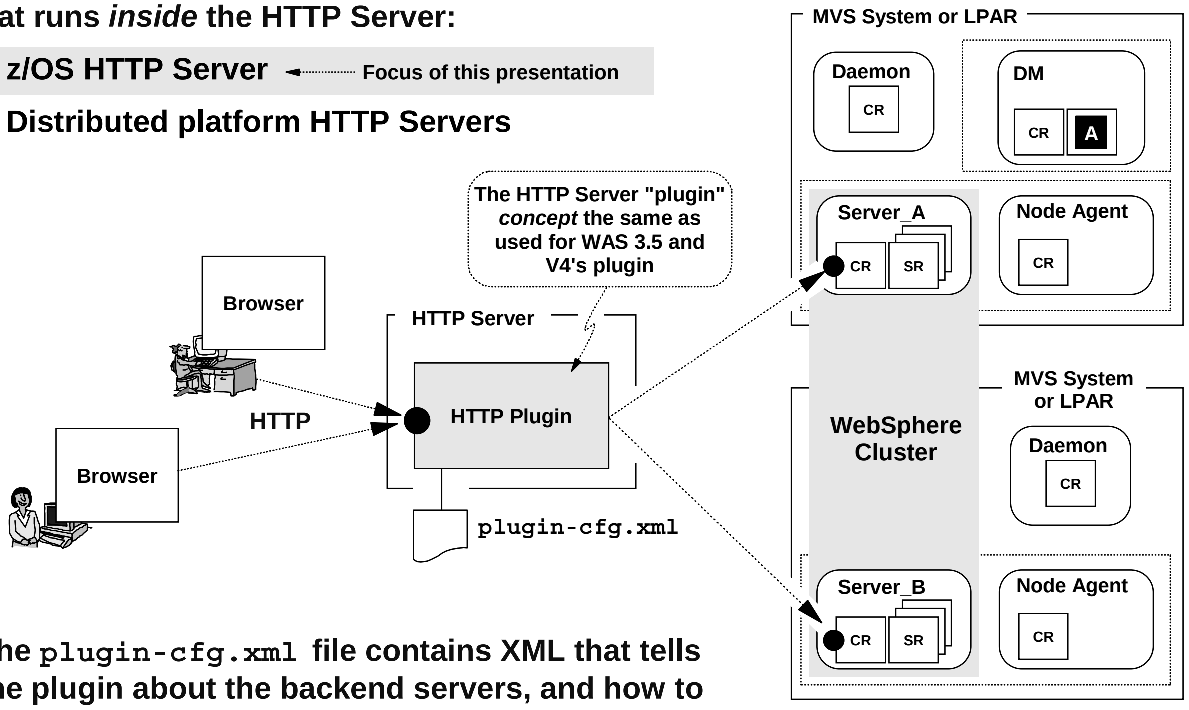
■

This is the agenda we'll cover in this presentation. It's quite a bit of information, so settle in and let's go.

## WebSphere HTTP Plugin for z/OS

**Code provided with WebSphere for z/OS Version 5 that runs *inside* the HTTP Server:**

- **z/OS HTTP Server** ← **Focus of this presentation**
- **Distributed platform HTTP Servers**



**The `plugin-cfg.xml` file contains XML that tells the plugin about the backend servers, and how to route requests to maintain "session affinity"**

The "WebSphere HTTP Plugin for z/OS" -- known hereafter as merely the "Plugin," unless otherwise noted -- is code provided with WebSphere Application Server Version 5 for z/OS. That code runs *inside* the HTTP Server as a "plugin" (hence the name) to the HTTP Server. Code is supplied to run in many different HTTP Servers, including distributed platform HTTP Servers such as Apache. For the sake of this presentation, the focus will be on the HTTP Server that is included with the z/OS operating system.

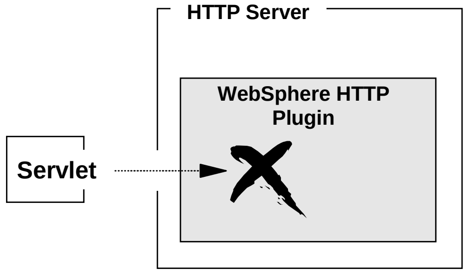
What's interesting is that the *concept* of this new Plugin is very similar to the *concept* of the earlier "WebSphere Application Server for z/OS Version 3.5" product. That too was a "plugin" -- it made use of the API function of the HTTP Server to run inside the web server. The "plugin" that came with WebSphere Application Server for z/OS Version 4 also made use of the HTTP Server's "plugin" API, just as this new Plugin does.

**Caution:** The similarity is limited to the concept of them all being plugins to the HTTP Server. Beyond that, their functions diverge quite a bit. We'll cover that next.

The configuration file for the new Plugin is called the `plugin-cfg.xml` file. We'll spend a lot of time in this presentation covering the contents of that file.

## What the new HTTP Plugin is NOT

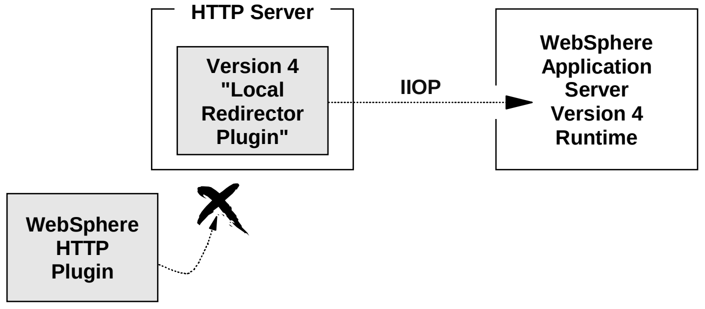
**It is not a servlet execution environment**



**In this sense it is *different* from:**

- **WebSphere Application Server for OS/390 and z/OS Version 3.5**
- **"Local Redirector Plugin" that came with WebSphere Application Server for OS/390 and z/OS Version 4**

**It is not a replacement of the "Local Redirector Plugin" used with WAS V4**


HTTP Server
Version 4 "Local Redirector Plugin"
IIOP
WebSphere Application Server Version 4 Runtime
WebSphere HTTP Plugin

**WebSphere HTTP Plugin flows HTTP, not IIOP.**

■

We offered a caution on the previous page that the similarity between this new "HTTP Plugin" and the older WAS V3.5 plugin or the WAS V4 plugin ended at the fact that all were plugins to the HTTP Server. Let's now cover what the "WebSphere HTTP Plugin for z/OS" is *not*:

- It is **not** an environment in which a servlet can be run. We say that to draw a distinction between it and the WAS V3.5 and WAS V4 plugins. The original WebSphere for z/OS Version 3.5 product was intended to be a servlet environment. The WAS V4 plugin used the V3.5 code base so the servlet environment was still present there as well. But the "WebSphere HTTP Plugin for z/OS" has no ability at all to run servlets *inside the plugin itself*.

  **Note:** It is, however, perfectly capable of *passing through* a request to run a servlet, where the servlet executes in the WebSphere Application Server for z/OS Version 5 runtime environment.

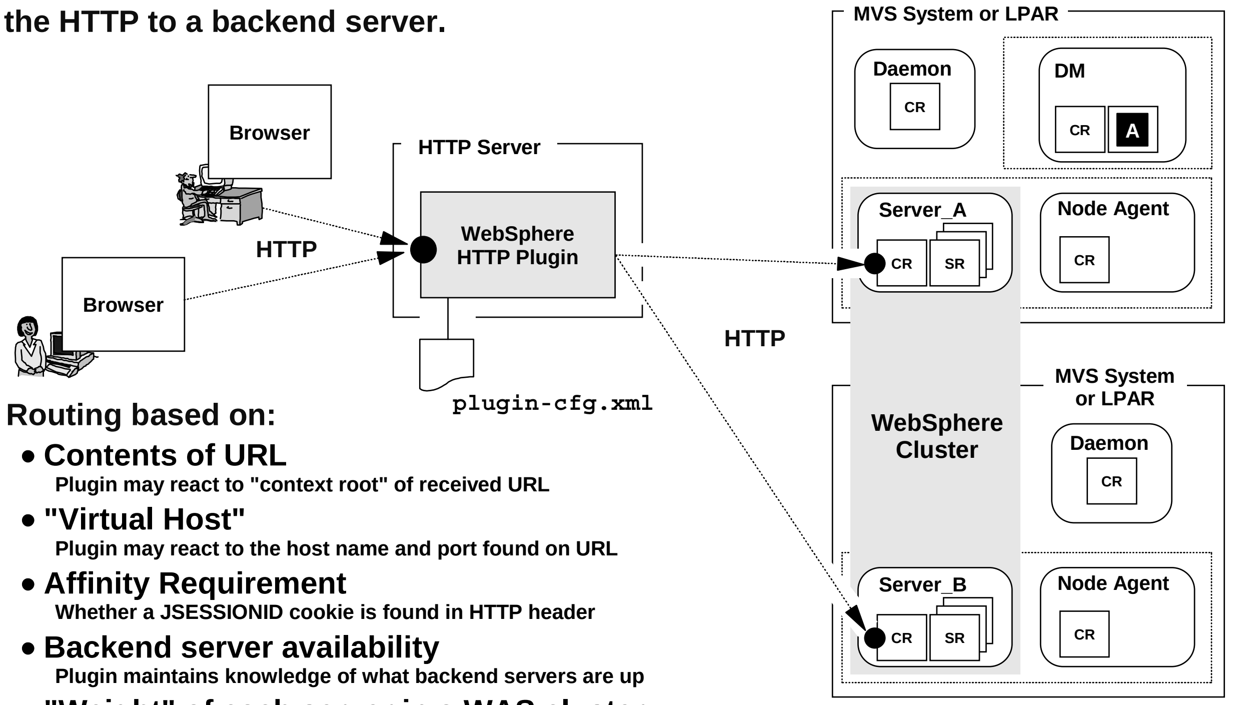
- The new "WebSphere HTTP Plugin for z/OS" is **not** a replacement for the Websphere for z/OS Version 4's "Local Redirector Plugin." The "Local Redirector Plugin" had a very special purpose: it was provided to act as a front-end HTTP listening device at a time when the WAS V4 runtime environment had no HTTP listeners. The "Local Redirector Plugin" received HTTP as input and flowed the request in the form of IIOP to the WAS V4 runtime.

  The new "WebSphere HTTP Plugin for z/OS" has no ability to flow IIOP out the back. Therefore, it can not be used as a replacement for the V4 "Local Redirector Plugin" where the backend runtime environment is still the V4 product. No IIOP will flow from this new plugin.

  **Note:** It *can* front-end Version 4 to provide session affinity. That implies HTTP flowing from the Plugin to the "Transport Handlers" of the Version 4 application servers.

## What the WebSphere HTTP Plugin IS

**A device that takes HTTP inbound and re-routes the HTTP to a backend server.**



**Routing based on:**

- **Contents of URL**
  **Plugin may react to "context root" of received URL**
- **"Virtual Host"**
  **Plugin may react to the host name and port found on URL**
- **Affinity Requirement**
  **Whether a JSESSIONID cookie is found in HTTP header**
- **Backend server availability**
  **Plugin maintains knowledge of what backend servers are up**
- **"Weight" of each server in a WAS cluster**
  **If "round-robin" distribution, then distribute based on defined "weight" of the servers in the cluster**

■

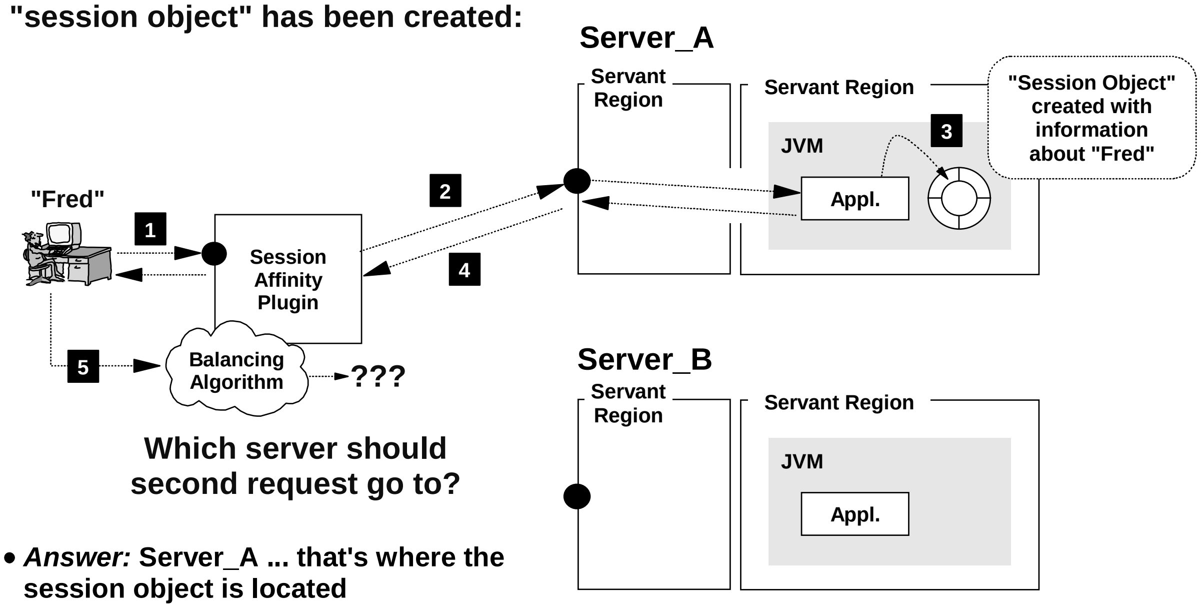
We've just finished explaining what the new "WebSphere HTTP Plugin for z/OS" is not. Let's now look at what it is.

- At its most basic, the new Plugin is a redirector of HTTP based on information found in the `plugin-cfg.xml` file. The objective is to match an inbound HTTP request to a defined backend "Server Cluster," and then route the request to one of the members of that cluster.
- The routing is based on a number of factors, as shown in the chart above:
  - Based on a pattern match against the contents of the URL. Often, this is based on the "context root" of the URL.
  - Based on a match against the host value found on the URL. This is known as matching the "virtual host."
  - A combination of the URL and virtual host
  - Once a Server Cluster has been selected, the Plugin will route to a particular server member based on the contents of the HTTP header; specifically a special "affinity cookie" known as JSESSIONID.
  - The Plugin is able to determine if a backend server is up or not, and if not, then avoid routing the request to that server.
  - In the event no session affinity is necessary, the Plugin will "round-robin" the requests to the servers in the cluster, and "weights" are used to balance the flows in proportion to the weights.

There's a lot that needs to be explained. Let's start with what "Session Affinity" is.

## What is "Session Affinity?"

**"Session Affinity" is the routing of requests back to the server in which a client's "session object" has been created:**



**Which server should second request go to?**

- ***Answer:*** **Server_A ... that's where the session object is located**
- ***Question:*** **How does Plugin know which server to route second request to?**
- ***Answer:*** **based on information put in HTTP header ... the "Unique ID"**

**Each server in WebSphere is given a "Unique ID" ...**

▪

To start the discussion of "Session Affinity," it's important to describe was a "Session Object" is. Application developers who design web applications have the opportunity to capture information about a user and hold it in a Java object known as a "Session Object." The information they hold in there is up to the developer ... it can be as simple as the name of the user, or it may have all sorts of different information. Session objects by default are held in the JVM's memory.

The key to "Session Affinity" is understanding that once a "Session Object" is created, it's important to make certain that all subsequent requests returned by that user flow back to the server in which the session object resides. This is particularly critical when an application that uses session objects resides in "cluster" where two or more servers are running the application at the same time.

**Note:** Please see the following white paper for a more in-depth overview of "session objects" and "session affinity":

```
http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP100316
```

That paper was written for WebSphere V4.01, but the concepts illustrated in the first section are the same. The methods of configuring session management in Version 5 are different now.

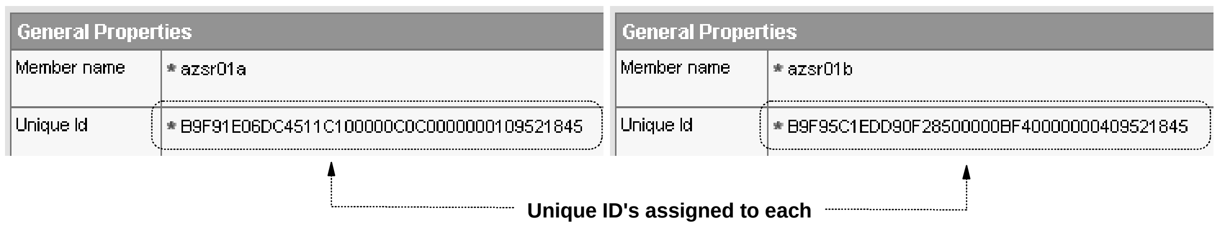
Also, there are ways to "persist" session objects -- store them in DB2, for example -- and thus not make it so critical to maintain session affinity. However, affinity is generally preferred for performance reasons. It's always better to fetch something from memory rather than a data store.

In the picture above, the initial session object for "Fred" was created in "Server_A." That means that when Fred returns the second request, the flow should go back to "Server_A." How does the Plugin know to send it back to "Server_A?" By interogating the HTTP header for something known as a "Unique ID," placed in the header by WebSphere. Each server WebSphere has its own Unique ID.

## Cluster Members and "Unique ID"

**WebSphere assigns each server a "Unique ID", which can be seen in Admin Console:**

**Server:** `azsr01a` **Server:** `azsr01b`



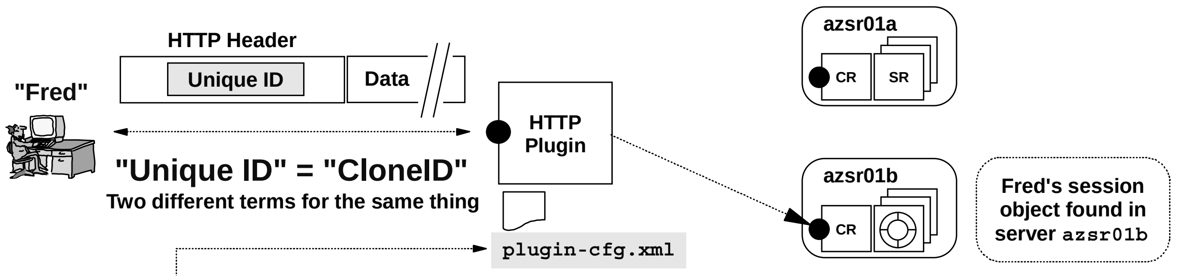
General Properties

| | |
|---|---|
| Member name | * azsr01a |
| Unique Id | * B9F91E06DC4511C100000C0C0000000109521845 |

General Properties

| | |
|---|---|
| Member name | * azsr01b |
| Unique Id | * B9F95C1EDD90F28500000BF40000000409521845 |

**WebSphere will put "Unique ID" into HTTP header after a session object is created:**



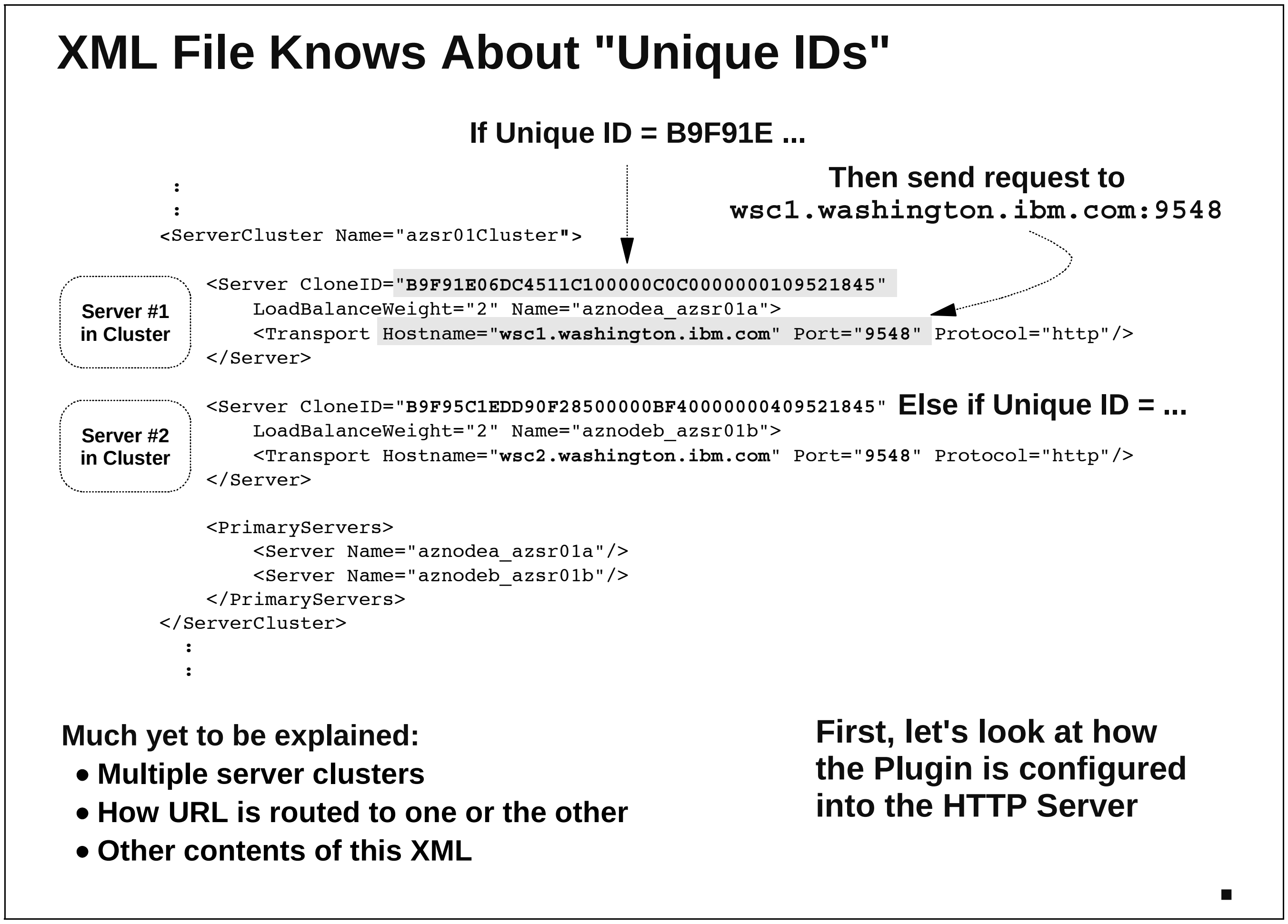
**File `plugin-cfg.xml` has information about where to route based on CloneID...**

All servers in WebSphere Application Server Version 5 for z/OS are given a "Unique ID." The Unique ID for any given server can be viewed by navigating to the "General Properties" for that server in the Admin Console. The Unique ID isn't only for clustered servers; it applies to all servers. But it's use becomes particularly important when dealing with the Plugin and session affinity within a cluster.

Whenever an application creates a session object, the Websphere "session manager" will place in the HTTP header the Unique ID of the server in which the object resides. The user's browser will return that Unique ID in requests that flow back to the site.

It turns out that the phrase "Unique ID" is the same thing as "CloneID." Understanding this is important to understanding how the Unique ID placed in the HTTP header is correlated to a particular cluster member when the Plugin routes requests back to where the session object is created.

The file plugin-cfg.xml -- the configuration file for the Plugin -- is where the Unique ID (the CloneID) is specified and where it's correlated to a particular server. We'll take a very high-level look at this next.



## XML File Knows About "Unique IDs"

**If Unique ID = B9F91E ...**

**Then send request to**
**wsc1.washington.ibm.com:9548**

```
  :
  :
<ServerCluster Name="azsr01Cluster">

    <Server CloneID="B9F91E06DC4511C100000C0C0000000109521845"
        LoadBalanceWeight="2" Name="aznodea_azsr01a">
        <Transport Hostname="wsc1.washington.ibm.com" Port="9548" Protocol="http"/>
    </Server>

    <Server CloneID="B9F95C1EDD90F28500000BF40000000409521845"
        LoadBalanceWeight="2" Name="aznodeb_azsr01b">
        <Transport Hostname="wsc2.washington.ibm.com" Port="9548" Protocol="http"/>
    </Server>

    <PrimaryServers>
        <Server Name="aznodea_azsr01a"/>
        <Server Name="aznodeb_azsr01b"/>
    </PrimaryServers>
</ServerCluster>
  :
  :
```

Server #1 in Cluster
Server #2 in Cluster
Else if Unique ID = ...

**Much yet to be explained:**

- **Multiple server clusters**
- **How URL is routed to one or the other**
- **Other contents of this XML**

**First, let's look at how the Plugin is configured into the HTTP Server**

■

Here's a peek at the contents of the XML file used by the Plugin. What we're showing here is one section of the XML file, the `<ServerCluster>` section for one cluster. The <ServerCluster> section is used to provide information about all the servers in a cluster, and define such necessary routing information as the Unique ID of the server, and the host name and port to where the requests are to be sent.

**Note:** There's a very important first step to this process that's not shown on this chart. Before a request can be handled within a `<ServerCluster>` section, it needs first to be routed to it. That's done with something called the `<Route>` statement. We'll cover that in a bit.

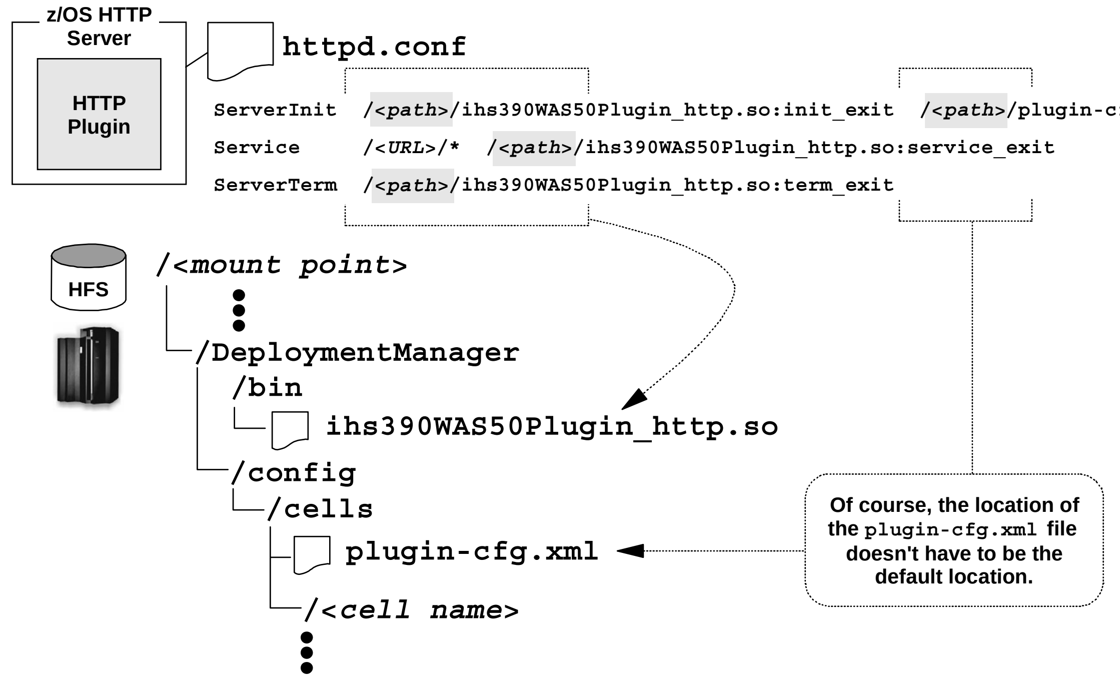
Assume that a request is mapped to the `<ServerCluster>` shown in this chart. The Plugin then goes through some logic that looks like this:

- It sees if the HTTP header has the "Unique ID" (or CloneID) in the HTTP header. If not, then the Plugin will round-robin between the servers in the cluster, based on the `LoadBalanceWeight` values defined.
- However, if a "Unique ID" is present in the HTTP header, that means session affinity routing is necessary. The Plugin then matches the Unique ID found in the header against the `CloneID=` values found in the XML. If a match is found, the request is then mapped to the `<Transport>` defined.

All that said, there's still quit a bit left to explain. To start the process, we'll first go to a discussion of how the Plugin is configured into the HTTP Server.

## How the HTTP Plugin is Configured

**Very similar to how WebSphere for z/OS V3.5 was configured, and similar to how the V4 "Local Redirector Plugin" was configured:**



```
httpd.conf

ServerInit  /<path>/ihs390WAS50Plugin_http.so:init_exit  /<path>/plugin-cfg.xml
Service     /<URL>/*  /<path>/ihs390WAS50Plugin_http.so:service_exit
ServerTerm  /<path>/ihs390WAS50Plugin_http.so:term_exit
```

```
/<mount point>
   :
  /DeploymentManager
    /bin
      ihs390WAS50Plugin_http.so
    /config
      /cells
        plugin-cfg.xml
        /<cell name>
          :
```

z/OS HTTP Server
HTTP Plugin
HFS
Of course, the location of the plugin-cfg.xml file doesn't have to be the default location.

The "WebSphere HTTP Plugin for z/OS" configures into the HTTP Server in the same manner in which the older WAS V3.5 and WAS V4 plugins did. Three statements are needed in the HTTP Server's `httpd.conf` file:

**`ServerInit`** -- This statement is used to tell the HTTP Server about the Plugin's executable module, and the configuration XML file to be used by the plugin. You will have *one* instance of this statement. The Plugin code itself is:

```
ihs390WAS50Plugin_http.so
```

and is located in the `/DeploymentManager/bin` directory under the "Config root" of your cell, indicated with `<path>` in the chart.

The "exit" that is specified on the module is critical. For the `ServerInit` statement the value *must* be **`:`**`init_exit`.

Finally, the XML file to be used is specified as a parameter immediately following the **`:`**`init_exit`, separated by a blank space. The default location is:

```
/<config root>/DeploymentManager/config/cells/plugin-cfg.xml
```

but may in fact be any location or file name you wish.

**Note:** One advantage to having the pointer be to the default location and file name is that WebSphere has a facility to generate the XML file for you. It'll place the generated XML file at the default location. The downside, of course, is that if you've made hand-changes to the file, the automatic generation process will overlay the file. So if you're planning on making hand-changes, it's better to copy the generated file to a different location and point to your separate, customized copy.

**`Service`** -- This statement is used to "catch" received URLs and pass them into the Plugin for processing. You will have as many `Service` statements as you have different URLs you wish the Plugin to handle.

The format of the statement has four parts:

- The keyword `Service`
- A URL mask used to "catch" a URL and apply it to the Plugin. For example, a value of `/MyIVT/*` would "catch" all URLs that have `/MyIVT/` as the first part of the URL after the host value. If you have another URL -- say, `/YourIVT/` -- that you wanted to also process in the Plugin, you would simply code a second `Service` statement for that URL. You may code as many `Service` statements as you need.
- The full directory and file name of the `ihs390WAS50Plugin_http.so` module (which should be identical to the directory and file name specified on the `ServerInit` statement, but be careful of the "exit" value, explained next)
- The "exit" value of `service_exit`. It is critical that this value be used, and not the exit values for the `ServerInit` or `ServerTerm` statements.

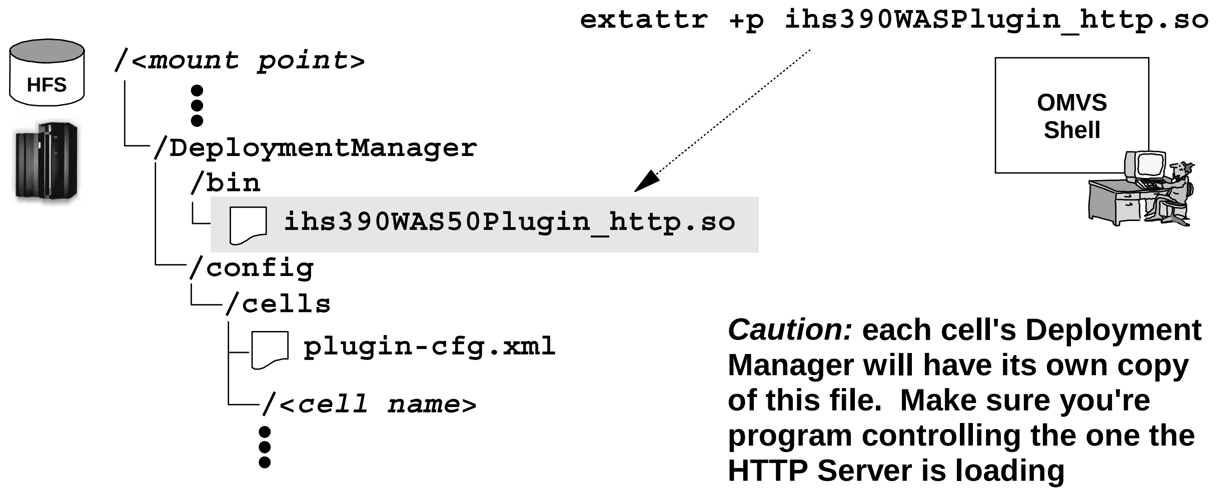
**`ServerTerm`** -- This statement is used to gracefully stop the Plugin when the HTTP Server itself is stopped. There will be only one `ServerTerm` statement. The format is fairly simple:

- The keyword `ServerTerm`
- The full directory and file name of the `ihs390WAS50Plugin_http.so` module (which should be identical to the directory and file name specified on the `ServerInit` statement and the `Service` statements, but be careful of the "exit" value, explained next)
- The "exit" value of `term_exit`. It is critical that this value be used, and not the exit values for the `ServerInit` or `Service` statements.

The next step is to "program control" the module.

## Program Control ".so" File

**Just like with WAS V3.5, the HTTP Plugin code has to be program controlled to load into HTTP Server and operate properly:**



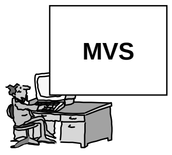
***Caution:*** **each cell's Deployment Manager will have its own copy of this file. Make sure you're program controlling the one the HTTP Server is loading**

**Starting the HTTP Server with this new Plugin is just like in the past ...**

When you create your Deployment Manager, the `ihs390WAS50Plugin_http.so` file will be copied into the directory structure of the DMGR, under the "Config Root." That's not a symbolic link to the actual file located somewhere else; that's an actual copy of the file. Two points on that:

1. The file needs to be program controlled before it can be used. The command to program control the file is shown in the chart.
2. If you have multiple Deployment Managers configured, each will have a copy of this file. Make sure the `extattr +p` command you enter applies to the copy of the module you intend to use.

## Starting the HTTP Server with Plugin



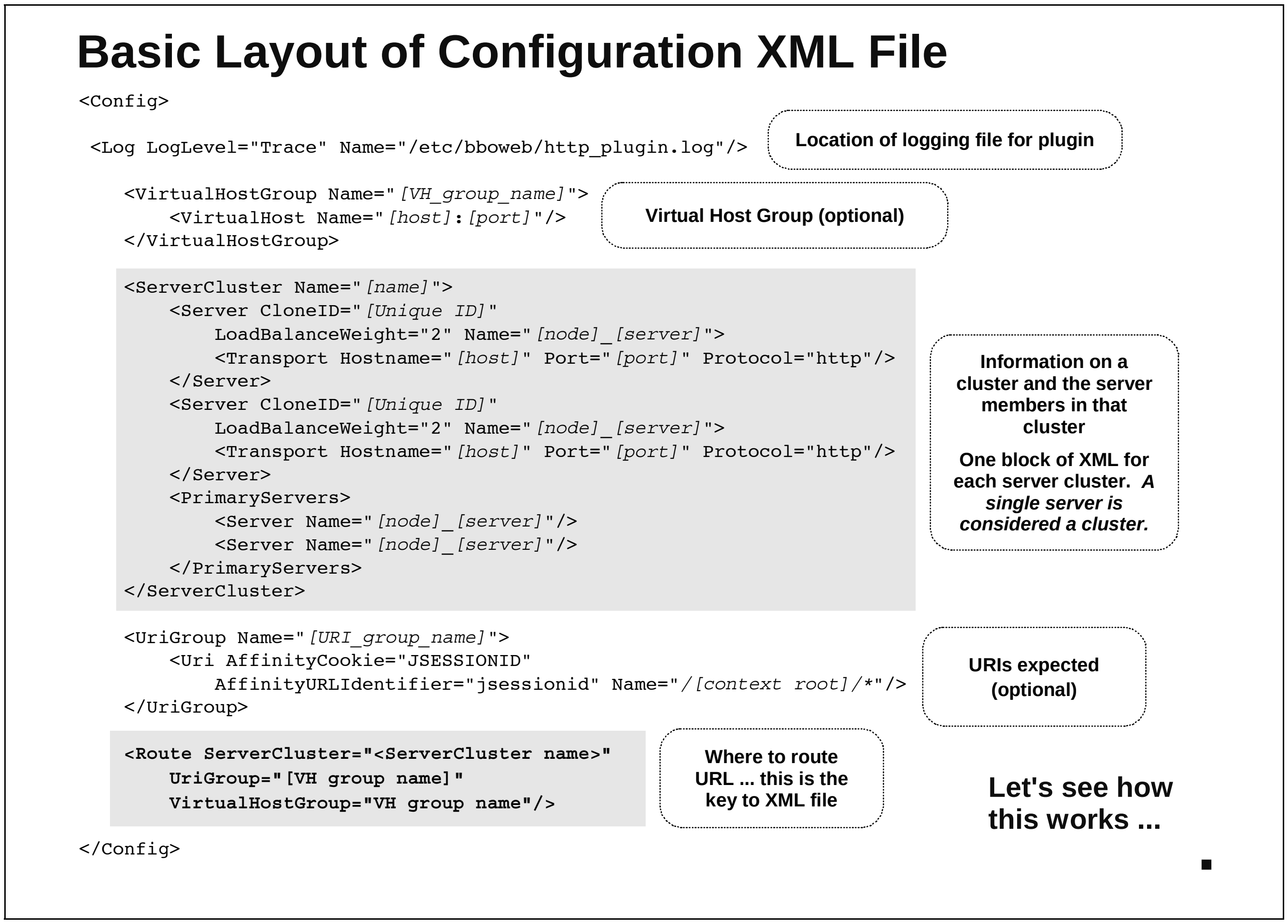
```
S <PROC>
   JESMSGLG JES2              2 BBOWEB   S
   JESJCL   JES2              3 BBOWEB   S
   JESYSMSG JES2              4 BBOWEB   S
   SYSPRINT BBOWEB          101 BBOWEB   O
   SYSOUT   BBOWEB          105 BBOWEB   O
```

```
Licensed Material - Property of IBM
5655-I35 (C) Copyright IBM Corp. 2000, 2003

All Rights Reserved.

U.S. Government users - RESTRICTED RIGHTS - Use, Duplication, or
Disclosure restricted by GSA-ADP schedule contract with IBM Corp.
IBM is a registered trademark of the IBM Corp.

WebSphere HTTP Plug-in for z/OS and OS/390  Version 5.0 Service Level 0.0 is starting

WebSphere HTTP Plug-in for z/OS and OS/390
                        initializing with configuration file : /<path>/plugin-cfg.xml

WebSphere HTTP Plug-in for z/OS and OS/390  initialization went OK :-)
```

**Quite a few things can go wrong ... we'll cover those later. Next let's look at the `plugin-cfg.xml` file, which is the configuration file used by the Plugin**

In almost every way the starting of the HTTP Server and the Plugin is just as it has always been. The indication of successful initialization of the Plugin is found in the `SYSOUT` of the started task, and the now familiar "smiley face" is what tells the positive story.

If we stopped there, we'd fall far short of the total story. There's a whole bunch of things that can go wrong and prevent the smiley face from appearing. What we'll cover next is contents of the XML file itself and how that works, then we'll go through a fairly extensive troubleshooting and debugging section.



## Basic Layout of Configuration XML File

```
<Config>

 <Log LogLevel="Trace" Name="/etc/bboweb/http_plugin.log"/>

    <VirtualHostGroup Name="[VH_group_name]">
        <VirtualHost Name="[host]:[port]"/>
    </VirtualHostGroup>

    <ServerCluster Name="[name]">
        <Server CloneID="[Unique ID]"
            LoadBalanceWeight="2" Name="[node]_[server]">
            <Transport Hostname="[host]" Port="[port]" Protocol="http"/>
        </Server>
        <Server CloneID="[Unique ID]"
            LoadBalanceWeight="2" Name="[node]_[server]">
            <Transport Hostname="[host]" Port="[port]" Protocol="http"/>
        </Server>
        <PrimaryServers>
            <Server Name="[node]_[server]"/>
            <Server Name="[node]_[server]"/>
        </PrimaryServers>
    </ServerCluster>

    <UriGroup Name="[URI_group_name]">
        <Uri AffinityCookie="JSESSIONID"
            AffinityURLIdentifier="jsessionid" Name="/[context root]/*"/>
    </UriGroup>

    <Route ServerCluster="<ServerCluster name>"
        UriGroup="[VH group name]"
        VirtualHostGroup="VH group name"/>

</Config>
```

**Location of logging file for plugin**

**Virtual Host Group (optional)**

**Information on a cluster and the server members in that cluster**

**One block of XML for each server cluster. *A single server is considered a cluster.***

**URIs expected (optional)**

**Where to route URL ... this is the key to XML file**

**Let's see how this works ...**

The XML file initially looks intimidating, but in reality is fairly simple once you get used to it. We'll quickly review the basics here, then go into greater detail in the following pages. There are five basic sections of the file:

1. The `<Log>` section -- This simply points to the log file that will be used, and the level of logging that will be done. We'll cover this later.
2. The `<VirtualHostGroup>` section -- This is listed as "optional." In truth, it's optional only if there's a `<UriGroup>` section. If no `<UriGroup>`, then `<VirtuaHostGroup>` becomes necessary. One or the other must exist. The `<VirtualHostGroup>` section is used to define host values that are used to compare against received URLs to see how the request received is to be handled. Multiple `<VirtualHostGroup>` sections are permitted.
3. The `<UriGroup>` section -- Also listed as "optional" (provided there's a `<VirtualHostGroup>` section), this is used to define a pattern mask that is used to compare against URIs to see how the request received is to be handled. Multiple `<UriGroup>` sections are permitted.
4. The `<Route>` section -- This is the heart of the XML file. The `<Route>` section is used to connect a `<UriGroup>` match (or `<VirtualHostGroup>` match, or both) to a Server Cluster. Ultimately, that's the goal: take an inbound request and route it to a server cluster member. The `<Route>` section is what helps do this. Multiple `<Route>` sections are permitted.
5. The `<ServerCluster>` section -- This section is used to define a server cluster and the server members that make up the cluster. This is where the "Unique ID" of the cluster members are provided, and this is where the host name and port numbers of the individual cluster members are provided. Multiple `<ServerCluster>` sections are permitted.

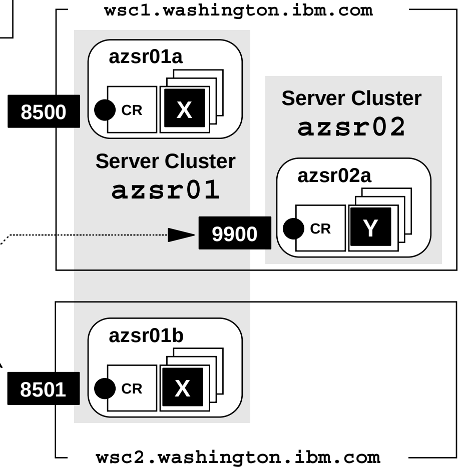
## Multiple ServerClusters in XML

**plugin-cfg.xml**

```
<ServerCluster Name="azsr01">
    <Server CloneID="B9F0..."
        <Transport
          Hostname="wsc1.washington.ibm.com"
          Port="8500"/>
    </Server>
    <Server CloneID="A7FC..."
        <Transport
          Hostname="wsc2.washington.ibm.com"
          Port="8501"/>
    </Server>
</ServerCluster>

<ServerCluster Name="azsr02">
    <Server CloneID="C3FF..."
        <Transport
          Hostname="wsc1.washington.ibm.com"
          Port="9900"/>
    </Server>
</ServerCluster>
```

**Note**

**This is not *exactly* how XML looks ... simplied here to save space on page**



**This illustrates how a single server is still considered part of a "ServerCluster"**

**Applications:**

| | | |
|---|---|---|
| `azsr01` **cluster: "X"** | **Context root:** `/X` |
| `azsr02` **cluster: "Y"** | **Context root:** `/Y` |

**Next let's see how the Plugin knows to route a URL to one ServerCluster versus another ...**

Let's first focus on the <ServerCluster> sections. As stated before, these are used to define a WebSphere cluster and the cluster members that make up the cluster. The best way to explain this is with an example. The picture on the right side of the chart illustrates a complex with two MVS images and two Server Clusters. Cluster `azsr01` has two members and spans the `wsc1` and `wsc2` MVS images, while cluster `azsr02` has only one member and reside on only the `wsc1` system. The ports each cluster member listens on is shown in the chart.

**Note:** The examples shown hereafter are loosely based on the `WP100367` white paper configuration found at `www.ibm.com/support/techdocs`. Some things have been removed from the example so the pictures are cleaner, but overall it's very close to the configuration illustrated in `WP100367`.

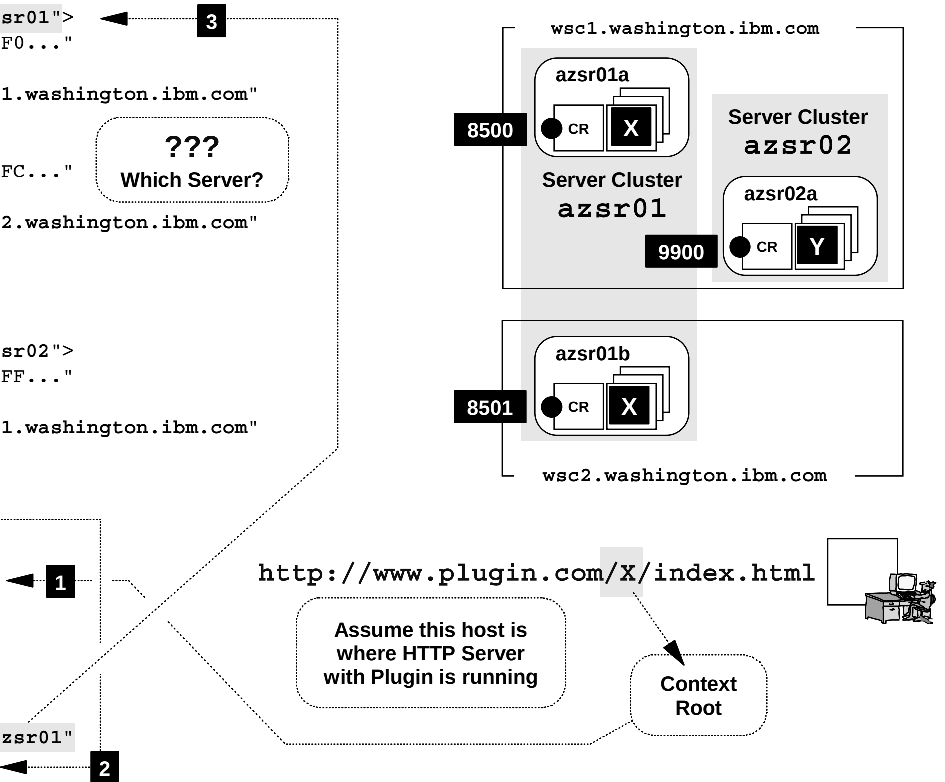
The XML file will have two `<ServerCluster>` sections, one for each cluster found in the actual runtime environment.

**Note:** A single server is considered to be a member of a cluster. The white paper WP100367 reinforces this and shows why it's important to keep this consideration in mind, particularly as it relates to naming conventions.

Notice how each `<ServerCluster>` section has one or more `<Server>` sections. The `<Server>` sections define the individual serves in the cluster. Notice further how each `<Server>` section provides the `CloneID` value, the `Hostname=` and `Port=` for the server. It is that information that provides what's necessary to take a URL and route it to the actual application server that will service the request. Now imagine that server cluster `azsr01` has application `/X` and `azsr02` has application `/Y`. What would map an inbound URL to one cluster or another? That's covered next.

## URIGroups and the `<Route>` Block

```
<ServerCluster Name="azsr01">
    <Server CloneID="B9F0..."
        <Transport
          Hostname="wsc1.washington.ibm.com"
          Port="8500"/>
    </Server>
    <Server CloneID="A7FC..."
        <Transport
          Hostname="wsc2.washington.ibm.com"
          Port="8501"/>
    </Server>
</ServerCluster>

<ServerCluster Name="azsr02">
    <Server CloneID="C3FF..."
        <Transport
          Hostname="wsc1.washington.ibm.com"
          Port="9900"/>
    </Server>
</ServerCluster>

<UriGroup Name="ApplX">
    <Uri Name="/X/*"/>
</UriGroup>
<UriGroup Name="ApplY">
    <Uri Name="/Y/*"/>
</UriGroup>

<Route ServerCluster="azsr01"
    UriGroup="ApplX"/>
<Route ServerCluster="azsr02"
    UriGroup="ApplY"/>
```



**Next: routing between servers in a Cluster ...**

Let's take an example of a mapping a URL to a cluster using a match on the URI contents. Let's use as an example a URI of `/X/index.html`. The `plugin-cfg.xml` file is the one shown in the chart (some of the XML is not shown; this was done to save space on the chart).

1. The URL is matched against the contents of the XML, specifically the `<UriGroup>` information. In this configuration file there are *two* `<UriGroup>` sections. One has a `Uri Name=` value of `/X/*` and the other has a `Uri Name=` value of `/Y/*`. The two <UriGroup> sections are differentiated with a `Name=` value of `"ApplX"` and `"ApplY"` respectively.

   **Note:** There could be a hundred different `<UriGroup>` sections; there's no limit on how many. Also, any given `<UriGroup>` section might have multiple `Uri Name=` values specified (we cover that later). Finally, in this example we're showing what happens when *only* `<UriGroup>` (and not `<VirtualHostGroup>` ) is present. We'll show later what happens when both are present.

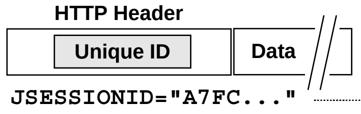
   The URL itself has a value of `/X/index.html`, so it matches against the `Uri Name="/X/*"` found in the `"ApplX"` section. With that in hand, the Plugin now goes in search of a `<Route>` block that references the "ApplX" Uri Group.

2. It finds the `<Route>` section with `UriGroup="ApplX"`. This <Route> section has a value of `ServerCluster="azsr01"` specified on it. This is the pointer to the `<ServerCluster>` block of XML that is to be used for this request. The next step is to find the `<ServerCluster>` block of XML.

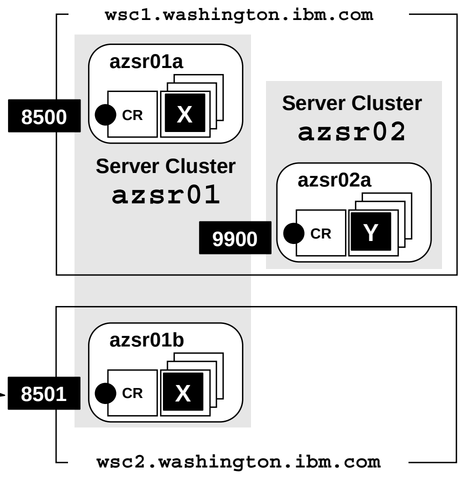
3. Sure enough, there's a `<ServerCluster>` block with `Name="azsr01"`. The incoming request has been successfully associated with a cluster. But that cluster has two different servers in it. So how does the Plugin know which server to send the request to? That's next.

## Affinity or Round-Robin Routing

**We saw how `<UriGroup>` and `<Route>` worked together to get URL to Server*Cluster*. Server it goes to depends on if HTTP Header has "AffinityCookie"**



```
<ServerCluster Name="azsr01">
    <Server CloneID="B9F0..."
        <Transport
          LoadBalanceWeight="2"
          Hostname="wsc1.washington.ibm.com"
          Port="8500"/>
    </Server>
    <Server CloneID="A7FC..."
        <Transport
          LoadBalanceWeight="2"
          Hostname="wsc2.washington.ibm.com"
          Port="8501"/>
    </Server>
</ServerCluster>

<UriGroup Name="ApplX">
    <Uri AffinityCookie="JSESSIONID"
     Name="/X/*"/>

<Route ServerCluster="azsr01"
    UriGroup="ApplX"/>
```



**WebSphere "Session Manager" places JESSIONID cookie into HTTP Header when (and if) session object created**

**If no JSESSIONID in HTTP Header, then round-robin based on "LoadBalanceWeight"**

The previous chart showed how a URL was interogated to see how to map it to a `<Route>` block, and how the `<Route>` block was mapped to a `<ServerCluster>` block. The cluster had two servers in it, so the question is: how does the Plugin know which server to send the reqeust to?

It depends on whether the HTTP header has an "affinity token" inside it. This "affinity token" may or may not be present, depending on whether the application to which the user is connected makes use of session objects, and whether a session object has yet been created. If no session object has been created, then no "affinity token" will be placed in the HTTP header by WebSphere. *If the token is present, it'll be the "Unique ID" of the WebSphere server in which the session object was created.*

In the `<ServerCluster Name="azsr01">` block of XML there is defined two `<Server>` blocks. Each has associated with it the `CloneID` (equal to the "Unique ID") of the server. The Plugin will compare the `UniqueID` found in the HTTP header (if one exists) with the `CloneID=` value found in the XML. If it finds a match, it'll route the request to the `<Server>` defined with that `CloneID=` value.

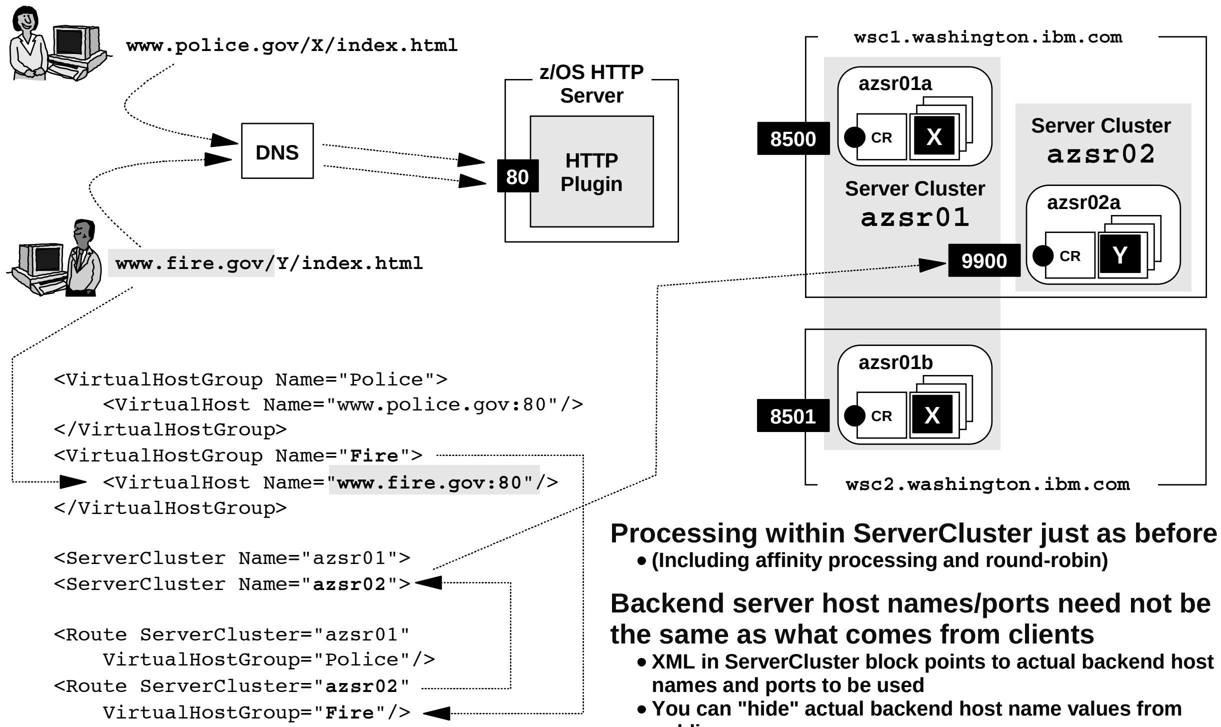
If the received request has no Unique ID in the HTTP header, then the Plugin will simply round-robin the request between the servers defined in the `<ServerCluster>.` The distribution will be based on the `LoadBalanceWeight=` values defined in the XML>.

**Note:** Under what conditions would a URL *not* have a Unique ID in the HTTP header?
- When the application doesn't make use of session objects
- The very first request to an application that does use session objects
- Any time during connectivity with an application but before a session object is created
- After the application destroys the session object

Now let's look at matching on the "virtual host."

## Virtual Hosts and Routing to Cluster

**In addition to routing based on Context Root, you may also route based on host value found on URL:**



```
<VirtualHostGroup Name="Police">
    <VirtualHost Name="www.police.gov:80"/>
</VirtualHostGroup>
<VirtualHostGroup Name="Fire">
    <VirtualHost Name="www.fire.gov:80"/>
</VirtualHostGroup>

<ServerCluster Name="azsr01">
<ServerCluster Name="azsr02">

<Route ServerCluster="azsr01"
    VirtualHostGroup="Police"/>
<Route ServerCluster="azsr02"
    VirtualHostGroup="Fire"/>
```

**Processing within ServerCluster just as before**
- **(Including affinity processing and round-robin)**

**Backend server host names/ports need not be the same as what comes from clients**
- **XML in ServerCluster block points to actual backend host names and ports to be used**
- **You can "hide" actual backend host name values from public**

To start this discussion, let's first review what a "virtual host" is and why that function is in the product. Virtual hosts in WebSphere go way back ... Version 3.5 at least. The purpose of the function was to provide a way to logically isolate applications from one another, based on the host name found on the URL.

Users out in the world who point their browsers at a host like `www.police.gov` have that address resolved to an actual IP address (the familiar dotted decimal address) by the Domain Name Service (DNS). It turns out the DNS will support the resolving of many different host name to the same address. So both `www.police.gov` and `www.fire.gov` could be configured to be the *same* HTTP server.

Suppose in this example that the police department's application ("X") and the fire department's application ("Y") are hosted in the same WebSphere cell, but in different server clusters. The desire is to make sure that only those people with `www.police.gov` are able to run the "X" application, and only people with `www.fire.gov` are able to run the "Y" application. Nobody with `www.police.gov` can run the "Y" application, or `www.fire.gov` the "X" application.

How to keep them separate? By coding a `<VirtualHostGroup>` definition in the XML and referencing that virtual host in the `<Route>` definition. Consider the example above: two users, each going to their respective hosts. The inbound URL is, let's say, `www.fire.gov/Y/index.html`. Here's what happens:

- The Plugin looks to see if it can locate a match between the host name found on the URL and a `VirtualHost Name=` definition in a `<VirtualHostGroup>` .

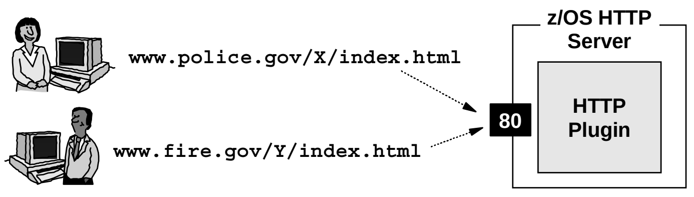
> **Note:** On the previous chart it was suggested that the Plugin looks first at a match on `<UriGroup>`. In truth, it checks `<VirtualHostGroup>` first, then `<UriGroup>`. In the example we're showing here it's not relevant what the search order is: the XML so far has had either `<UriGroup>` or `<VirtualHostGroup>` but not both ... yet: we'll show an example of that in a bit.

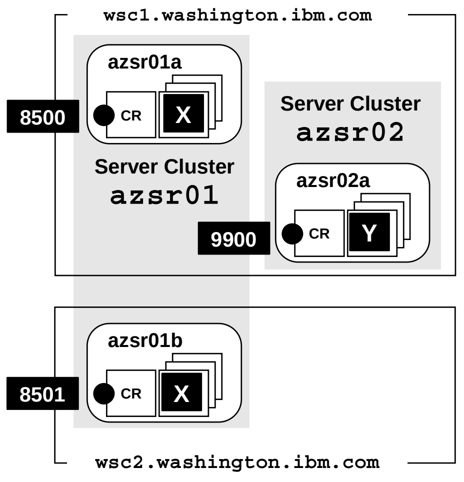
- Finding a hit with the `<VirtualHostGroup>` with `VirtualHostName="www.fire.gov:80"`, it stores the `Name=` value for that `<VirtualHostGroup>` (in this case, `Name="Fire"`), and then it goes in search of a `<Route>` block with the a reference to `<VirtualHostGroup>` named "Fire".
- There is in fact a `<Route>` block with `VirtualHostName="Fire"` ... it's the `<Route>` block with `<ServerCluster Name="azsr02">` specified on it.
- The Plugin then goes to the `<ServerCluster>` definition and then routes to the cluster member in the same way it did in the previous example with the `<UriGroup>` processing.

There's a subtle but important thing that went on here ... the virtual host matching is based on what's on the URL used to get the request to the Plugin, not *necessarily* the host of the backend server where the application actually resides. For example, the actual application to which the request was routed might have been something like `prod3.applhost.com`. That value -- `prod3.applhost.com` -- would have been defined in the `<Server>` block so the Plugin could get the request back to the server. But the user supplied a different host -- `www.fire.gov` -- on the URL. That host resolved to the IP address of the HTTP Server, and the virtual host matching was done on that host, not the host name of the backend server where the application resides.

> **Important:** The application installed in the backend application server would still have to be bound to a virtual host alias. The key there is to have it bound to a virtual host alias that'll match the definition found in the `<Server>` definition of the `<ServerCluster>` block of XML. For example, let's say the client's URL specified `www.fire.gov`, but the `<Server>` definition mapped that to `prod3.applhost.com` listening on port `9900`. The virtual host alias to which the application is bound would be either `*.9900` or `prod3.applhost.com:9900`, but *not* `www.fire.gov:80`

## Combination: URI *and* Virtual Host





```
<VirtualHostGroup Name="Police">
    <VirtualHost Name="www.police.gov:80"/>
</VirtualHostGroup>
<VirtualHostGroup Name="Fire">
    <VirtualHost Name="www.fire.gov:80"/>
</VirtualHostGroup>

<ServerCluster Name="azsr01">

<ServerCluster Name="azsr02">

<UriGroup Name="ApplX">
    <Uri Name="/X/*"/>
</UriGroup>
<UriGroup Name="ApplY">
    <Uri Name="/Y/*"/>
</UriGroup>

<Route ServerCluster="azsr01"
    UriGroup="ApplX"
    VirtualHostGroup="Police"/>
<Route ServerCluster="azsr02"
    UriGroup="ApplY"
    VirtualHostGroup="Fire"/>
```

**Only requests with *both* `www.police.gov` and context root of `/X` will get routed to `azsr01` cluster**

- **If *only* `UriGroup` on `Route`, then *all* requests with that context root get routed there, regardless of host name on URL**
- **If *only* `VirtualHostGroup` on `Route`, then *all* requests for that host get routed there, regardless of context root value**

**One more variation on this, then we'll get to troubleshooting**

You knew it was coming, and here it is ... both <`UriGroup`> and <`VirtualHostGroup`>! It is possible -- and indeed this the default -- to code both on a <`Route`> definition so the mapping to the server cluster. This means that before a URL is mapped to a backend server, it has to pass *two* tests: does the virtual host match and does the URI match?

The coding in the <`Route`> statement is fairly straight-forward: both a `UriGroup`= and `VirtualHostGroup`= value is provided, as shown in the chart.

**Note:** This then requires that the referenced <`UriGroup`> and referenced <`VirtualHostGroup`> be present in the XML file. If they're not there -- that is, if they're referenced in the <`Route`> but not actually present elsewhere in the XML -- then the Plugin won't initialize. More on initialization failures later.

This brings up an interesting set of questions: when both are <u>not</u> coded, what's allowed to be routed to the server cluster?

- If only <`UriGroup`> is found on the <`Route`> statement, then all URL's with the URI values in the referenced <`UriGroup`> will be mapped to the <`ServerCluster`>, regardless of the virtual host value. No virtual host checking is done.
- Conversely, if only <`VirtualHostGroup`> is found on the <`Route`> statement, then all URLs that match the virtual host values defined in the referenced <`VirtualHostGroup`> will be mapped to the <`ServerCluster`>, regardless of the URI value.

It's only by coding both that you get the greater degree of granularity.

One more variation on this, then on to the troubleshooting section.

## Multiple Context Roots per UriGroup

```
<VirtualHostGroup Name="Police">
    <VirtualHost Name="www.police.gov:80"/>
</VirtualHostGroup>
<VirtualHostGroup Name="Fire">
    <VirtualHost Name="www.fire.gov:80"/>
</VirtualHostGroup>

<ServerCluster Name="azsr01">

<ServerCluster Name="azsr02">

<UriGroup Name="ApplX">
    <Uri Name="/X/*"/>
    <Uri Name="/A/*"/>
    <Uri Name="/B/*"/>
    <Uri Name="/C/*"/>
      :
</UriGroup>

<UriGroup Name="ApplY">
    <Uri Name="/Y/*"/>
</UriGroup>

<Route ServerCluster="azsr01"
    UriGroup="ApplX" ◄
    VirtualHostGroup="Police"/>
<Route ServerCluster="azsr02"
    UriGroup="ApplY"
    VirtualHostGroup="Fire"/>
```

**You may code multiple URIs in the `<UriGroup>` block of XML**

- **Many different context roots will get routed to ServerCluster `azsr01`**

**In this example `<Route>` has `VirtualHostGroup` as well.**

- **All those URLs must have host of `www.police.gov`**

**Yes, multiple `VirtualHost` names permitted per `VirtualHostGroup`**

**Lots of permutations to this**

**There's an opportunity to introduce ambiguity into the XML. You should be careful to avoid this ...**

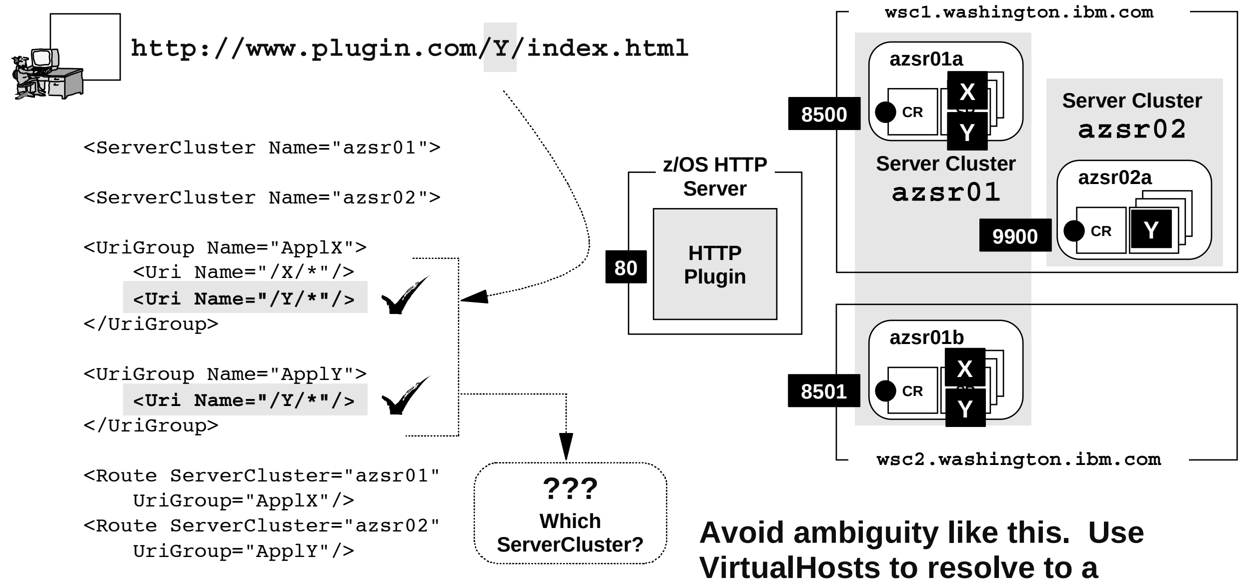
■

You may have already picked this up ... the <`UriGroup`> may accept any number of <`UriName=...`> values. This is how you would map a lot of different applications to the same server cluster. In this example, four different application context roots would all map to `azsr01`.

Multiple <`VirtualHost Name=...`> values can be added to a <`VirtualHostGroup`> as well. Same concept as with <`UriGroup`>.

There are a lot of different ways in which you can combine these things to map requests to server clusters. There's also an opportunity to introduce some ambiguity, and you should avoid doing this. The next chart covers such a case.

## Avoid Ambiguity



```
<ServerCluster Name="azsr01">

<ServerCluster Name="azsr02">

<UriGroup Name="ApplX">
    <Uri Name="/X/*"/>
    <Uri Name="/Y/*"/>
</UriGroup>

<UriGroup Name="ApplY">
    <Uri Name="/Y/*"/>
</UriGroup>

<Route ServerCluster="azsr01"
    UriGroup="ApplX"/>
<Route ServerCluster="azsr02"
    UriGroup="ApplY"/>
```

**Avoid ambiguity like this. Use VirtualHosts to resolve to a single ServerCluster.**

**Or don't code second ServerCluster's `<Uri Name="/Y/*">` in XML**

**It appears the last `<Route>` statement in XML that matches is the one that applies ... but my testing wasn't that exhaustive. Other rules may apply.**

**Next: Where does `plugin-cfg.xml` file come from initially?**

Just like there was in Version 4's `webcontainer.conf` file, there's an opportunity to get too tricky with the coding of the `plugin-cfg.xml`. Consider this example: The server cluster `azsr01` has two application installed: X and Y. Server cluster `azsr02` has application Y installed as well. Two `<UriGroups>` are provided, both with `<Uri Name="/Y/*"/>` defined.

**Note:** Notice how there's no `<VirtualHostGroup>` specification.

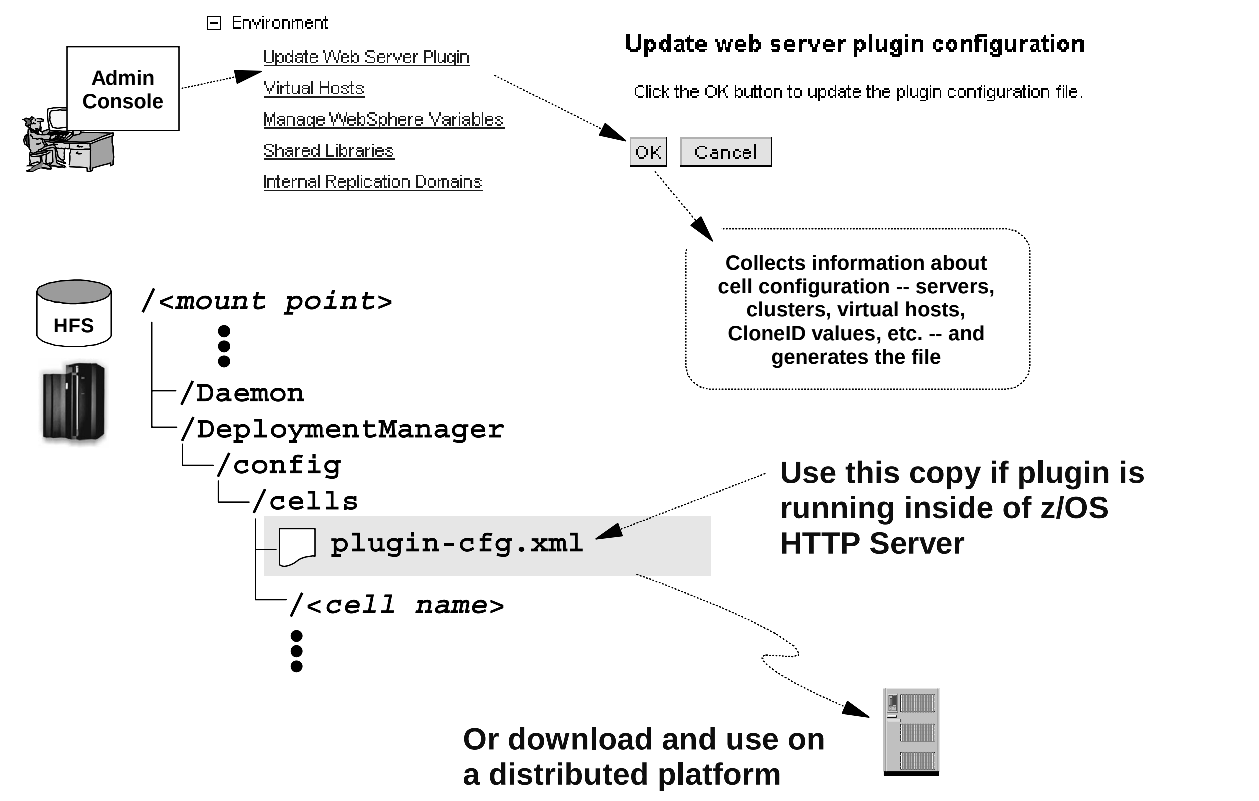
There's ambiguity here. A received URL with a context root of `/Y/index.html` will match on both `<UriGroup Name="ApplX"/>` *and* `<UriGroup Name="ApplY"/>`. Now which `<Route>` block will be chosen? Based on some testing, it appears that the Plugin will match the last `<Route>` statement it sees in the XML file that maps to a `<UriGroup>`. In this case that would be server cluster `azsr02`.

**Note:** Be careful, though: that conclusion isn't based on exhaustive testing. There might be other things that come into play that wasn't tested for. The basic point here is this: ambiguity is unpredicatable. Avoid it.

Does this mean that the same application -- application "Y" in this example -- can't be installed in two different server clusters? That's not the right conclusion. The same application can be installed in two different server clusters. But if you're going through the Plugin to get to the application, you need to remove the ambiguity from the XML. One way to do that is to use `<VirtualHostGroup>` (or a combination of that and `<UriGroup>`), and route the requests to the appropriate server cluster.

The next question is: where does this XML file come from? Is it necessary to code it up from a blank sheet of paper?

## Generating `plugin-cfg.xml`


Environment
Admin Console
Update Web Server Plugin
Virtual Hosts
Manage WebSphere Variables
Shared Libraries
Internal Replication Domains
Update web server plugin configuration
Click the OK button to update the plugin configuration file.
OK
Cancel
Collects information about cell configuration -- servers, clusters, virtual hosts, CloneID values, etc. -- and generates the file
HFS
/<mount point>
/Daemon
/DeploymentManager
/config
/cells
plugin-cfg.xml
/<cell name>
Use this copy if plugin is running inside of z/OS HTTP Server
Or download and use on a distributed platform

Thankfully, you *don't* have to develop the XML file from scratch. WebSphere Application Server for z/OS Version 5 will generate the `plugin-cfg.xml` file.

The generated copy is based on information about the WebSphere Application Server configuration seen by the administrative application. The administrative application rummarges around in the configuration files for the application server, and from that it generates an XML file.

On the next chart we'll look at what is in the generated file.

## Few Notes About Generated XML

### Generated XML a great starting point, but probably not exactly what you need ...

```
<?xml version="1.0" encoding="Cp1047"?>
<Config>
    <Log LogLevel="Trace" Name="/wasv5config/azcell/DeploymentManager/logs"/>
    <VirtualHostGroup Name="default_host">
        <VirtualHost Name="wsc1.washington.ibm.com:9518"/>
        <VirtualHost Name="wsc1.washington.ibm.com:9519"/>
        <VirtualHost Name="wsc2.washington.ibm.com:9548"/>
        <VirtualHost Name="wsc2.washington.ibm.com:9558"/>
    </VirtualHostGroup>
    <ServerCluster Name="azsr01Cluster">
        <Server CloneID="B9F91E06DC4511C100000C0C0000000109521845"
                     dBalanceWeight="2" Name="aznodea_azsr01a">
                     nsport Hostname="wsc1.washington.ibm.com" Port="9548" Protocol="http"/>
                     >
                     CloneID="B9F95C1EDD90F28500000BF40000000409521
                     dBalanceWeight="2" Name="aznodeb_azsr01b">
            <Transport Hostname="wsc2.washington.ibm.com" Port=
        </Server>
    </ServerCluster>
    <ServerCluster Name="dmgr_azdmnode_Cluster">
        <Server CloneID="B9F9295C449786C4000001380000001E09521845" Name="azdmnode_dmgr">
            <Transport Hostname="wsc1.washington.ibm.com" Port="9518" Protocol="http"/>
        </Server>
    </ServerCluster>
    <UriGroup Name="default_host_azsr01Cluster_URIs">
        <Uri AffinityCookie="JSESSIONID"
            AffinityURLIdentifier="jsessionid" Name="/mem/*"/>
        <Uri AffinityCookie="JSESSIONID"
            AffinityURLIdentifier="jsessionid" Name="/MyIVT/*"/
    </UriGroup>
    <Route ServerCluster="azsr01Cluster"
        UriGroup="default_host_azsr01Cluster_URIs" VirtualHostGroup="default_host"/>
</Config>
```

**Virtual Host for the Plugin itself won't appear here. You'll probably need to hand-code another.**

**May not look *exactly* like actual generated file**

**Creates `ServerCluster` for Deployment Manager. Unnecessary unless you're coming through Plugin to get to Admin Console**

**Routes generated use *both* `UriGroup` and `VirtualHostGroup`**

■

The generated XML file will serve as a great starting point, but you will still likely have to go into it and perform a few updates.

- The Virtual Host list will contain those Virtual Host aliases created in the WebSphere Application Server runtime. But those most likely will not be what you'll need to do virtual host checking at the Plugin level. The Plugin will do virtual checking based on the host name found on the client's URL coming into the HTTP Server. That's most likely a different port from the runtime application servers, and may very well be a different system altogether. So take a good look at the virtual hosts generated and see if you need to add another one for the host used by clients to access the HTTP server.

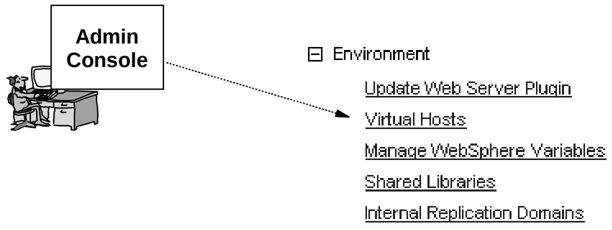
**Note:** In fact, any virtual host beyond that found on the client's URL is not necessary in this file. And if you're not doing any virtual host mapping down in the `<Route>` block, no virtual hosts are necessary.

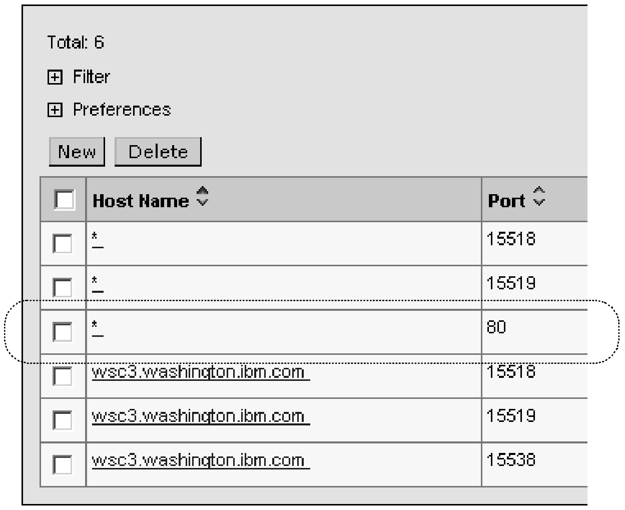
- A `<ServerCluster>` for the Deployment Manager will be created. This is probably unnecessary because you won't be putting any applications in the Deployment Manager. And it's unlikely (and not recommended for security purposes) to access the administrative application through the Plugin. So that whole `<ServerCluster>` could be removed. It's presence does not harm, unless it permits someone from gaining access to the administrative application you don't want accessing it.

- The `<Route>` blocks at the bottom of the generated XML file will match based on both URI and virtual host. If that's what you want, then fine. But if you want to match on one or the other but not both, then you'll need to change that.

## One Update Needed in Runtime

**Must create a "Virtual Host Alias" with port 80:**





| Host Name | Port |
|---|---|
| * | 15518 |
| * | 15519 |
| * | 80 |
| wsc3.washington.ibm.com | 15518 |
| wsc3.washington.ibm.com | 15519 |
| wsc3.washington.ibm.com | 15538 |

**The "Host Name" must match what client used to access HTTP server**

- **Easiest: asterisk wild card ... permits any value**

**Interestingly, the port must be 80, regardless of the port on which the HTTP server is listening**

**To run application, that application must be bound to Virtual Host in which this alias is defined.**

**On to troubleshooting ...**

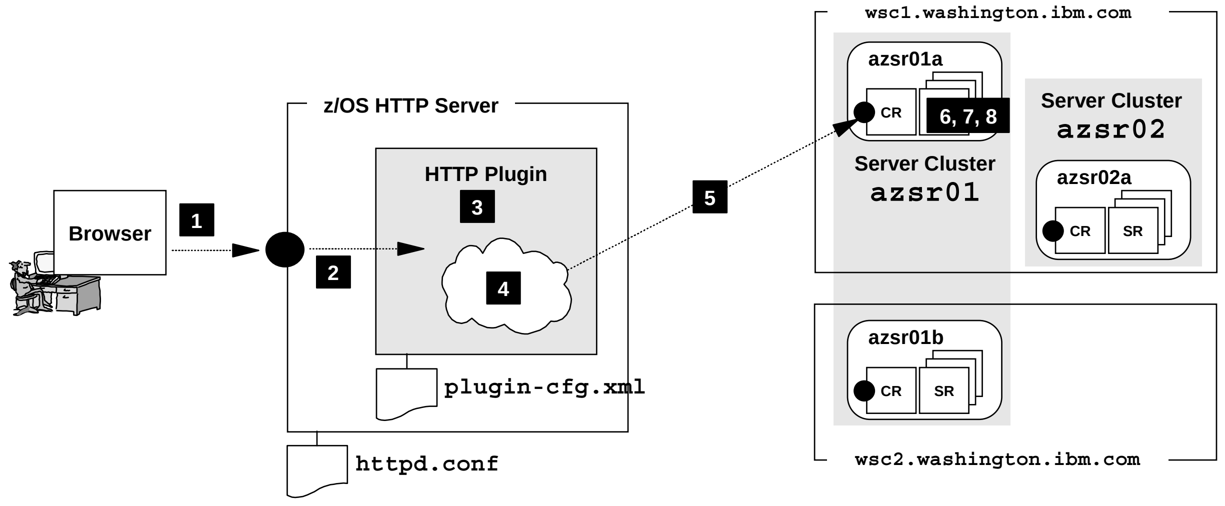
There is one thing you must do on the WebSphere Application Server for z/OS runtime to ready it to accept traffic from the Plugin: you need to create a "Virtual Host Alias." The HTTP flow coming from the Plugin will carry with it the host name of the system on which the Plugin runs, as well as -- interestingly -- port 80 (regardless of what port the Plugin is actually listening on).

What the chart shows is a Virtual Host Alias created with an asterisk for the "Host Name" and `80` for the "Port." The asterisk means any host value found on the URL will be accepted, and port 80 is what makes this whole thing work.

If you wanted to code an explicit host name, you would code the host name the client would use to get the request to the system running the Plugin. That may be a host value completely different from the host on which the WebSphere Application Server for z/OS runtime is running. Whatever the client uses to get the URL to the Plugin is the host you code into the virtual host alias. (Or code asterisk, which permits any host value.)

To run the application, the application itself must be bound to the Virtual Host in which this alias is defined. It's quite possible to have multiple Virtual Hosts defined to WebSphere Application Server. If one application is bound to one VH and another application is bound to another, and you want to run both applications with traffic from the Plugin, then you would have to code the port-80 alias in both Virtual Hosts.

## Troubleshooting Overview



1 **URL gets to HTTP Server**

2 **URL passed into Plugin**

3 **Plugin initialized**

4 **URL matches XML processing**

5 **Plugin sees Server as up**

6 **Plugin's "VH Alias" present**

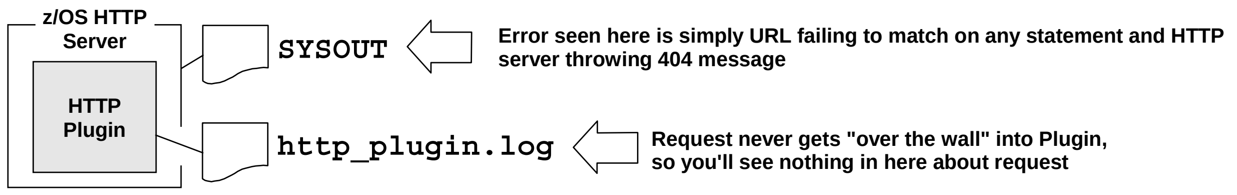
7 **Server recognizes context root**

8 **Application is running**

Here's a schematic diagram showing the things that need to go right to permit this whole system to work. The troubleshooting charts that follow all key off this chart.

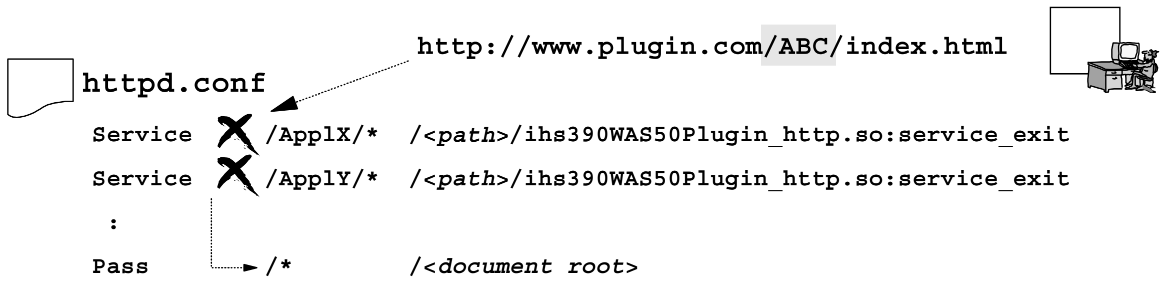
1. First, the URL must be able to get to the system running the Plugin. All manner of things can keep that URL from getting there: DNS problems, firewall problems, HTTP server not up, etc. We'll assume all those are working and focus on the Plugin and the other stuff downstream.
2. The URL must be passed from the HTTP Server into the Plugin. That requires a properly coded `Service` statement.
3. The Plugin must be initialized inside the HTTP Server. There's a bunch of things that'll prevent that from happening.
4. Once inside the Plugin, the URL must match against the XML in the Plugin's configuration file. If a match against the XML doesn't take place, the XML can't process the URL.
5. The Plugin will maintain knowledge of which of the defined servers in the WebSphere Application Server for z/OS runtime are up and accessible. So a match on the XML will work only if the backend server is actually up.
6. Once the URL gets passed back to the runtime, a Virtual Host alias must be defined in the runtime to permit the received URL to be processed.
7. The runtime must recognize the context root on the received URL
8. Finally, the application must be up, of course.

## Request Must Match "Service"

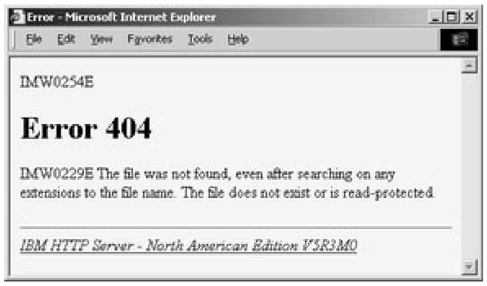


**Error seen here is simply URL failing to match on any statement and HTTP server throwing 404 message**

**Request never gets "over the wall" into Plugin, so you'll see nothing in here about request**



```
Service    /ApplX/*   /<path>/ihs390WAS50Plugin_http.so:service_exit
Service    /ApplY/*   /<path>/ihs390WAS50Plugin_http.so:service_exit
  :
Pass       /*         /<document root>
```


Error - Microsoft Internet Explorer
File Edit View Favorites Tools Help

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 - North American Edition V5R3M0*

**Lack of a proper "Service" statement for the URL typically results in 404 message**

**Remember to update `httpd.conf` when new applications deployed in backend runtime**

Before the Plugin will be able to handle any request, the Plugin has to receive the request from the HTTP Server. This is done with a `Service` statement. The received URL must match a properly coded `Service` statement for the request to even be *considered* for handling by the Plugin. If no Service statement matches, the request will fall through and be considered a request for a static web page, and it'll probably result in a failure to find the requested page. This is the standard 404 error coming out of the HTTP Server.

**Note:** The key is the fact the message is coming out of the HTTP Server. That tells you the request never made it beyond the HTTP Server.

Just because a request matches a Service statement doesn't necessarily mean the request will be handled by the Plugin. There are many things that can go wrong. We'll cover those next.

## Did Plugin Initialize?

```
JESMSGLG JES2                   2 BBOWEB   S
JESJCL   JES2                   3 BBOWEB   S
JESYSMSG JES2                   4 BBOWEB   S
SYSPRINT BBOWEB               101 BBOWEB   O
SYSOUT   BBOWEB               105 BBOWEB   O
```

**Look for the "Smiley Face"**

```
WebSphere HTTP Plug-in for z/OS and OS/390  initialization went OK :-)
```

**If no "Smiley Face," then look for the "Frowny Face:"**

```
WebSphere HTTP Plug-in for z/OS and OS/390  initialization FAILED (rc = 4) :-(
```

**It's possible that the Plugin failed to initialize even though no " `:-(` " is present:**

**Hint: when no " `:-(` " found, search on 'Failed to'**

**For example, if pointer to Plugin's module is incorrect**

```
Failed to load DLL module /<Config Root>/DeploymentManager/bin/ihs390WAS50Plugin_http.so
EDC5205S DLL module not found. (errnojr=0534011c)
```

**Let's take a look at all the things that must be present in `httpd.conf` configuration to permit Plugin to intialize ...**

■

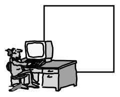
Assuming that the URL from the client found its way to the HTTP server and was picked up by the HTTP listener and matched on a `Service` statement, the first question is whether the Plugin is initialized. Much like the old WebSphere V3.5 product, what you look for as evidence of the Plugin's initialization is the "smiley face" in the `SYSOUT`.

If you don't find the smiley face, look for a "frowny face." Presence of that is a solid indication that the Plugin did not initialize, and it suggests the failure lies somewhere in the Plugin itself, as opposed to the Plugin's configuration settings in the `httpd.conf` file.

If no smiley or frowny face is present, it will mean the Plugin has failed to initialize for reasons related to the inability of the HTTP Server to load the plugin. Search for the string "Failed to" to find the error pointer.

What do you see on your browser if the Plugin isn't initilized? That's next.

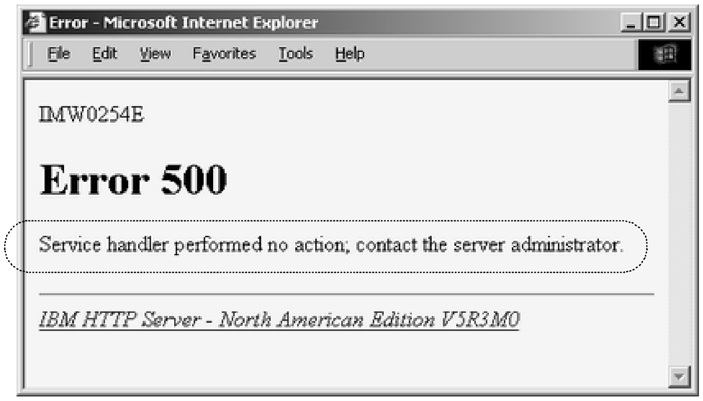
## Browser Symptom: No Plugin



```
http://www.plugin.com/ApplX/index.html
```

```
httpd.conf

ServerInit  /<path>/ihs390WAS50Plugin_http.so:init_exit  /<path>/plugin-cfg.xml
Service     /ApplX/*  /<path>/ihs390WAS50Plugin_http.so:service_exit
ServerTerm  /<path>/ihs390WAS50Plugin_http.so:term_exit
```

**But the Plugin isn't initialized, so the `service_exit` isn't available**



Error - Microsoft Internet Explorer
File Edit View Favorites Tools Help

IMW0254E

Error 500

Service handler performed no action; contact the server administrator.

*IBM HTTP Server - North American Edition V5R3M0*

**Caution! "Service Handler Performed No Action" may result even when Plugin is initialized.**

**Always check for Plugin initialization as first thing**

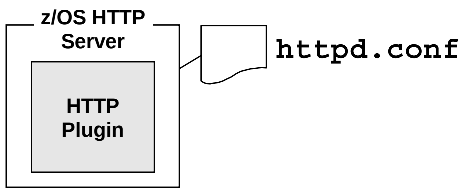
If the Plugin isn't initialized and you try to drive a URL against it, you'll get a browser message that says "Service Handler Performed no Action".

**Caution:** This error message can result even when the Plugin is initialized. It's a fairly generic error symptom and should not automatically be used to suggest the Plugin isn't initialized.

The problem here is that the URL was received by the HTTP server, and it matched against a valid `Service` statement, but the Plugin wasn't there to receive the request.

You should always check to make sure the Plugin has initialized before driving any work against the HTTP Server. This is particularly true when changes have been made to any configuration files.

## Primary Causes of Initialization Failures



**If Plugin doesn't initialize, look at four primary causes first. All have to do with the `ServerInit` statement in the `htttpd.conf` file:**

```
                            1                         2          3         4
ServerInit  /<path>/ihs390WAS50Plugin_http.so:init_exit  /<path>/plugin-cfg.xml
Service     /<URL>/*  /<path>/ihs390WAS50Plugin_http.so:service_exit
ServerTerm  /<path>/ihs390WAS50Plugin_http.so:term_exit
```

**1 Plugin ".so" module not found or can't be loaded**
- **Check directory path to module**
- **Check case (it matters on directory paths)**
- **Check file name, including case**
- **Check permissions on directories and file itself (HTTP Server ID needs "read" minimum)**

**2 The "exit" specified on module incorrect**
- **Must be `init_exit`, not `service_exit` or `term_exit` like the other statements**

**3 Plugin's XML file not found or can't be loaded**
- **Plugin's XML file specified as parameter on end of ServerInit statement**
- **Check directory path to file, file name, case and permissions**

**4 Bad contents of Plugin's XML file**
- **See Plugin's log file for pointer to line of XML file that's in error**

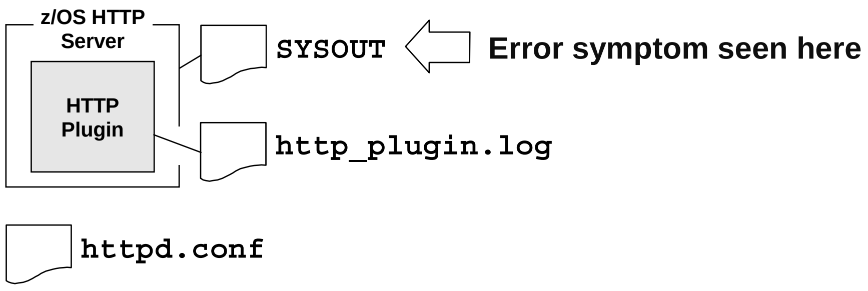
■

There are a few key things that'll prevent the Plugin from initializing. The focus area is the `ServerInit` statement in the `httpd.conf` file.

1. The executable file for the Plugin -- the "*.so file" -- must be located and loaded. If the HTTP Server can't find, or can't load, that module, initialization will fail. Check the things shown on the chart to make sure the pointer is absolutely correct.
2. The "exit" on the end of the `ServerInit` must be `init_exit`. If you copied one of the other lines to use as a model for the ServerInit statement, there's a possibility that one of the other "exits" is on the statement (`service_exit` or `term_exit`).
3. The pointer to the `plugin-cfg.xml` file is either wrong or permissions don't permit the loading of the file.
4. Finally, the contents of the `plugin-cfg.xml` file may prevent the initialization of the Plugin.

Let's explore the symptoms you'll see for each of these.

## Plugin Module Not Found or Loaded



```
ServerInit  /<path>/ihs390WAS50Plugin_http.so:init_exit  /<path>/plugin-cfg.xml
```

Anything that prevents the HTTP Server from locating and loading module will cause this problem

- Incorrect directory; incorrect case
- Wrong module name; incorrect case
- Restrictive permissions

**No "smiley," no "frowny" ... just the following:**

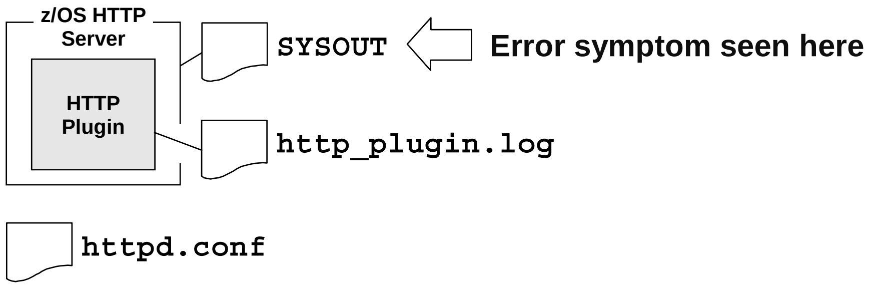
```
  :
Failed to load DLL module /<path>/DeploymentManager/bin/ihs390WAS50Plugin_http.so
EDC5205S DLL module not found. (errnojr=0534011c)
  :
```

**HTTP Server must be able to locate module before it can load it**

If the ServerInit statement is coded in such a way that the Plugin's module can't be found or loaded, what you'll see is an error symptom in the HTTP Server's SYSOUT log. You won't find the smiley face, of course, nor the frowny face. What you'll see is a message saying "Failed to load DLL module." Again, search on "Failed to" when you can't find the smiley or frowny faces.

The key message here is that the HTTP Server must be able to find and load the module. That's the fundamental first step.

## Wrong "Exit" on ServerInit



```
ServerInit  /<path>/ihs390WAS50Plugin_http.so:wrong_exit  /<path>/plugin-cfg.xml
```

**Common error: copying `Service` statement line to form `ServerInit` and then forgetting to change exit**

**Anything other than `:init_exit` is incorrect exit.**

```
API... Successful loading shared library "/<path>/ihs390WAS50Plugin_http.so"
API... Trying to get fn pointer "wrong_exit" from module "/<path>/ihs390WAS50Plugin_http.so"
Failed to load function wrong_exit: EDC5214I Requested function not found in this DLL.
IMW0437E Return code 123 loading function wrong_exit from DLL module /<Plugin module>
IMW0438E Serverinit Error: server did not load functions from DLL module /<Plugin module>
```

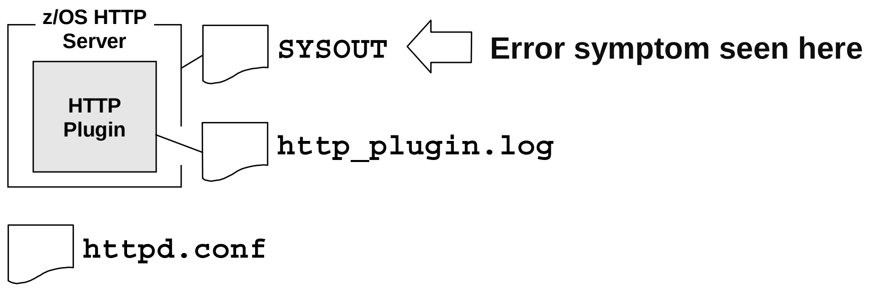
**Plugin module has three exits: `init_exit`, `server_exit` and `term_exit`. Only `init_exit` used to load module.**

If the "exit" specified on the `ServerInit` statement is anything other than `init_exit`, you'll get an error initializing the Plugin. Again, no smiley or frowny face. What you'll see is what's illustrated on the chart:

- The string "Failed to" will be found
- On that same line you'll see the HTTP Server telling you that the exit specified on the `ServerInit` statement (`wrong_exit` in this example) could not be loaded because it wasn't found in the DLL.

You'll see something very similar if you specify one of the other valid exits -- `service_exit` or `term_exit` -- on the `ServerInit`. Those are valid exits in the DLL, but not valid for the `ServerInit`.

## plugin-cfg.xml Not Found or Specified



```
ServerInit  /<path>/ihs390WAS50Plugin_http.so:wrong_exit  /<path>/plugin-cfg.xml
```

- Incorrect directory; incorrect case
- Parameter simply missing
- Wrong file name; incorrect case
- Restrictive permissions

**Anything that prevents the Plugin from locating and reading XMLwill cause this problem**

```
WebSphere HTTP Plug-in for z/OS and OS/390 initializing with configuration file : /<path>/<file>
ws_common: websphereUpdateConfig: Failed parsing the plugin config file
WebSphere HTTP Plug-in for z/OS and OS/390  initialization FAILED (rc = 3) :-(
IMW0438E Serverinit Error: server did not load functions from DLL module /<Plugin
```

**Find on :-(**

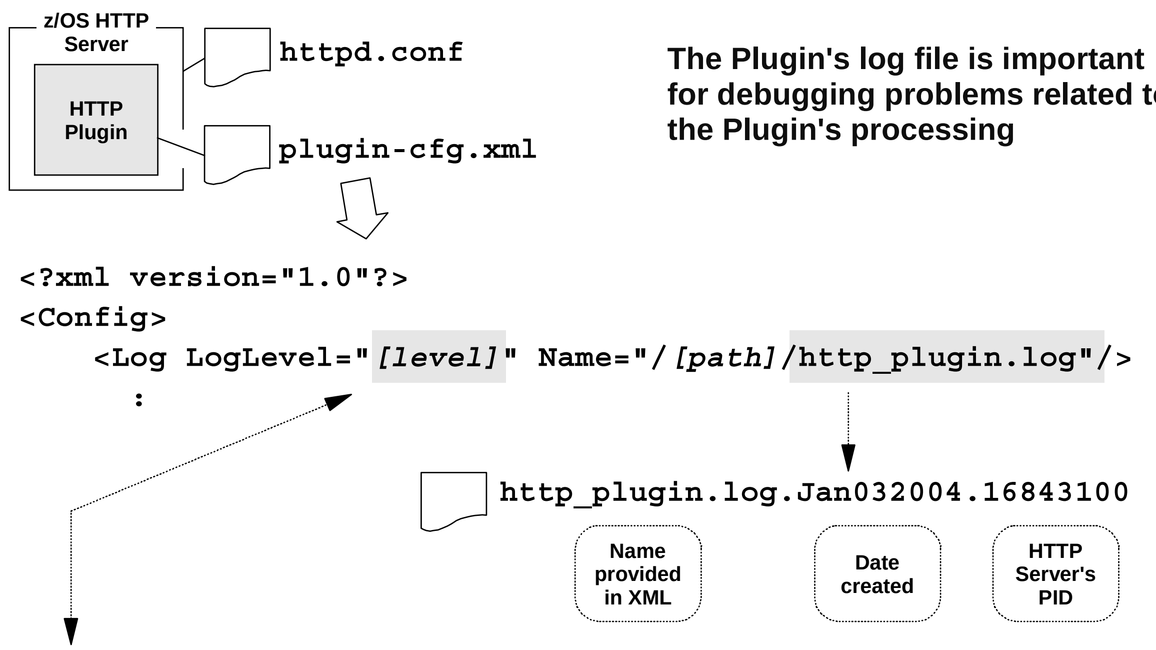
**Note "rc=3" ... bad XML contents is a "rc=4". "rc=3" means file itself can't be found, rather than what's inside file is bad.**

**Plugin module must be able to find XML file. There's no "default" that's taken.**

Once the proper DLL and exit is loaded, the Plugin will go in search of the configuration XML file. Unlike the old WAS V3.5 product, there is no default `was.conf` that's pulled from the installation directory. If the Plugin can't find the XML file pointed to, the Plugin won't initialize.

This error will generate a frowny face. You'll also see the key message: "Failed parsing the plugin config file" and "rc=3". The return code is important because it's drawing a distinction between a file that can't be found or loaded (rc=3) and one that was found, read, and full of errors (rc=4).

## Where Plugin Logging Goes



```
<?xml version="1.0"?>
<Config>
    <Log LogLevel="[level]" Name="/[path]/http_plugin.log"/>
      :
```

```
http_plugin.log.Jan032004.16843100
```

**The Plugin's log file is important for debugging problems related to the Plugin's processing**

| | |
|---|---|
| `Error` | Error messages |
| `Warn` | Warning + Error messages |
| `Trace` | Trace + Warning + Error messages |

**File is in EBCDIC and is quite readable.**
**Beware of "Trace" -- lots of output.**
**Default location:**
`/<config root>/DeploymentManager/logs`

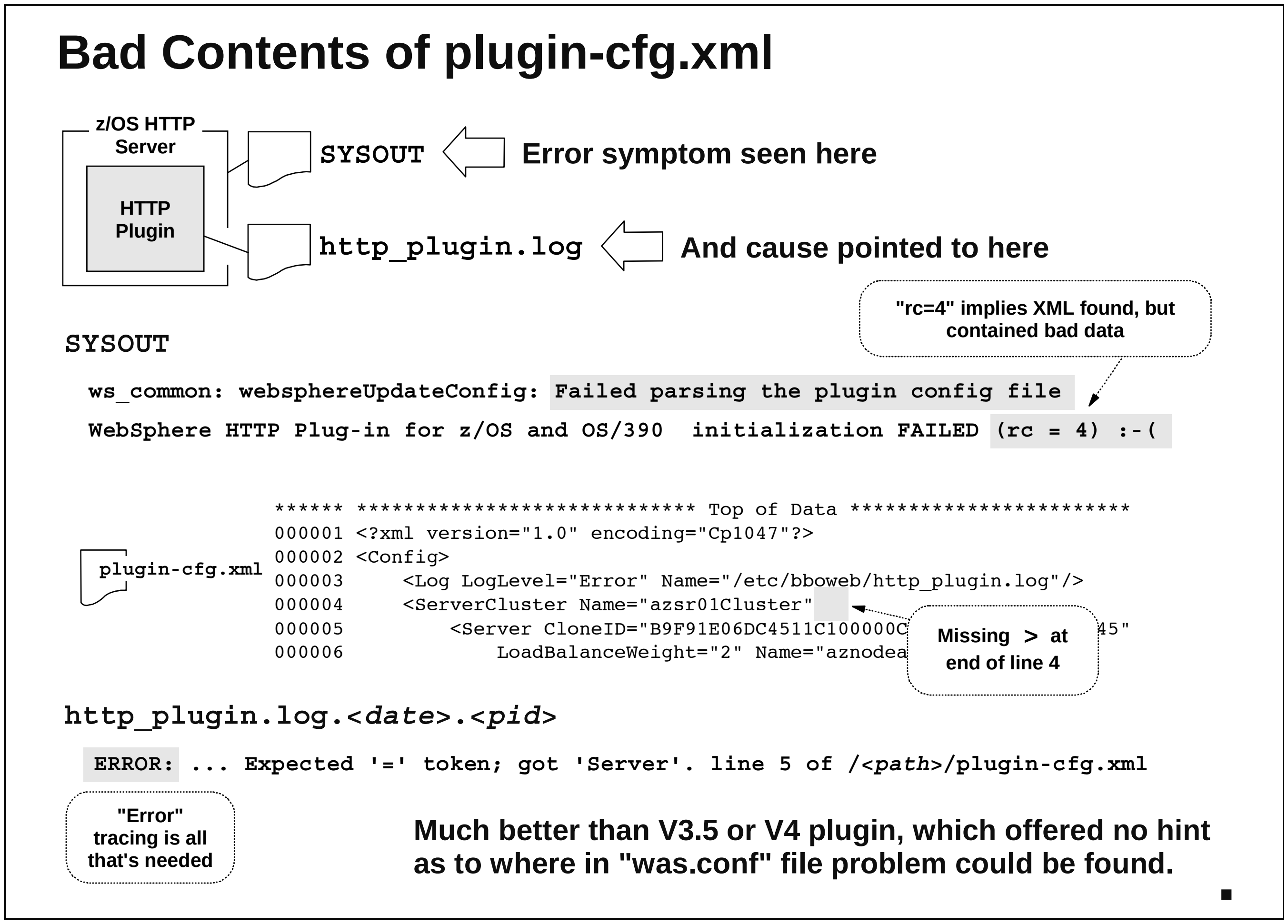
Once we're past the other issues, and the Plugin itself is initialized, we're facing the next wave of potential problems. Those problems will be logged into the Plugin's log, which goes to the location specified in the `plugin-cfg.xml` file.

The `<Log>` statement in the `plugin-cfg.xml` file provides two things: the amount of logging to be performed, and the place where the logging will be written:

- **Amount of logging** -- three options here: `Error`, `Warn` and `Trace`. `Error` has the least amount of logging, `Trace` the most. Be careful with `Trace`: it puts out a lot of messages.
- **Location of logging** -- here you specify the directory and file name of the logging file. Be sure the HTTP Server's ID has the ability to write to the directory specified. The Plugin will modify the supplied file name and append the date and the "Process ID" of the HTTP server from which the logging is occurring.

  **Note:** There is no time-stamp in the file name itself, so be aware when multiple files are present.

By default the log will be written to the Deployment Manager's `/logs` directory down under the configuration root. If you want to change that location, then do so. However, remember this each time you re-generate the `plugin-cfg.xml` file because the generated file will be back to the default location.



## Bad Contents of plugin-cfg.xml

z/OS HTTP Server
HTTP Plugin
SYSOUT
Error symptom seen here
http_plugin.log
And cause pointed to here
"rc=4" implies XML found, but contained bad data
plugin-cfg.xml
Missing > at end of line 4
"Error" tracing is all that's needed

SYSOUT

```
ws_common: websphereUpdateConfig: Failed parsing the plugin config file
WebSphere HTTP Plug-in for z/OS and OS/390  initialization FAILED (rc = 4) :-(
```

```
****** ***************************** Top of Data ************************
000001 <?xml version="1.0" encoding="Cp1047"?>
000002 <Config>
000003     <Log LogLevel="Error" Name="/etc/bboweb/http_plugin.log"/>
000004     <ServerCluster Name="azsr01Cluster"
000005         <Server CloneID="B9F91E06DC4511C100000C                45"
000006             LoadBalanceWeight="2" Name="aznodea
```

http_plugin.log.<*date*>.<*pid*>

```
ERROR: ... Expected '=' token; got 'Server'. line 5 of /<path>/plugin-cfg.xml
```

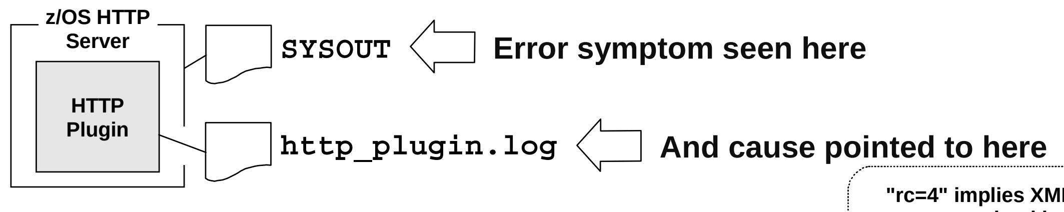
**Much better than V3.5 or V4 plugin, which offered no hint as to where in "was.conf" file problem could be found.**

Let us assume that the Plugin successfully initialized, and the `plugin-cfg.xml` file was located and read. Now, let's look at what happens if there's a problem inside the XML.

Here we are fortunate, for the Plugin log will tell you what line the error was on (this is a departure from the WAS V3.5 days, where it did not indicate where the problem was). In the example above, the absence of a closing right-bracket is the culprit. Even still, the error message is still a touch cryptic: the problem is actually on line 4, but the parser doesn't "see" the problem until it gets to line 5 and encounters something it didn't expect.

**Note:** Bad contents of the XML file results in a RC=4 status, not 3. Recall that RC=3 meant the XML file was not found. RC=4 means it was found, but contained errors.

## Can't Resolve Host Name in XML



"rc=4" implies XML found, but contained bad data

**SYSOUT**

```
ws_common: websphereUpdateConfig: Failed parsing the plugin config file
WebSphere HTTP Plug-in for z/OS and OS/390  initialization FAILED (rc = 4) :-(
```

plugin-cfg.xml

```
000004     <ServerCluster Name="azsr01Cluster">
000005         <Server CloneID="B9F91E06DC4511C100000C0C000
000006             LoadBalanceWeight="2" Name="aznodea_azsr
000007             <Transport Hostname="www.not-there.com" P
```

Hostname specified can't be resolved

**http_plugin.log.<*date*>.<*pid*>**

```
ERROR: ws_transport: transportSetServerAddress: unable to resolve host name: <host name>
ERROR: lib_sxp: sxpParse: End element returned FALSE for Transport. line 7 of <plugin-cfg.xml>
```

"Error" tracing is all that's needed

**Watch out for typos in your host name values**

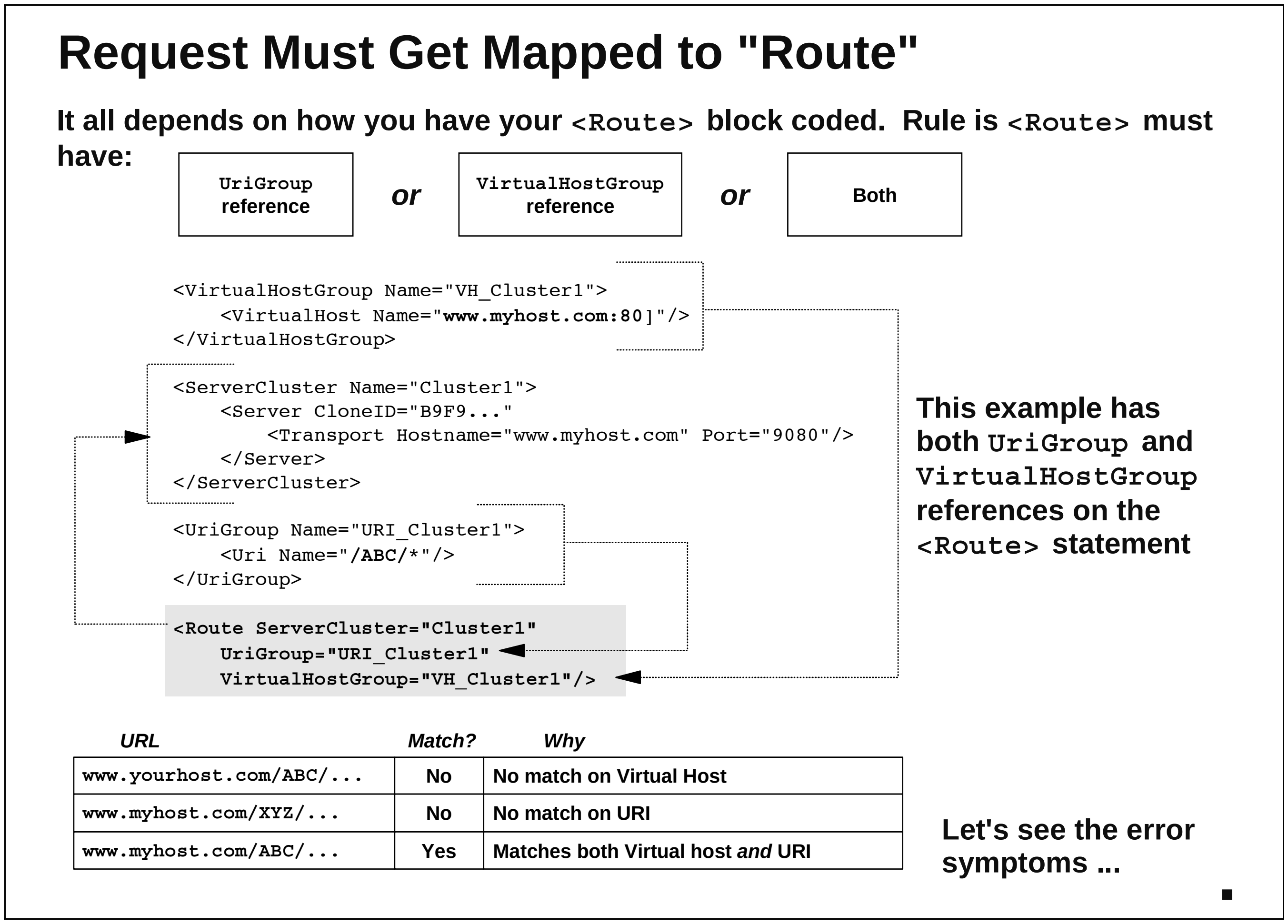
**(IP addresses aren't resolved; wrong addresses are treated like an IP stack that's not present; Plugin will simply balance to other server in cluster if possible)**

■

Here's a somewhat obscure problem: the host name specified on the `Hostname=` values in the XML can't be resolved to an actual host. This problem will not likely result from the default generated `plugin-cfg.xml` file, but may result if you hand-edit the file. What you'll see is this:

- Plugin not initialized (frowny face with rc=4)
- Log shows error and points to line where host name is found in `plugin-cfg.xml`
- Tells entire story: can't resolve the host name.

One way around this is to code the actual IP address rather than the host name. But if the host name you code can't be resolved, and you're certain it's the right value, then there's something wrong with the IP configuration of your system that would prevent that name from being resolved properly.



## Request Must Get Mapped to "Route"

**It all depends on how you have your `<Route>` block coded. Rule is `<Route>` must have:**

| | | | | |
|---|---|---|---|---|
| `UriGroup` reference | *or* | `VirtualHostGroup` reference | *or* | Both |

```
<VirtualHostGroup Name="VH_Cluster1">
    <VirtualHost Name="www.myhost.com:80]"/>
</VirtualHostGroup>

<ServerCluster Name="Cluster1">
    <Server CloneID="B9F9..."
        <Transport Hostname="www.myhost.com" Port="9080"/>
    </Server>
</ServerCluster>

<UriGroup Name="URI_Cluster1">
    <Uri Name="/ABC/*"/>
</UriGroup>

<Route ServerCluster="Cluster1"
    UriGroup="URI_Cluster1"
    VirtualHostGroup="VH_Cluster1"/>
```

**This example has both `UriGroup` and `VirtualHostGroup` references on the `<Route>` statement**

| *URL* | *Match?* | *Why* |
|---|---|---|
| `www.yourhost.com/ABC/...` | No | No match on Virtual Host |
| `www.myhost.com/XYZ/...` | No | No match on URI |
| `www.myhost.com/ABC/...` | Yes | Matches both Virtual host *and* URI |

**Let's see the error symptoms ...**

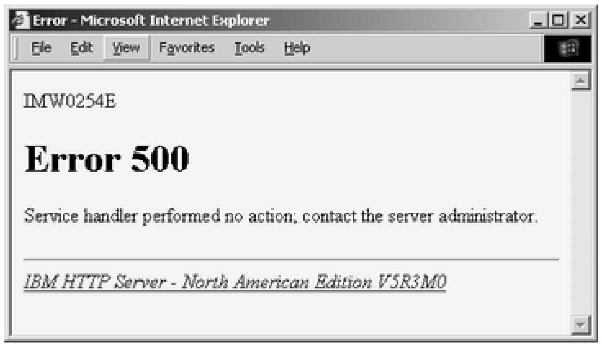
Now we get into the very heart of the XML file and the mapping of the received URL to the `<Route>` block that applies. A `<Route>` block needs to have either a `UriGroup` reference, a `VirtualHostGroup` reference or both. It can't have neither. Something has to be used to map the URL to the `<Route>`.

The example above shows the default mapping where the `<Route>` block has *both* the `UriGroup` and the `VirtualHostGroup` references. The virtual host is `www.myhost.com` and the URI is `/ABC/*`. What that means is that the only URL that'll map to that is one that has *both* `www.myhost.com` and `/ABC/*` on it. Anything different from that will not map.

Recall that the <Route> block does not need to have both `UriGroup` and `VirtualHostGroup` references. It may have one or the other. So if it had only a `UriGroup` reference and the received URL didn't match any of the URIs, then a failure would occur. The same holds for a `VirtualHostGroup` reference where the host on the URL doesn't match.

What happens when a received URL fails to map to a `<Route>`? That's next.

## Route Not Mapped Symptom


Error - Microsoft Internet Explorer
File Edit View Favorites Tools Help

IMW0254E

**Error 500**

Service handler performed no action; contact the server administrator.

*IBM HTTP Server - North American Edition V5R3M0*

**Regardless of type of failure:**

- **Failure to match URI**
- **Failure to match VirtualHost**
- **Failure to match combination of both**

**The browser error is the same:**

- **Error 500 -- "Service handler performed no action."**

**Caution: this might also indicate the Plugin isn't initialized.**

**The Plugin's log tells the story:**

**No match of URI**

```
TRACE: ws_common: webspherеUriMatch: Failed to match: /Test/index.html
TRACE: ws_common: websphereFindServerGroup: No route found
TRACE: ws_common: websphereHandleRequest: Failed to find a server group
TRACE: ws_common: websphereEndRequest: Ending the request
```

**No match of VHost**

```
TRACE: ws_common: websphereVhostMatch: Failed to match: www.plugin.com:8070
TRACE: ws_common: websphereFindServerGroup: No route found
TRACE: ws_common: websphereHandleRequest: Failed to find a server group
TRACE: ws_common: websphereEndRequest: Ending the request
```

**If *both* VH and URI are mismatched, VH error will appear in Trace**

**Need `Loglevel="Trace"` set for this information**

**Next up: when the backend server isn't available ...**

What we see here is that the "Error 500" message gets used for lots of different problems. This is true regardless of the mapping failure: failure on URI, failure on VH, failure on both.

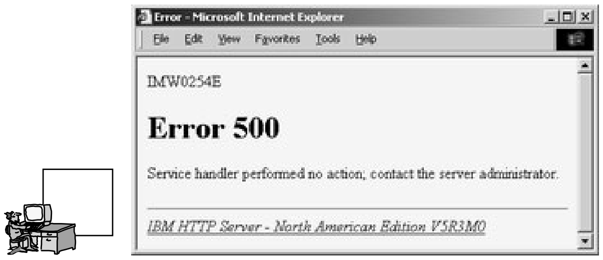
The only way to isolate this down the actual failure is to go into the Plugin's log. If you can't locate the log file, it may be because the Plugin isn't initialized. "Error 500" can result for that cause as well.

You'll need the "Trace" option set to capture the failure cause here. And what you'll see is what's presented in the chart:

- When the `<Route>` block contains only `UriGroup`, a failure to match a URI will show a "Failed to match" message with the URI value received.
- When the `<Route>` block contains only `VirtualHostGroup`, a failure to match a host will show a "Failed to match" message with the host value received.
- When *both* `UriGroup` and `VirtualHostGroup` are coded on `<Route>`, the error message will be the same as what appears when only `VirtualHostGroup` is coded.

## Plugin Must See Server as "Up"

**What happens when a URL maps to a `<Route>`, but all the servers in that Cluster are down? (For example, you simply forgot to start those servers)**


Error - Microsoft Internet Explorer
File Edit View Favorites Tools Help

IMW0254E

**Error 500**

Service handler performed no action; contact the server administrator.

*IBM HTTP Server - North American Edition V5R3M0*

**Browser error is the ubiquitous "Error 500"**

```
TRACE: ws_common: websphereExecute: Executing the transaction with the app server
TRACE: ws_                    tStream: Getting the stream to the app server
TRACE: ws_                    StreamDequeue: Checking for existing stream from the queue
ERROR: ws_                    Stream: Failed to connect to app server, OS err=1128
ERROR: ws_                    cute: Failed to create the stream
ERROR: ws_server: serverSetFailoverStatus: Marking aznodea_azsr01a down
  :
TRACE: ws_common: websphereExecute: Executing the transaction with the app server
TRACE: ws_                    tStream: Getting the stream to the app server
TRACE: ws_                    StreamDequeue: Checking for existing stream from the queue
ERROR: ws_                    Stream: Failed to connect to app server, OS err=1128
ERROR: ws_                    cute: Failed to create the stream
ERROR: ws_server: serverSetFailoverStatus: Marking aznodeb_azsr01b down
  :
ERROR: ws_common: websphereWriteRequestReadResponse:
           Failed to find an app server to handle this request
```

**Tries first server in Cluster and fails. Marks server as "down"**

**Tries second server in Cluster and fails. Marks server as "down"**

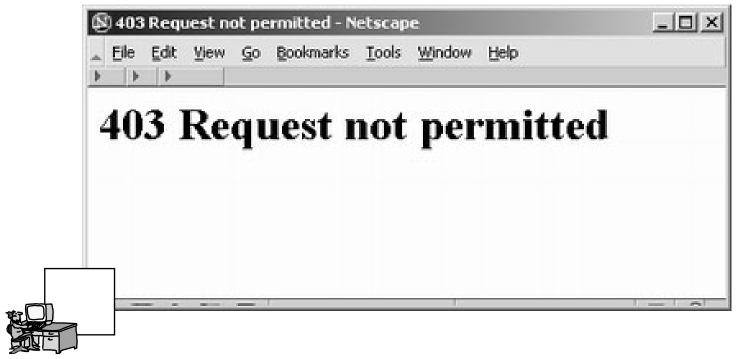
**Runs out of servers in cluster and gives up**

Let's assume that the received URL has mapped to a `<Route>` block and thus to a `<ServerCluster>` group. For the request to go back to the runtime servers, those servers have to be up and running. What happens if they're not?

You'll see the same "Error 500" on the browser screen. We're starting to see that "Error 500" is not a very valuable debugging indication. In the Plugin's log -- again, with Trace on -- you'll see the Plugin trying the first server in the `<ServerCluster>` and failing that, try the second. Eventually it'll run out of servers in the cluster and give up: "Failed to find an app server to handle this request."

## "TrustedProxy" Not Set

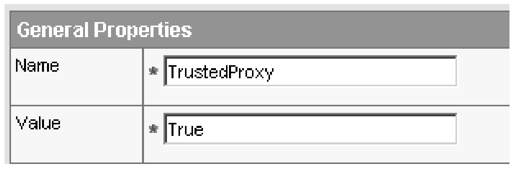
**If the server's HTTP port does not have the custom property "TrustedProxy" set, then the server won't permit the flow from the Plugin:**


403 Request not permitted - Netscape
File Edit View Go Bookmarks Tools Window Help

403 Request not permitted

**The plugin serves as a proxy -- it forwards requests on to the application server. This tells the server to "trust" the inbound request.**

**Application Servers ⇨ *<server>* ⇨ Web Container ⇨ HTTP Transport ⇨ *<port>* ⇨ Custom Properties**



| General Properties | |
|---|---|
| Name | * TrustedProxy |
| Value | * True |

- **Do this for both the non-SSL and SSL port**
- **Do this for all servers that receive plugin flows**
- **Stop/restart server to pick up change**

■

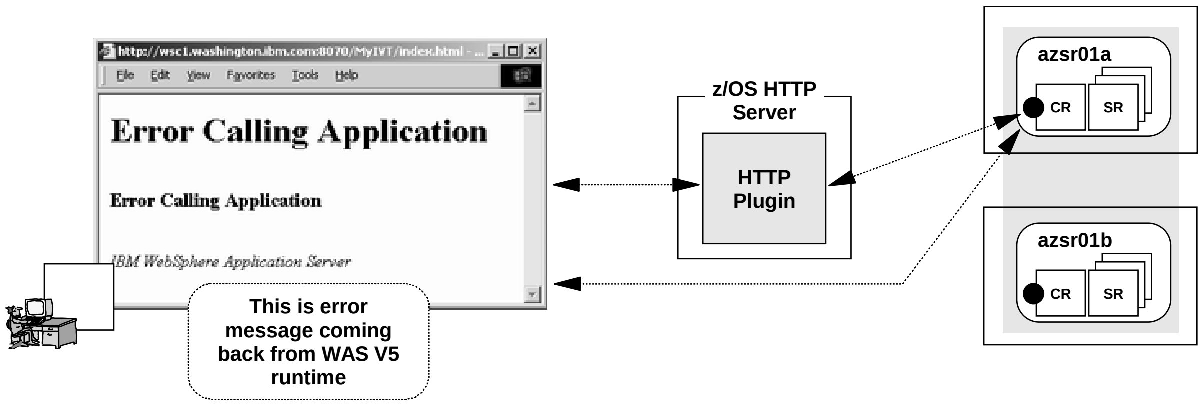
The plugin acts as a proxy, forwarding requests on to the application server in the WebSphere runtime. In order to tell the server that the flow from the proxy can be trusted, it's necessary to create a custom property under the application server's HTTP ports with name of `TrustedProxy` and a value of `True`.

Failure to do this will result in an error message of "403 Request not permitted" coming back from the application server. You can tell this isn't coming from the plugin itself because it's not a "500" message and there's no indication that it's coming from the HTTP server. Therefore, both those pieces of the puzzle -- the HTTP server and the plugin -- are doing their job. The flow goes up to the application server in the WebSphere runtime, but the server rejects it because the "TrustedProxy" setting is not there.

To set this value, drill down in the Admin Console under the application server as shown in the chart above. You'll have to do this for all the servers that will receive flows from the plugin, and you'll have to do it for both the non-SSL port and the SSL port for each of the servers. When you've set the property, then stop/restart the servers to pick up the change.

## No Virtual Host Match for Client URL

**If no virtual host alias in the WebSphere runtime matches the URL sent in by the client, then WAS runtime will reject:**


http://wsc1.washington.ibm.com:8070/MyIVT/index.html - ...
File Edit View Favorites Tools Help
z/OS HTTP Server
HTTP Plugin
azsr01a
CR
SR
azsr01b
CR
SR
This is error message coming back from WAS V5 runtime

Error Calling Application

Error Calling Application

IBM WebSphere Application Server

**Key Points:**

- **Plugin is doing its job ... Plugin trace will show normal processing**
- **This illustrates the difference between virtual host in plugin XML and virtual host in WAS runtime**
- **Key off the browser error message -- this is Application Server message, which means flow got to application server**
  **All issues related to Plugin initialization, route mapping and servers being up are not the issue**

This is the case where the virtual host alias in the WebSphere Application Server for z/OS runtime doesn't match the value found on the URL sent by the original client. There's a couple of reasons why this may happen:

- The virtual host alias was never coded in the first place
- The virutal host alias was coded, but is slightly different from the value received from the client
- The application is mapped to a different VH from the one in which this alias was created.

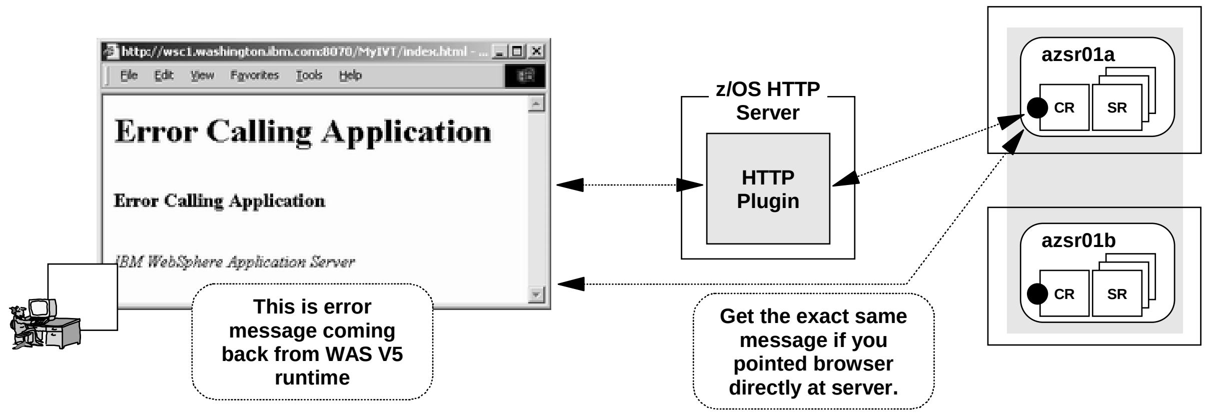
The error you'll see on the browser is "Error Calling Application," and you'll see also that the message came from the "WebSphere Application Server." This is a telling piece of information: it indicates that the request got back to the WAS runtime environment. That means the Plugin is doing it's job ... don't bother poring through the Plugin's trace ... you'll see it simply reporting that everything worked okay.

Notice how we have two different Virtual Host values in play here: one in the Plugin's XML file, one in the WAS V5 runtime. The Plugin's XML file is only invoked if the `<Route>` block has a `VirtualHostGroup` reference. But if it does, then the request received from the client must match that VH in order for the Plugin to consider sending the request back to the server runtime. The application server runtime has its own Virtual Host definitions, and here they're a requirement. Applications must be mapped to a Virtual Host, and an inbound URL must pass muster with the defined Virtual Host before the application will be invoked.

**Note:** Provide a virtual host with the port used by the HTTP server. Coding a virtual host with a *host* value of asterisk makes things easier.

## URL Context Root Must Match Appl's

**The UriGroup values in the `plugin-cfg.xml` may not match the actual Context Roots in the server. The Plugin will pass the request back, only to have it fail:**


http://wsc1.washington.ibm.com:8070/MyIVT/index.html - ...

Error Calling Application

Error Calling Application

IBM WebSphere Application Server
File Edit View Favorites Tools Help
This is error message coming back from WAS V5 runtime
z/OS HTTP Server
HTTP Plugin
Get the exact same message if you pointed browser directly at server.
azsr01a
CR
SR
azsr01b
CR
SR

**Key Points:**

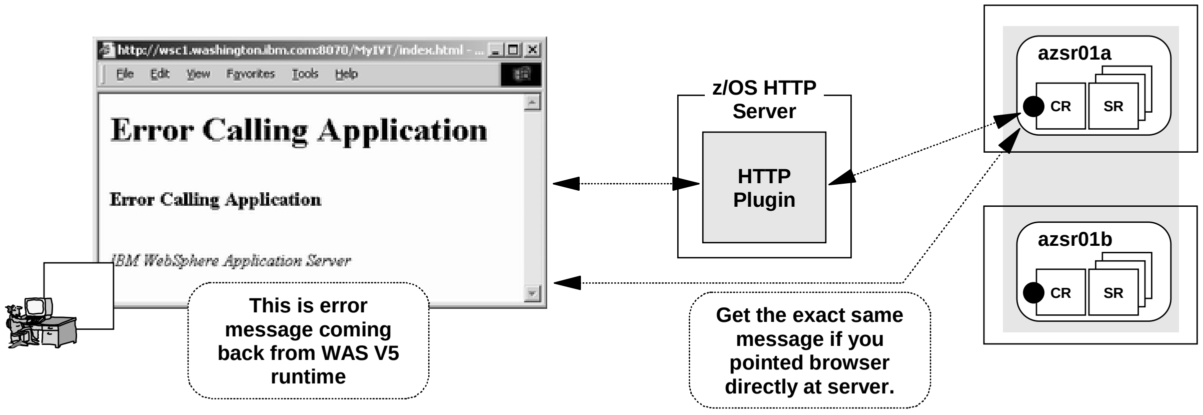
- **Plugin is doing its job ... Plugin trace will show normal processing**
- **This illustrates Plugin has no idea what applications are installed**
  **Generated XML will have the Context Roots of actual applications. But XML file is open to hand-editing and Plugin will send along any request that maps to a route.**
- **Debugging this will require looking at application server traces**
  **Plugin trace can be used to determine which application server request went to**

If the context root on the URL doesn't match a defined context root for a deployed application, WebSphere will reject the request with the same "Error Calling Application." Again, given that the error message came out of the WebSphere runtime, we know we're past the Plugin. This shows that the Plugin knows nothing of the applications that are deployed back in the runtime. The generated XML will have the context roots of the deployed application at the time the XML was generated. But subsequent application installations won't be reflected in the XML until the file is regenerated, or the XML file is hand-edited.

**Note:** This is different from the Version 4 "Local Redirector Plugin," which *did* maintain knowledge of the deployed applications. But that plugin performed a completely different role from this one.

## Application Must Be Started

**If the application is valid in every respect except just not started, then you get error message out of WebSphere Application Server runtime:**


http://wsc1.washington.ibm.com:8070/MyIVT/index.html - ...
File Edit View Favorites Tools Help
Error Calling Application
Error Calling Application
IBM WebSphere Application Server
This is error message coming back from WAS V5 runtime
z/OS HTTP Server
HTTP Plugin
Get the exact same message if you pointed browser directly at server.
azsr01a
CR
SR
azsr01b
CR
SR

**Key Points:**

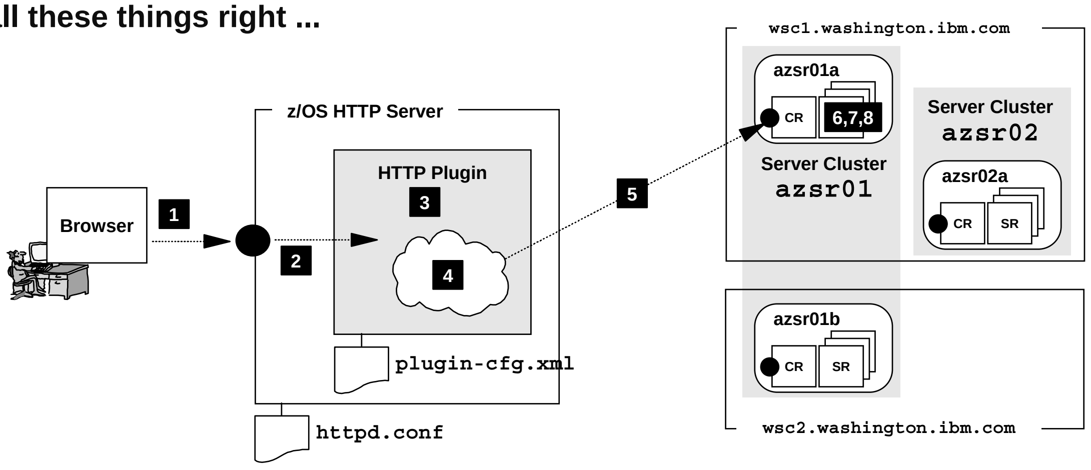
- **Plugin is doing its job ... Plugin trace will show normal processing**
- **This illustrates Plugin doesn't know about application status**
- **Key off the browser error message -- this is Application Server message, which means flow got to application server**
  **All issues related to Plugin initialization, route mapping and servers being up are not the issue**

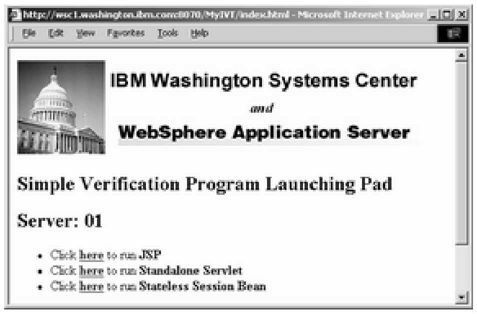
■

Once you've gotten past all the other hurdles, the runtime will try to drive the application itself. The application has to be started, of course. If the application is down you get ... that's right, the "Error Calling Application" message.

## Success!

**Get all these things right ...**




http://wsc1.washington.ibm.com:8070/MyIVT/index.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help

IBM Washington Systems Center

*and*

**WebSphere Application Server**

**Simple Verification Program Launching Pad**

**Server: 01**

- Click here to run **JSP**
- Click here to run **Standalone Servlet**
- Click here to run **Stateless Session Bean**

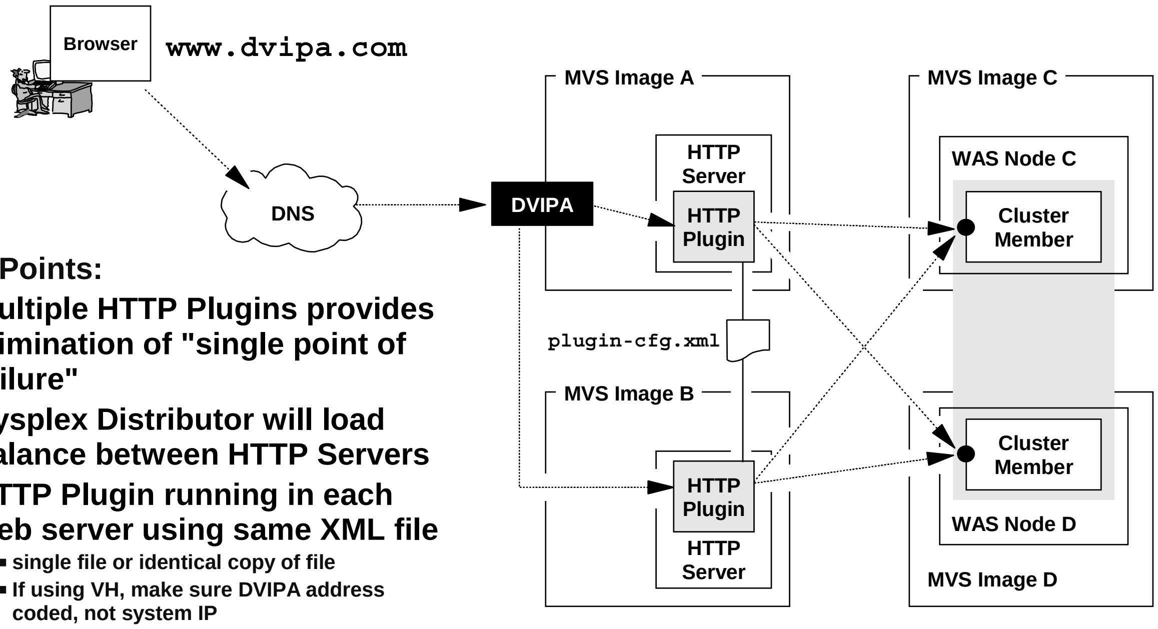
**And the system will perform and return application result to the browser**

**Last point: combining Plugin with Sysplex Distributor ...**

With all those things in place, you're able to drive the application.

## Sysplex Distributor and Plugin

**Even if Session Affinity is a requirement, it's possible to incorporate Sysplex Distributor out front of multiple Plugins:**



**Key Points:**

- **Multiple HTTP Plugins provides elimination of "single point of failure"**
- **Sysplex Distributor will load balance between HTTP Servers**
- **HTTP Plugin running in each web server using same XML file**
  - **single file or identical copy of file**
  - **If using VH, make sure DVIPA address coded, not system IP**
- **Plugin maintains Session Affinity to backend servers**

The final topic we'll address is that of providing some redundancy for the Plugin. This will involve using Sysplex Distributor to route the inbound requests to multiple copies of the HTTP Server, each running the Plugin.

**???** Doesn't this break session affinity? No. Sysplex Distributor will balance between two identical HTTP servers. The Plugin running inside each will be identical to one another. From there the Plugin will maintain affinity to the proper backend server.

The key to this is making sure that the two HTTP Servers make use of identical `plugin-cfg.xml` files (or the same file, if shared). If both copies of the Plugin are running the same configuration file, then the inbound request will be handled the same way, regardless of which HTTP server Sysplex Distributor routed the request to.

End of Document