Getting "Out Front" of WebSphere

The HTTP Server Plugin

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

Donald C. Bagwell
IBM Washington Systems Center
Gaithersburg, MD
301-240-3016
dbagwell@us.ibm.com

This document can be found on the web at:
http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/PRS1467

Version Date: November 21, 2006
See page 35 for the document change history
Many thanks to Louis Wilen and Lee-Win Tai of the IBM Washington Systems Center for their help understanding the new features of this Plugin. And Mike Kearney and Bob Teichman for security-related issues.
What This Presentation Is About

The focus is on the HTTP Server "Plugin" that comes with WebSphere for z/OS V6:

The focus of this presentation is the HTTP Server "Plugin" that comes with WebSphere for z/OS Version 6. This code is called a "Plugin" because it runs inside the HTTP Server -- it "plugs into" the API structure of the HTTP server. So in this presentation we'll introduce the Plugin, provide some background on its main role -- "session affinity" -- and then get into a discussion of how the Plugin is configured and used.

Note: This presentation is not an exhaustive source of information about this Plugin. This topic is actually a fairly broad one, encompassing many different networking and application disciplines. The purpose of this presentation is to knock down the initial confusion about this new V6 function so you can get the basics configured properly.

Flow of Presentations will be:

- Introduction to the Plugin
- Introduction to Session Affinity
- Configuring Webserver to WAS Admin
- Configuring the Plugin itself
- Using the Plugin

Questions are welcomed!
Each Appserver Has HTTP Listener

In WebSphere, all application servers have their own HTTP listeners. Having each user point their browser to a certain host:port pair is unworkable for a lot of different reasons:

- Use of non-default port value
- No common URL for all users
- Firewall would have to permit all those different ports
- No "DMZ"

Downsides to this:

Answer is to put something "out front" of the WebSphere configuration ...

To start, let's point out that when you configure a WebSphere "Network Deployment" configuration with multiple application servers, each server has its own HTTP listeners. If you were to have end-users point their browsers directly at the listeners on the application servers themselves, it would create an almost unworkable situation:

- Non-default port designations would have to be made on the URLs sent in
- You couldn't have a universal one-size-fits-all URL for everyone
- The firewall would have to be opened for multiple ports
- There'd be no "DMZ" (de-militarized zone; the practice of having two firewalls between the unprotected Internet and the secure WebSphere servers)

This is the case for all WebSphere platforms, not just WebSphere on z/OS. WebSphere's architecture is built on the idea of each application server being somewhat autonomous and possessing its own HTTP listener ports.

Note: For internal applications -- ones behind the firewall -- going directly at the WAS listener port may be acceptable, particularly for low-use applications.

The answer to these issues is to put something out front of the WebSphere application servers. The next chart shows what that provides.
Common "Out Front" HTTP Device

By placing an HTTP device out front of the application server configuration, the problems of the previous chart are addressed.

When a common HTTP device is placed out front, it offers a solution to the problems addressed in the previous chart: a DMZ can be set up; the primary firewall can be tightly controlled (with only one port punched through it); users in the field can use one common URL to access any of the servers in the background WebSphere configuration.

In the real world, virtually all high-volume designs implement some form of front-end HTTP device like this ... most often the design calls for multiple such devices with some very sophisticated load balancing and failover mechanisms. There are lots of solutions in the world for this -- routers, WebSphere Edge Server, Sysplex Distributor. But the one we’ll focus on in this presentation is the HTTP Server Plugin that comes with WebSphere.
Introducing "The Plugin"

The WebSphere HTTP Server Plugin is code that runs inside various web servers: IBM HTTP Server, Apache, IIS, Sun Java System. Requests are passed over to the Plugin, where they’re handled based on a configuration file:

Pass to Plugin?

A little bit to talk about here

Plugin

Pass to Plugin?

A little bit to talk about here

Handle as standard HTTP traffic (HTML, static content)

That's fine ... but what's so special about the Plugin?

The Plugin provides something called "Session Affinity" ...
**Session Affinity Overview, Part 1**

HTTP is a "stateless" protocol, which means that each request is considered separate from others.

Some applications -- not all -- maintain information about a user between requests. This is done by creating "session objects" that contain the information.

![Diagram showing session affinity](chart)

Information is placed in the HTTP flow between browser and server to identify "Fred" -- by default a JSESSIONID cookie with a long hex string containing session ID and server ID.

Note: but not user's name, password or other personal information

Note: other routing methods -- URL rewrite and SSL ID value

Let's do a quick overview of Session Affinity. This will take two charts. We start with a statement of fact: HTTP is itself a "stateless" protocol. What that means is that between each request, there's nothing by default in the HTTP protocol that would allow an application to know that the request just received is related to user "Fred," and that the request is directly related to the previous request from "Fred."

Some applications -- not all, but some -- need to maintain information about users between clicks of the mouse at the user's browser. They do this by creating "session objects" -- Java objects maintained in memory that have information about the user. When a session object is created by WebSphere, WebSphere then modifies the HTTP header and places a JSESSIONID cookie that flows back to the user's browser. That cookie is returned with each HTTP request sent in by the user.

**Note:** The cookie is just a long hex string with the session ID and the serverID, which is known as a "clone ID." The cookies does not contain the user's ID, password, or any information about the user other than the session ID and the server ID. Note also that the use of a JSESSIONID cookie is not the only mechanism, just the default. Two other ways are URL rewriting and using the SSL ID value.

If the picture only had one application server, then the complexity of this would be minimal. Fred would return back to the same server each time, and his session object would simply be there. But what about if the application is in a cluster, where the application resides in multiple servers?
Session Affinity Overview, Part 2

In a WebSphere cluster, the application runs in multiple servers at the same time. But session object will reside only where originally built. How is that handled?

Performance studies confirm that best thing to do is route Fred back server where session object resides

Role of the Plugin is to do just that: route to server based on JSESSIONID information in HTTP

Notes:
- It is possible to store object in relational database and fetch it back. Known as “persistence.” DB read/write overhead.
- It is possible to copy object JVM memory-to-memory. Known as “domain replication.” Better, but still overhead.
- Best is to route back to original server.
- Combination of both offers failover advantages

How the plugin does this routing is covered next ...

When an application is installed into a WebSphere cluster, the application will be deployed to all the “members” (servers) of the cluster. So in the example above, the cluster consists of two copies of the application, one in each cluster member. When a session object is created, the object resides in the JVM memory of one of the servers, but not both.

Note: But it’s important to point out that it could reside in both. There are mechanisms to copy session objects between JVM memories. This is known as “domain replication.” It’s also possible to store the session object in a database and fetch it back in the other server. So there are all manner of sophisticated ways to make objects available elsewhere. But they all suffer from the same thing -- performance overhead.

When a session object resides in one server but not both, then it makes the most sense to insure that subsequent requests go back to the same server. Performance studies bear this out. It also alleviates the need for applications to exercise routines to recreate the object or fetch it from a persistent data store.

This is exactly what the Plugin is designed to do -- understand that a request is related to an already-created session object, and route the request back to the server in which that data object resides. How the Plugin does that is covered next.
The Plugin's config file contains information about the back-end WebSphere configuration, what cluster exist and what servers are in each cluster, and what URIs are to be routed to which clusters. It's a rather complicated XML file on the surface, but once you get into it a bit, it's not so bad. Rather than dwell on the specifics of that, let's focus on the very heart of it, which is the `<ServerCluster>` block of XML. It's here that the members of a cluster are defined, and for each the "CloneID" -- the unique identifier for that server -- is defined. The JSESSIONID also carries the CloneID value, and this is how the Plugin matches a request up with the server to which it is to go.

So when a request comes in and a JSESSIONID cookie exists, the Plugin will see the value of the cookie in the HTTP header and attempt to match it up against the defined CloneID values in the XML. If it gets a hit -- as is illustrated here -- then the Plugin knows that the request is to be routed to the server defined on the `<Transport>` definition. The `<Transport>` definition has the host IP name and port on which the application server is listening.

Initial requests received by the Plugin will carry with them no JSESSIONID cookies. What does the Plugin do with those? It does a round-robin distribution based on "weights" you can set. (The weights are not shown in the XML above ... that was to keep the chart simple.)

**Note:** The XML is much more than what's shown above. This was a very simplified view intended to introduce you to the key issue of routing based on the CloneID. We have a lot more to discuss.
New to V6 -- Server Type of "WebServer"

In Version 5, the WebSphere configuration did not know about the webserver where the plugin would run. It assumed it was "out there somewhere," and that was all.

With V6, the webservers are defined to (not created by) the WebSphere cell:
(Act of configuring up httpd.conf file and JCL proc for webserver separate; once done, you let WebSphere know about it)

This provides several benefits:

- Limit plugin-cfg.xml definitions to only those apps that come through that webserver
- Automatically propagate updated plugin-cfg.xml to webserver
- Start / stop webserver from Admin Console

In the past WebSphere had no configuration knowledge of webserver. Now it does. In the past these things were manual. Now you can make them more automatic.

Next: "managed" and "unmanaged" nodes ...

Before WebSphere V6, the HTTP Server was used to route traffic to the WebSphere application servers, but the WebSphere administrative function knew nothing about the HTTP Server. It was just "out there" somewhere. In Version 6 of WebSphere we have a new type of server displayed on the Admin Console. It's called "Web servers," and what it does it provide a way to associate a webserver with the WebSphere cell.

Note: But it doesn't provide a way to actually create the HTTP server. That's still a separate process. What this does is provide a way to associate an already-existing webserver with the WebSphere runtime.

This is not simply window-dressing ... there is value in this:

- One of the things this allows WebSphere to do is limit the amount of information in the plugin-cfg.xml file to just that required for the applications "mapped" to this webserver (more on this process in a bit). For large WebSphere configurations with many different applications, this keeps the XML file from becoming excessive.

- This also provides the opportunity to automatically copy the regenerated plugin-cfg.xml file out to the webserver's directory. If the webserver is IHS V6 on a distributed platform, this is done via a special administrative process in IHS. For IHS on z/OS, the propagation is only possible if the webserver is defined in a "managed node" (more on this in a bit). There, propagation is really nothing more than the standard node synchronization.

- Finally, it's now possible to start and stop the webserver from the WebSphere admin console.

A lot of this depends on how the webserver is configured and related to the WebSphere runtime. There's a concept of a "managed" node and an "unmanaged" node that needs to be covered next.
Nodes -- Managed and Unmanaged

WebSphere requires that the webserver be associated with a node. With V6 there are two types of nodes: "managed" and "unmanaged". Difference is Node Agent.

Managed Node
- Has Node Agent
- DMGR can "manage" webserver:
  - synchronize plugin-cfg.xml file out
  - start/stop the webserver
  - edit the webserver configuration file

Unmanaged Node
- DMGR can't "manage" webserver
- Manually copy plugin-cfg.xml to webserver
- Exception: IBM HTTP Server V6 on distributed:
  - Has special "Administration Server"
  - DMGR communicates directly with that
  - Provides function like Node Agent

"Unmanaged" = no Node Agent

Let's look at a few examples ...

With WebSphere for z/OS Version 6, we now have the notion of nodes being "managed" or "unmanaged." It all revolves around the node having -- or not having -- a Node Agent. If the node has a Node Agent, then it's "managed." The webserver may reside within either type of node -- managed or unmanaged -- and the capabilities to control the webserver are determined by which kind of node the webserver is "in."

**Note:** What really happens is you associate an already-existing webserver with a node. You do this through the Admin Console. If the webserver is running on a system that has a Node Agent, then it's possible to associate that webserver with a "managed" node. But if the webserver is not running anywhere near a Node Agent, then you may need to define an "unmanaged" node (no Node Agent) and associate the webserver with it.

**Managed Node** -- if the node in which the webserver is associated has a Node Agent, then the Deployment Manager has a way to act upon the webserver, or "manage" it. This includes the ability to start and stop the webserver and even to edit the webserver's configuration file. All this is done through the Node Agent -- if the Node Agent isn't running, then these capabilities aren't present.

**Unmanaged Node** -- here the webserver runs where no Node Agent is present, such as a box in the DMZ. With no Node Agent, the Deployment Manager has no real opportunity to affect the webserver. So it's like it was in the past -- manually copying the plugin-cfg.xml file out to the webserver. The exception to this is the IBM HTTP Server V6 on the distributed platform, which has an "administration server" that provides function similar to a node agent. The HTTP Server on z/OS does not have this "administration server" capability.
Let's take a look at two examples of this on non-z/OS platforms.

- On the left side of the picture we have an OEM webserver (such as Apache, or IIS) on a server box somewhere. Since there's no Node Agent running on the box, the node that's defined to WebSphere will be an "unmanaged" node. The webserver is then associated with the unmanaged node. Here the XML file for the plugin is copied out manually, and any starting and stopping of the webserver must be done manually as well. There is, in other words, little "management" of the webserver through the Admin Console of WebSphere.

- On the right side of the picture is a notable exception to this -- the case where the webserver is the IBM HTTP Server V6 on the distributed platform. Since no Node Agent is present, it would still be an "unmanaged" node. But here you have the opportunity to use a function of that webserver to provide some management capabilities. Because that version of the IBM HTTP Server has an "administration server," WebSphere has the ability to communicate with the HTTP Server and manage it. Here you can automatically copy the plugin-cfg.xml file out to the HTTP Server, start and stop it, and edit the httpd.conf file.

Note how both scenarios had the webserver in an unmanaged node. And note how in both cases the unmanaged node had to be defined to WebSphere and the webserver associated with the unmanaged node. All this is very important because it provides a way to tell the WebSphere configuration that the webserver exists.

Now, what about the case of a z/OS environment?
Example z/OS Topologies

And two examples of how this might be done on z/OS ...

**Unmanaged Node**
- Node: Unmanaged node defined to WebSphere
- Webserver associated with it

**Managed Node**
- Node: "Empty" Managed Node
- Webserver associated with it

Here are two z/OS examples:

- The picture on the top shows a WebSphere cell in one LPAR, and the IBM HTTP Server for z/OS in another LPAR. Since no node agent exists in that LPAR, the webserver is associated with an "unmanaged node" that you create through the Admin Console. Here you have no management of the webserver -- you must copy the plugin-cfg.xml file manually, and you must start and stop the webserver by hand.

  **Note:** This may seem like "business as usual" from V5, but there's a difference. You have to define the unmanaged node (new in V6), you have to define the webserver and associate it with a node (new in V6) and you have to map applications to the defined webserver (new in V6).

- The picture on the bottom shows two LPARs, but the one on the left has a Node Agent, which means that the cell managed by the Deployment Manager extends over to the LPAR on the left. The HTTP Server in that LPAR is then associated with the "managed" node. This enables the Deployment Manager to "manage" the webserver through the Node Agent.

We've mentioned on several occasions how "managed" and "unmanaged" nodes are defined to WebSphere. That's a necessary prerequisite to associating a webserver to a node. Let's see how nodes are defined.
Defining Nodes to WebSphere

In all cases, the webserver must be "associated with a node." How are nodes defined to WebSphere so you can associate the webserver with it?

**Managed Node**

Business as usual:
- Standalone Server and Federate
- Empty Managed node

Both cases ISPF panels are run, custom jobs created and executed

This is the standard method of building managed nodes

**Defining an Unmanaged Node**

System Administration ▹ Nodes

Managed node contains a WebSphere node agent process which results in running the add node.

Unmanaged node:
An unmanaged node management where a node is in the topologies

Host Name

Platform Type

Provide node long name, IP host where it's located and the platform type

Simply creates unmanaged node structure so webserver may be associated with it

Let's see how webserver is "associated" with a node ...

There are two kind of nodes to WebSphere -- managed and unmanaged, as we've said before. Here's how they're defined:

- **Managed Node** -- this is done in the same way we've done it on z/OS since the days of V5 -- we run through the ISPF panels, create a series of customized jobs, and we run those jobs. That process creates the node and then the node is "federated" into the cell. It's at that point -- the point of federation -- that the Node Agent is built. V6 introduced a new type of node -- an "empty" node -- but all that really is a node with a Node Agent and no application servers. The point here is that a managed node is created in the same way you've create a node to hold application servers.

- **Unmanaged Node** -- this is done through the Admin Console. No customized JCL jobs are created. The Deployment Manager simply creates a node structure in its "Master Configuration" to represent this "unmanaged" node. In the Admin Console you simply go to "Nodes" and click on the "Add Node" button. You're given a choice of "managed" or "unmanaged."

**Note:** Adding a "managed node" implies federating in a "Standalone Application Server node" that already exists and is running. On z/OS you can't avoid the ISPF panels to create the essentially building blocks of a node. If you have a Standalone server present, clicking the "Add Node" allows you to specify the host and port of that Standalone server, which then invokes the `addNode.sh` shell script which federates that node into the cell.

You simply supply a name, a host IP address and the platform type. The node structure is built in the HFS and the Deployment Manager is happy.

**Note:** The host IP address is used to ping the webserver to see if its up.

Let's now see how a webserver is associated with a node -- managed or unmanaged.
Defining Webserver to Node

Once you have the node defined -- either managed (with Node Agent) or unmanaged (prior panel) -- you may then define webserver and associate it with the node:

Once the nodes are in place -- managed or unmanaged -- you may now define (or "associate") a webserver to the node.

**Note:** The creation of the webserver is a separate task. It is *not* created through the Admin Console.

It's a very simple process:

- Under "Server" click on "Web servers," then click on "New"
- Under "Step 1" select the node (managed or unmanaged) and provide the webserver name.
  
  **Note:** This is why defining the node first is important -- it then appears in this dropdown list.

- Then you indicate the type of server, the port it listens on for HTTP, and two directories where configuration files are kept.
- The last two steps are simple validation point-and-click steps ("Are you sure?" type of things). Then save and synchronize as usual.

What you get is this ...
What Admin Console Shows

Assuming the webserver was already started -- with or without the Plugin configured -- here’s what you’ll see:

The green arrow simply means a test request to the webserver’s HTTP port was answered. The webserver is up. It says nothing about the Plugin’s status.

By clicking the link that represents the webserver, we can set properties related to that webserver, including things for the plugin-cfg.xml file...

After you’ve defined the webserver and associated it with a node, you’ll now be able to see the webserver in the Admin Console, under "Web servers." It’ll show a line for each webserver you defined, along with the node it’s associated with, the version of the node (not the webserver, but the node itself) and the status of the webserver.

Note: The green arrow is an indication of the HTTP Server’s up-status, not the status of the plugin running inside the webserver.

The webserver name is represented as a link, and by clicking on this you can get access to the properties of the webserver. This is important because you’ll very likely want to set properties different from the default settings. Let’s look at them now.
General Properties of the Webserver

From the "General Properties" panel you have access to some useful settings:

- **Web server name**: geweb
- **Type**: IMS
- **Use a secure protocol**: [ ]
- **Port**: 8080
- **Installation path**: /u/bagwell/geweb
- **Configuration file name**: Edit

Note assumption that httpd.conf is under /conf directory. Change here if not.

Click here to edit the httpd.conf file

Let's look more closely at the "Plug-in Properties" ...

When you first click on a webserver's link, you get the "General Properties" of the defined webserver. Of note on this page are the following:

- The "configuration file name" for the webserver server is assumed to be httpd.conf under the /conf directory, which is under whatever value you supplied for the "Installation Path." If that is where your webserver configuration file is located, then great; but if not then you should change it here.
- The "Log File" link lets you set where the webserver's access log and error log are located.
- The "Configuration File" link is a way to edit the webserver's configuration file -- provided it's in a managed node or it's an IBM HTTP Server V6 webserver. (Webservers in unmanaged nodes that aren't IBM HTTP Server V6 servers can't be edited.)
- The "Plug-in" properties file is what we wish to explore now. That's where we can define information about the plugin that runs inside the webserver.
Plugin Properties

When a new plugin-cfg.xml file is generated for the web server, these settings will be used ...

Where the Plugin's configuration XML file resides

The name of the XML file that will be generated

Interval at which the plugin will see if XML has been changed and refresh itself

Plugin log file location and level of logging

The "Plugin Properties" panel allows you to specify information about the Plugin that resides inside the webserver. Let's walk through the settings here:

- The "Plug-in installation location" defines where the plugin-cfg.xml file is going to reside. If IHS V6 on distributed, that file may be automatically copied out to the webserver. So it'll need to know where the file is to go.

- The next line defines what the plugin's configuration file name will be. The default is plugin-cfg.xml, but you may change that to be whatever value you wish.

- The first checkbox determines if the plugin-cfg.xml file is to be re-generated each time some change occurs to the environment that requires an update to the file. You may wish to control this manually; if so, then uncheck this box. But if you want WebSphere to update the file in its master configuration, then check this box. (See the explanation after this bullet list for what changes trigger a regeneration of the file.)

  **Note:** The act of copying the file out to the webserver is a different thing ... that's next.

- The second checkbox determines if the plugin configuration file is to be copied out to the webserver. This is only applicable for IBM HTTP Server V6 running on a distributed box and the "Administration Server" function is set up and running.

- The "Refresh configuration interval" is a property that finds its way into the plugin-cfg.xml file, and determines how frequently the plugin itself will take a look at the plugin-cfg.xml file to see if it has been changed. The default is 60 seconds, which means every minute the plugin will check.

- The "Plug-in logging" section defines where logging will occur, and to what level it will be done.

As for the "automatic regeneration of the plugin-cfg.xml file, what things trigger the regeneration? The following list tells the story:

By default it assumes Plugin log is at /logs directory under Plugin location. Make sure this points to valid directory.
• deployment.xml or serverindex.xml changed
  This would include things like adding an application, or changing a port value. This would trigger automatic plug-in config file generation and propagation for all web servers defined in the corresponding node.

• server.xml changed for a web server
  For instance, if you made a modification to a web server definition in WebSphere. This would trigger automatic plugin configuration file generation and propagation for the corresponding web server.

• security.xml changed
  For instance, if you turned global security on or off. This would trigger automatic plugin configuration file generation/propagation for all web servers defined in the cell.

• virtualhosts.xml changed
  For example, if you added a virtual host alias, or modified one. This would trigger automatic plugin configuration file generation and propagation for all web servers defined in the cell.

The next thing to do is make sure applications are mapped to the appropriate webserver ...
Mapping App Modules to Webserver

In order to get application properties into `plugin-cfg.xml` file, it's necessary to map the webapp modules to the webserver as well as the application server:

When you get to the point in an application install where you select the server (or cluster), use the Ctrl key to select it and the webserver.

Other points:
- You can map already-installed apps to new webserver through Admin Console ("Map modules to servers")
- You can map same application to multiple webservers
- If you don't map application to webserver, information about that application won't appear in `plugin-cfg.xml`

In WebSphere for z/OS Version 6, we have a new ability to map applications at installation time to multiple servers. In the past we were able to map it to a server or a cluster, but now you can use the Ctrl key to select multiple targets for an application installation.

The reason why this is important is because if your users are coming through a webserver with the plugin, you'll want to make sure the `plugin-cfg.xml` file has knowledge of the applications in the server. Back in V5 this was accomplished by generating a single `plugin-cfg.xml` file for the entire cell, which contained information about all servers and all applications. With V6, the `plugin-cfg.xml` file is more selective; showing only information for those servers and applications that are "mapped" to the webserver.

Note: If you don't map applications to the webserver, the `plugin-cfg.xml` file is very sparse and won't properly route requests to the application. You can work around this by generating a "cell wide XML" file, which we discuss at the end of this paper.

You may map applications to webservers and application servers either at the time of installation, or after the fact. You may also map applications to multiple webservers if you have several of them out front serving different groups of backend servers.
Generating plugin-cfg.xml File

Generating the plugin-cfg.xml file is a simple process ... select webserver and click button. But where does it go?

In all cases ...

Managed or unmanaged node long name

The webserver's name

(See note about z/OS managed node)

If IHS V6 on Distributed

1. The location it'll write it out to is specified in the "Plugin Properties" panel. The directory and file name are specified in the fields indicated here.

2. The file will be written out only when it changes, so this first checkbox needs to be checked to automatically generate the file when some configuration change takes place.

Next: configuring plugin ...

Note: In the case of a managed node on z/OS, the file is also written out to the node's directory when synchronization occurs. So the plugin-cfg.xml file would exist -- after node synchronization -- in two places: in the DMGR's master configuration (under the webserver's directory) and out in the node's directory structure (under the webserver's directory). If your z/OS HTTP Server is configured to point to this location, you can have the plugin-cfg.xml "automatically refreshed" when synchronization occurs. The plugin will, by default, refresh its memory copy of plugin-cfg.xml every 60 seconds.

In the case of IHS V6 on a distributed platform box, the plugin-cfg.xml file may be automatically written out to the webserver's directory structure. In the picture above, follow the numbered blocks:

1. The location it'll write it out to is specified in the "Plugin Properties" panel. The directory and file name are specified in the fields indicated here.

2. The file will be written out only when it changes, so this first checkbox needs to be checked to automatically generate the file when some configuration change takes place.
3. The next checkbox determines if the newly generated file will be copied out to the webserver directory. If this is checked and the previous box is checked, the newly generated file will be written out to the "installation location" directory.

4. Finally, the "Refresh configuration interval" is used to put a property in the plugin-cfg.xml file so the plugin itself knows how often it is to check for new files written to its location. The default is 60 seconds, which means the plugin will "wake up" and check the plugin-cfg.xml file every minute to see if a new copy is there.

Let's now look at how the HTTP Server for z/OS is configured.
Configuring Plugin in IHS on z/OS

Essentially just the adding of three directives to the webserver's httpd.conf file. Key is pointing to copy of ihs390WAS60Plugin_http.so file ...

Where is executable module found? Every node has a "copy" --

Not quite ready to start server ...

Finally we come to the point where we can talk about how the plugin itself is configured. In short, it's done the same was as done in the past -- by directives coded in the httpd.conf file. The executable file name is new with V6, but the process is otherwise essentially the same:

- A ServerInit statement is used to point to the plugin executable. This tells the HTTP Server to load the executable at startup time. You'll have one ServerInit statement for the Plugin. The <path> value should be set pointing to the location of the file, which we'll discuss in a bit.

- The ServerInit statement takes one parameter -- a pointer to the plugin-cfg.xml file. This file can exist anywhere in the directory structure, but the pointer you specify here should be equal to the value you specified on the "Plugin Properties" panel of the Admin Console.

- The Service statement is used to pass requests over to the plugin. It does this by matching the received URI against a pattern string on the Service statement.

  Note: The "URI" is the part of the URL that comes after the host:port portion of the URL.

So, for instance, if the URI received for an application is /MyIVT/index.html, a Service statement with /MyIVT/* will be sufficient to "catch" the URI and pass it over to the plugin.

  Note: You will likely have many Service statements -- one for each application you wish to catch and throw over to the Plugin. This is a manual process. So if you add a new application, don't forget to add a Service statement.

- The ServerTerm is the opposite of ServerInit -- it tells the webserver to gracefully shut down the Plugin when the webserver itself is coming down.
Where are these modules to be found? Each node has a "copy" of it. The word "copy" is in double-quotes because what each node really possesses is a symbolic link that points back to a single copy of the module, which resides in the SMP/E HFS (or "product HFS" -- the hlq.SBBOHFS data set that was created when you installed WebSphere). It doesn't really matter which copy of this you point to, but it should be one that the webserver has access to. By default the permissions on that symbolic link will be 777, so you won't be blocked by those. But if you're using unshared HFS (meaning not accessible from all LPARs in the Sysplex), then you should make sure you point to a copy of the file on an HFS that the webserver can access.

What about the case of an unmanaged node, where the webserver is on an LPAR that doesn't have access to any HFS that has the symbolic link that points to the file? Well ... then you'd have to copy the file over to the LPAR.

There's a few more things to do before you start the webserver ...
Set Server "Trusted Proxy" to True

On z/OS, you need to tell WebSphere that it's okay for the server to accept private HTTP headers sent by the Plugin. This is done with TrustedProxy property.

A unique aspect of WebSphere on z/OS is that you may need to set the TrustedProxy property to "true" to allow the things to work. This is not a requirement 100% of the time -- it's really only needed for two reasons:

1. If you're using SSL client authentication. Here the client certificate will be passed from the Plugin back to WebSphere in private headers, and this property is needed so the application server can "trust" that the proxy (the HTTP Server and the Plugin) properly authenticated the client.

2. If the application needs to see the client IP address in the HTTP headers coming up from the Plugin. If TrustedProxy is off, those IP addresses are not visible.

This is one of those things where a definitive "turn it on" or "leave it off" statement can't be made. Turning it on will break SSL client authentication for those not coming through the plugin (in other words, directly at the HTTP port of the application server). Leaving it off will prevent applications from seeing the IP address of the clients, which is sometimes used as an additional security measure to ensure the same client is coming in each time.

In any event, the process for setting the property is shown above. It is done for each HTTP Transport -- non-SSL and SSL -- for each application server.
Webserver NOT on Port 80?

If your webserver is listening on something other than port 80, you'll probably have to add an "alias" to the default_host virtual host:

By default you'll have an alias for port 80, so if your webserver is listening on that port, you're covered

Now you can start the HTTP Server and test ...

Most production webservers listen on the default port of 80. This is done because browsers out in the world by default will point themselves at the default HTTP port of 80. In anticipation of this, WebSphere establishes a "Virtual Host Alias" with port 80. So if your webserver is listening on port 80, then you need not do anything with the Virtual Host Alias definitions in WebSphere.

However ... if your webserver is listening on something other than port 80, you'll have to add a Virtual Host alias to the WebSphere definition. For example, imagine a test environment where your webserver is listening on port 9999. That port will not be default be in the Virtual Host Alias list of WebSphere (where port 80 is there by default). So it would involve going into the Admin Console and adding a Virtual Host Alias for port 9999 (most likely with a "Hostname" value of asterisk, which means any host name value will be accepted), then stopping and restarting the application server.

Again, if your webserver is listening on port 80, you're covered. That alias is there by default.

You're now ready to start and test your HTTP Server.
Starting HTTP Server and Plugin

Same as before ... start HTTP Server and the Plugin starts automatically. Look for "smiley face" for indication of successful initialization of plugin:

In webserver SYSOUT ...

WebSphere HTTP Plug-in for z/OS and OS/390  Version 6.0 build level cf10515.05 release WAS601.ZNATV date 04/15/05 12:55:41 is starting

WebSphere HTTP Plug-in for z/OS and OS/390 initializing with configuration file: /u/bagwell/g6web/plugin-cfg.xml

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

Tells you what level of Plugin module is loading. Good way to validate you didn't accidentally point to downlevel code

Tells you location and name of configuration file that's being loaded

"Smiley face" tells you initialization went okay

Lots of simple errors can keep Plugin from initializing. We'll get to those in a minute.

Quick test of plugin operations ...

The HTTP Server is started in the way it always has been, and the Plugin starts automatically based on the ServerInit statement. The way you validate that the Plugin comes up properly is to look in the SYSOUT for the HTTP Server, and specifically to look for the "smiley face." If you see that, then you know the Plugin came up okay.

Note: There's a slew of things that can prevent the smiley face from appearing. We'll cover those in a minute.

Once the smiley face is found, then you can also then derive some other useful information. The version of the Plugin code is also displayed ... this will help you make sure you're using the same level of code as is being used by the application servers. The plugin's configuration file is also highlighted, so from this you can validate that the parameter on the ServerInit is being understood and employed by the Plugin.

Before we dig into problem determination methods, let's take a quick look at some tests to validate the proper operation of the Plugin.
Testing Connection to Application

After the plugin has initialized, you may test out the connection to the backend application servers. Here's a basic approach:

First, drive a request directly at the application server port
This will validate that the server is up and the application is running

Second, drive an HTML request to the webserver
This will validate that the HTTP Server is listening on that port and is capable of handing requests

Third, drive application URL at HTTP Server
This will validate Service statement and plugin-cfg.xml statements

Once you have a smiley face, you know the Plugin is up. But before you turn the system over to production users, there’s a few things you can do to test some basic operations.

1. First, from inside the firewall, shoot a request directly at the HTTP port of the application server to make sure the application server will server out the application. This will validate that the application is up and running and responding to the URL as you think it should be.

2. Next, drive a request at a known static HTML page (not the application request) in the webserver. This will validate that the HTTP server itself is listening on the expected port and will issue out pages.

3. Finally, issue the URL that requests the application. Point this request at the webserver. This will validate the ability of the webserver to pass the request over to the Plugin (proper Service statement operation) and the proper configuration of the plugin-cfg.xml file itself.

**Note:** Here's where mapping applications to the webserver is so important. That's what results in the proper definitions finding their way into the plugin-cfg.xml file when it's generated.

What these steps do is provide a basic set of validation routines. Let's look at a few common problems that can result.
Common Problems You May Encounter with Plugin Initialization
Three Common Errors

Three main categories:

---

**Typo pointing to was ihs390WAS60Plugin_http.so file**

Typo in pointer to module

Failed to load DLL module ...

No smiley or frowny ... search on “Failed to”

---

**Problem Accessing plugin-cfg.xml file**

Typo in pointer (file doesn't exist) or permissions problems:

Failed to stat plugin config file: /<path>/plugin-cfg.xml

initialization FAILED (rc = 3) :-(

RC=3 means file not found (see next item)

---

**Bad contents of plugin-cfg.xml file**

Manual modifications made to plugin-cfg.xml and mistake made so file can't be parsed:

Failed parsing the plugin config file

initialization FAILED (rc = 4) :-(

RC=4 means file found but problems inside

Look in Plugin log for indication of where in file problem exists

Only LOG=ERROR needed; it's pretty good about telling you which line had problem)

---

We'll focus on three very common errors people will encounter when first setting up their webserver. All three relate to the Plugin failing to initialize.

- The first occurs when the HTTP Server fails to access the ihs390WAS60Plugin_http.so file. If the HTTP Server can't access that executable module, then it can't load the plugin. The most common problem here is a typo in either the path to the file, or the file name itself. The symptom you see here is a no smiley face nor any “frowny face.” What you should do if you find neither of those indicators is to search on the string “Failed to.” The presence of that is a good indication that the plugin wasn't initialized or a clear failure, but rather simply not loaded.

- The second common problem occurs when the plugin-cfg.xml file can't be accessed. There's typically two reasons for this: either the pointer to the file is wrong (typo somewhere), or the file was copied over and the permissions won't allow the HTTP Server to access the file. The symptom you see here is a “frowny face” in the SYSOUT of the HTTP Server and a (rc = 3) indicator. That RC=3 indicator is telling -- it means the file itself wasn't found. That's contrasted with RC=4, which we describe next.

- The third common problem occurs when the plugin-cfg.xml file is found, but there's a syntax problem inside the file. This will not likely happen if the plugin-cfg.xml file is generated by WebSphere. But it may occur if you make modifications to the file after it's been generated (to add some specific thing to the file). RC=4 will be thrown, and this means “file found, but problem inside file.” The plugin log will tell you which line had the problem. The logging seems to have been improved since V5 and the message it gives is really rather specific and good.
Understanding the ISPF Dialog Webservice Options
ISPF Panels You Encounter

When configuring a node in the ISPF customizaton panels, you'll encounter a panel that looks like the following:

```
Web Server Configuration (1 of 1)

Web Server Name: webserver1
  The name used in defining the web server in the admin console.

Host.......... : wsc3.washington.ibm.com
  Hostname of the system on which the web server is located.

Port.......... : 80
  Port on which the web server is listening.

Application Mapping ((A)ll, (N)one, (D)efault): A
  Determines whether you want to map all, none, or the Default Application to the web server.

Deployment manager security is enabled: N
  User ID.............: G6ADMIN
```

**Key Points:**

- Creates *optional* job (`BBO*CFGW`)
- Job is batch WSADMIN script
- Everything it does you can do through Admin Panels
  
  *Exception: baseapp server ... must use batch job from panels for that*

Let's see what this thing does ...

---

**Important!**  This panel has been removed from the ISPF panels starting with V6.0.2.4.

If you have any experience with the WebSphere for z/OS Version 6 ISPF panels, you've probably encountered a panel that looks something like this. This panel is seen when configuring a Deployment Manager, an Empty Managed Node and the Federate Node process. If you're like me, you've probably just skipped over this panel. That's okay ... this panel is not required to make things work. Here are the key points:

- It creates a customized job -- `BBO*CFGW` -- that is optional. (The asterisk means that placeholder in the name changes, depending on which configuration path you took when the job was created.)
- What the job does is invoke WSADMIN (the WebSphere scripting tool) which then invokes a script.
- There's nothing this job does that can't also be done through the Admin Console.

**Note:** The one exception to this is when the WebSphere configuration is a Standalone Server. There, the Admin Console does not have an "add" button to define a new webserver. So using the ISPF Panels and the generated customized job is required.

Let's see what this thing does, then you can better determine if you ever want to configure this panel and execute the job.
What BBO*CFGW Does

The BBO*CFGW customized job -- the * represents a different letter for each of the four types of configuration where the job is created -- is a batch invocation of WSADMIN:

- Invokes WSADMIN
  - May require DMGR to be running ... see instruction member
- Calls script configureWebserverDefinition.jacl
- That script does the following, based on parameters defined on panel:
  - Creates an unmanaged node definition
  - Creates webserver definition and associates it with unmanaged node
  - If "All" selected for mapping applications, it loops through listing of all installed applications and creates mapping to new webserver

Use if ...
- You intend to use unmanaged node
- You have lots of applications already defined to WebSphere
- Standalone Server configuration

Don't use if ...
- You wish to use managed node
- You wish to see how the creation process is done through Admin

Important! The BBO*CFGW job was removed from the customized jobs starting with 6.0.2.4

The BBO*CFGW job does the following things, based on the information provided in the ISPF panel:

- It invokes WSADMIN
- It calls the JACL script configureWebserverDefinition.jacl, which is used by WSADMIN as the logic to do the webserver configuration.
- The script then does the following:
  - It uses JACL to create an unmanaged node definition in the cell. You may or may not need to have the DMGR running when you run this. You should read the instruction member that's written out to the CNTL data set.
  - Notice that this creates an unmanaged node. Managed nodes are created by federating a node into the DMGR cell.
  - It then creates a webserver with the host and port values you provided in the ISPF panel. It then associates the webserver with the unmanaged node created in the first step.
  - Finally, if "All" is specified for the application mapping, the JACL script creates a list of all installed applications and then loops through that list and maps those modules to the new webserver.
Command Line Generation of Plugin XML File
Generating "Cell-Wide" XML File

Using the Admin Console to generate the XML file means the file will only contain definitions for URIs mapped to *just that webserver*:

Generates XML for just that webserver, but not other. Check all boxes and it generates separate XML files/

The `GenPluginCfg.sh` shell script will generate a "cell wide" XML file which may then be manually copied out to a plugin:

```
./DeploymentManager/profiles/default/bin/GenPluginCfg.sh
```

Invoke with no parameters and will create file here ...

Note: shell script has parameters, but if none supplied then get this behavior

```
./DeploymentManager/profiles/default/config/cells/<cell>
```

We've stated earlier that when the `plugin-cfg.xml` file is generated in V6, it generates a file applicable to a given webserver, but not for an entire cell (unless, of course, you have one webserver defined and all applications are mapped to that one webserver). But the capability to generate a cell-wide XML file like we did in V5 is there, but it's only accessible through a batch shell script.

The shell script is called `GenPluginCfg.sh`, and is located in the Deployment Manager's `/bin` directory under `/profiles/bin`. This file can be invoked without parameters and what that'll do is generate a cell-wide `plugin-cfg.xml` file and place it at the location shown in the chart above.

**Note:** Please understand that the shell script actually has a set of parameters that will affect its behavior -- such as where the file will be written. What’s being shown here is the simplest way to invoke it.
Conclusion

- New server definition -- "Web servers"
- Web servers are associated with nodes:
  - "Managed" -- has a Node Agent
  - "Unmanaged" -- no Node Agent
- Applications are now "mapped" to application server and web server
  - This is how information about applications gets into plugin-cfg.xml
- Plugin's primary role is "session affinity," though plugin can be used as a simple forwarding device if clustering not being done
- Multiple web servers allows; applications may be mapped to multiple web servers.

And here are the final concluding comments, all of which were explained earlier in this presentation.
**Document Change History**

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>July 28, 2005</strong></td>
<td>Original document</td>
</tr>
<tr>
<td><strong>August 10, 2005</strong></td>
<td>Corrected section on the ISPF panels. Previous version did not indicate that batch WSADMIN script is required when configuring webserver for a Standalone Server configuration. For a Network Deployment configuration, all things can be done through the Admin Console. But the Standalone's Admin Console is limited and can’t “add” a node or add a webserver. Thus the need to do those things through WSADMIN. The WSADMIN script is created through the ISPF panels.</td>
</tr>
<tr>
<td><strong>March 9, 2006</strong></td>
<td>Corrected two things:&lt;br&gt;• Earlier edition of paper implied that “automatic propagation” of <strong>plugin-cfg.xml</strong> was available for IHS z/OS when defined in a “managed node.” The parallel was drawn between it and the case where IHS V6 on distributed was used. That parallel is not strictly true. When the IHS server is on z/OS -- even on a managed node -- the automatic propagation of the <strong>plugin-cfg.xml</strong> file is limited to what can be done through normal node synchronization. That is, the generated <strong>plugin-cfg.xml</strong> file is first placed in the DMGR's master configuration, then synchronized out to the managed node's configuration HFS. If the IHS for z/OS server is configured to pull the <strong>plugin-cfg.xml</strong> from the managed node's configuration directory, then “propagation” occurs. But with IHS on z/OS the ability to propagate to a “foreign file system” (meaning: not the node's file system) is not available. It is available when IHS V6 for distributed is employed. That has a special administrative process that is capable to doing that.&lt;br&gt;• Prior to 6.0.2.4 of WebSphere for z/OS, the configuration ISPF panels presented a panel for configuring the web server. That resulted in a customed job -- <strong>BBO*CFGW</strong> -- being created. With 6.0.2.4 and beyond, that panel (and job) has been removed.</td>
</tr>
<tr>
<td><strong>November 21, 2006</strong></td>
<td>Added a list of what changes to the cell configuration results in an automatic regeneration of the <strong>plugin-cfg.xml</strong> file. See page 17.</td>
</tr>
</tbody>
</table>