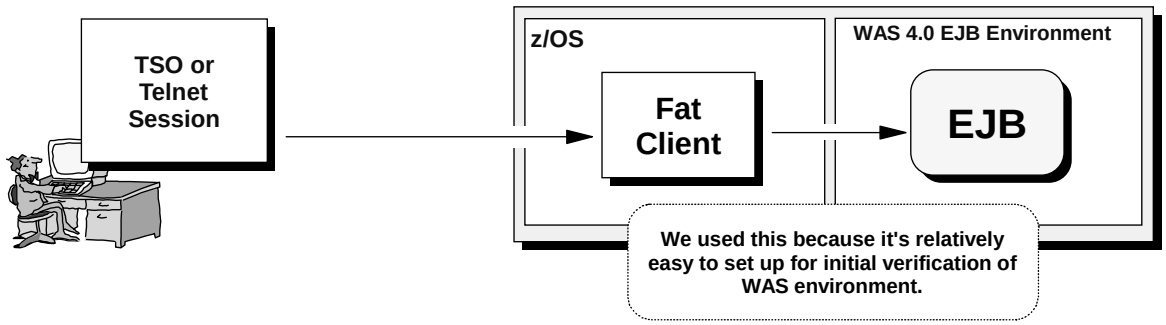
WebSphere Application Server
for z/OS and OS/390

Configuring Web Applications Which Access EJBs

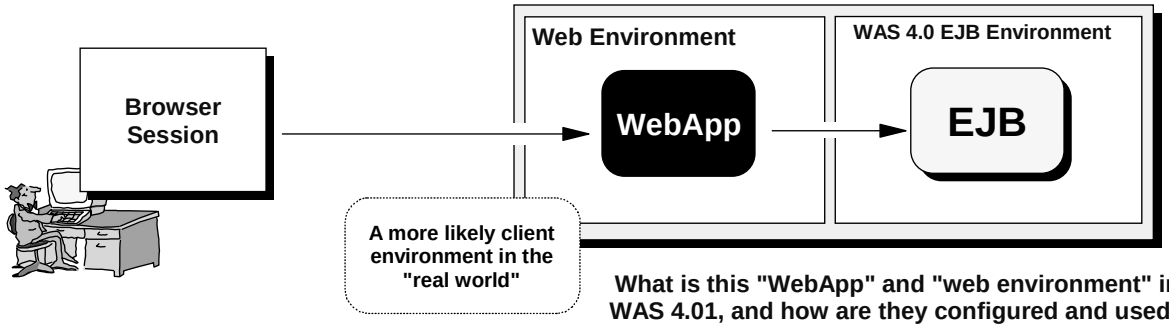
(This page intentionally left blank)

Client Access to EJBs

Client access used so far in this class ...



Now we turn our attention to a different form of client: a "WebApp" running in a "web environment" ...

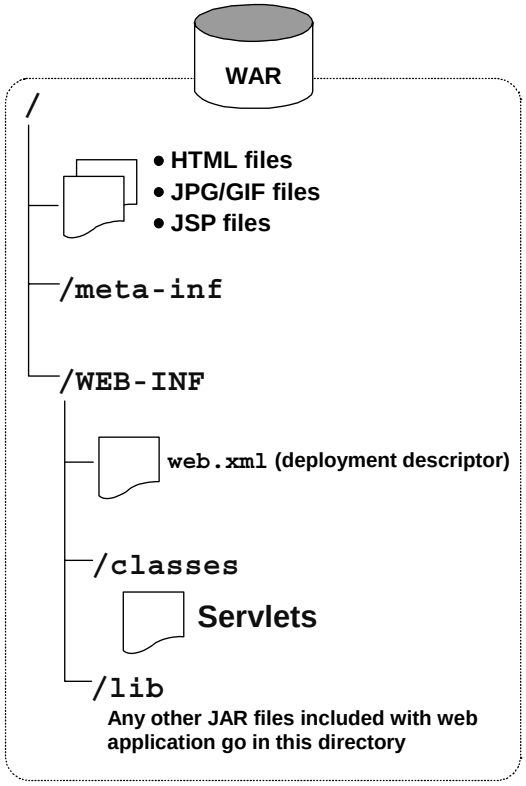


Up to this point in the class, we've used what's called a "fat client" to drive the EJB. We did that because the fat client is easy to set up and run from the OMVS environment. It validated the WAS environment, and that was the objective at that point.

Now we turn to Web Applications. Web applications run in a "web environment" and act as a client to the EJB, just like the "fat client" did.

This presentation will focus on what a WebApp is, how to configure its environment, and how to use them to act as clients to EJBs.

What's a "Web Application"?



Web Applications consist of servlets as well as supporting files such as HTML files, JPG/GIF and JSP files.

WebApps are packaged in WAR files. You built one of these in the previous lab.

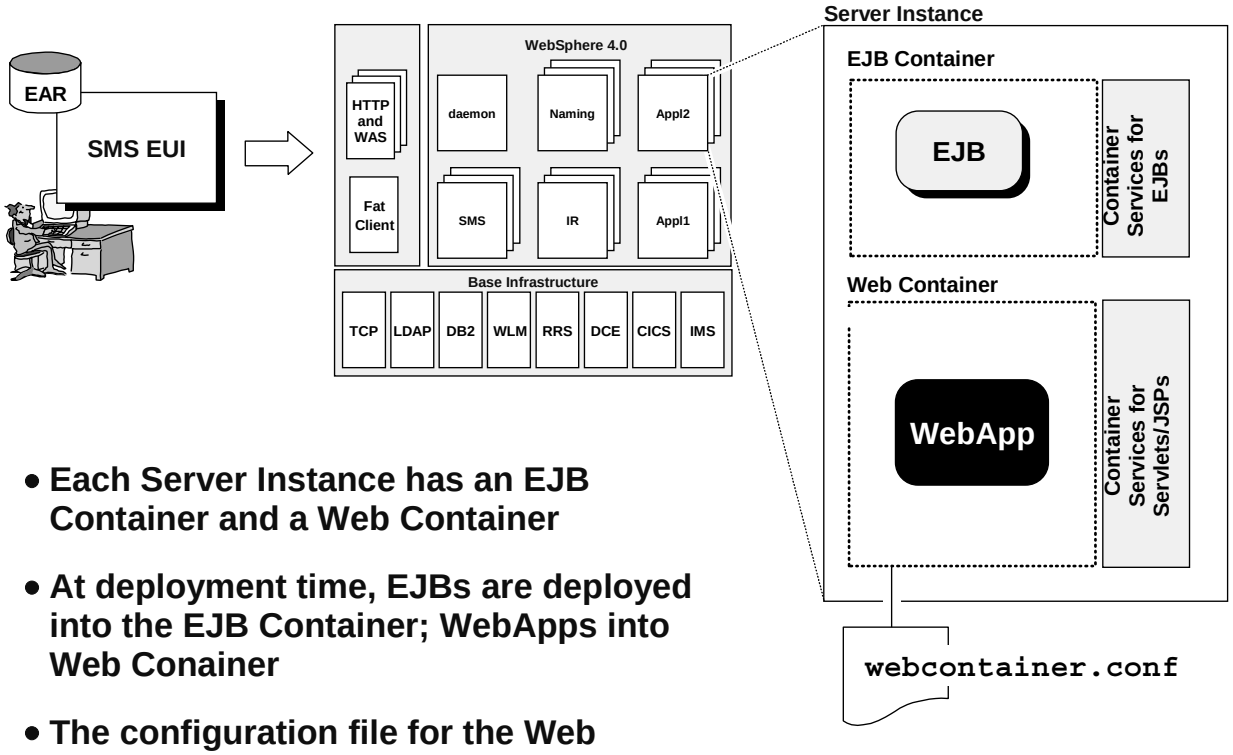
WebApps are not EJBs

Enterprise JavaBeans

Web applications are comprised of servlets (executable Java code, written to the Java 2 Standard Edition servlet specification), as well as static content such as HTML file, and JPG/GIF image files. In the J2EE world, web applications are packaged in a ZIP-format file called a WAR file (Web ARchive). This WAR file has within it a "deployment descriptor" called `webl.xml` that tells the J2EE server about the webapp contained in the WAR file, and how to run that webapp.

Webapps are *not* EJBs. EJBs have other requirements that webapps do not. So while both are Java programs, they are not the same thing.

WAS 4.01 Web Containers



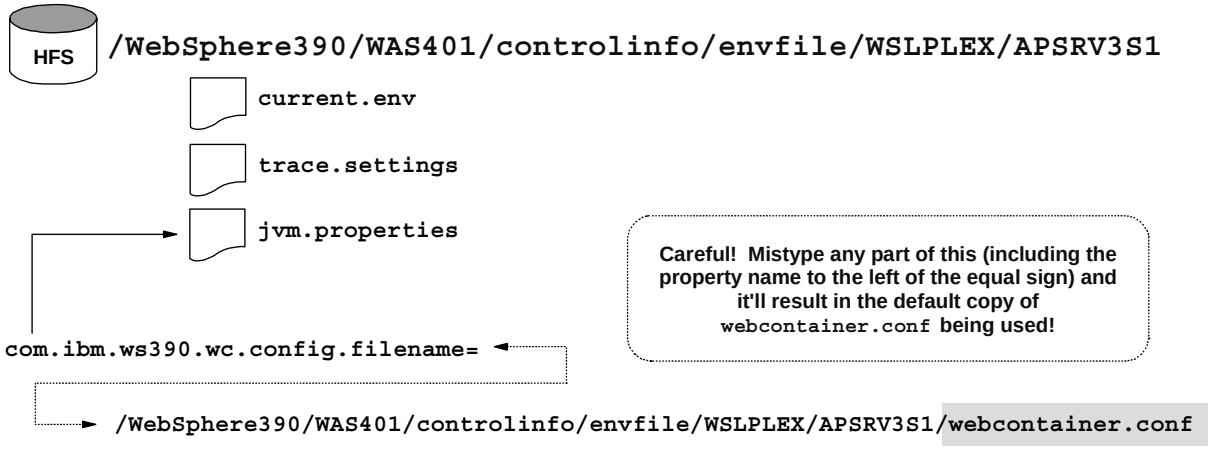
- **Each Server Instance has an EJB Container and a Web Container**
- **At deployment time, EJBs are deployed into the EJB Container; WebApps into Web Conainer**
- **The configuration file for the Web Container is `webcontainer.conf`**

As we discussed earlier in this course, each application server instance is supplied with an "EJB Container" and a "Web Container." These are logical software constructs within the application server, and they provide the services necessary to support the running of EJBs or WebApps. The purpose of their existence is to shield the EJB or WebApp from the underlying complexity of the platform, and to provide a set of common, standardized and defined services to the applications.

When you deploy an application that consists of both EJBs and WebApps (and PolicyIVP has both), the act of deploying the application will cause the EJBs to be installed in the EJB Container and the WebApps installed in the Web Container. Right now the WebApp you constructed by hand in the previous lab is installed in the Web Container and waiting activation.

Both containers are automatically created when you create the application server. The Web Container, however, requires that you provide a configuration file so the container can "come to life" and provide Web Services. That configuration file is known as the `webcontainer.conf` file.

The webcontainer.conf File



Contains the property settings for the web container

Lots of things in there, but two are of particular importance:

- **Virtual Host Definition**

  `host.default_host.alias=`
- **Context Root**

  `host.default_host.contextroots=`

Used to "bind" an application to a "virtual host" ... more on that in a bit

But first, some fundamentals

■

The `webcontainer.conf` file supplies the necessary information for the web container to operate. Each application server instance will have a pointer a `webcontainer.conf` file. That pointer is made out of the `jvm.properties` file, which resides in what is sometimes known as the "private directory" of the server instance. For instance, the "private directory" for your `APSRV3` server's `APSRV3S1` server instance is:

```
/WebSphere390/WAS401/controlinfo/envfile/WSLPLEX/APSRV3S1
```

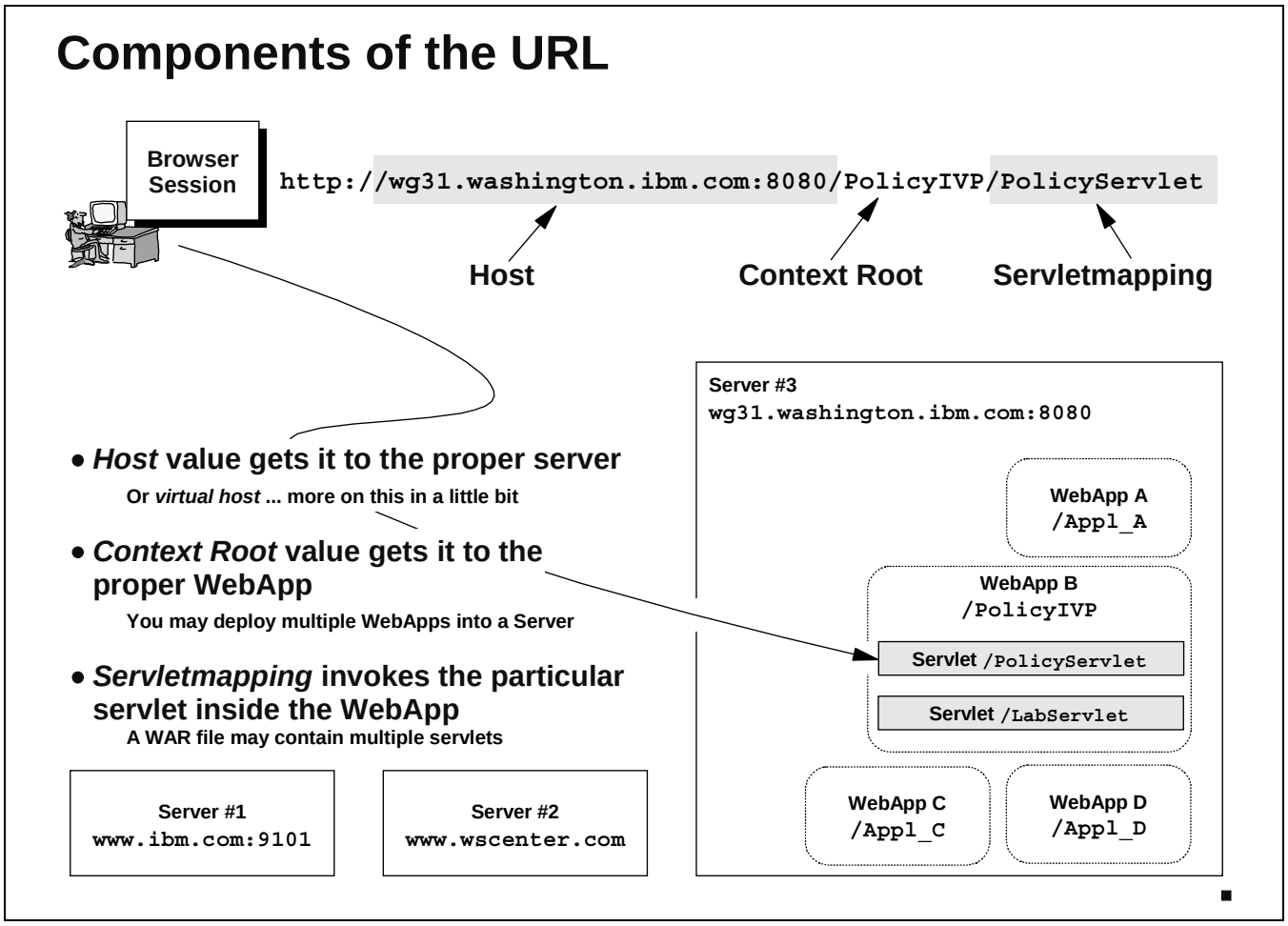
In that directory you'll find the `current.env` file for the instance, the `trace.settings` file and the `jvm.properties` file. The pointer to the `webcontainer.conf` file is a property in the `jvm.properties` file called `com.ibm.ws390.wc.config.filename=`. The directory *and filename* on the right side of the equal sign of that property is where the web container will go looking for the configuration file. The example shown above illustrates pointing back to the private directory. This makes some sense: keep all your configuration files in one place.

WebSphere V4.x supplies a sample copy of the webcontainer.conf file in:

```
/usr/lpp/WebSphere401/bin
```

Simply copy that to instance's private directory and make the pointer out of jvm.properties.

The `webcontainer.conf` file has quite a few properties in it, and in time you will come to understand what each does. For now, the two most important are the virtual host definitions and the context roots definitions, which we'll cover over the next few charts. Those are used to "bind applications to virtual hosts." More on that in a bit. First, it's important to cover some fundamental stuff.



To understand how all of this works, it's important to understand the components of the URL that's used to invoke a WebApp deployed on WebSphere V4.01. The URL has three basic pieces:

A "host" component -- this is used to help the network and WAS get your request to the right server. You might very well have multiple servers in your company. This also relates to the virtual host, which we'll cover in a bit.

A "context root" component -- a WAS 4.x application server may have any number of web applications deployed into it. The "context root" portion of the URL is used by WAS to route the URL to the proper WebApp. All WebApps deployed into WAS have a "context root" value set in the deployment descriptor (XML file) for the application. If you recall, you set the Context Root value in the previous lab using the AAT tool.

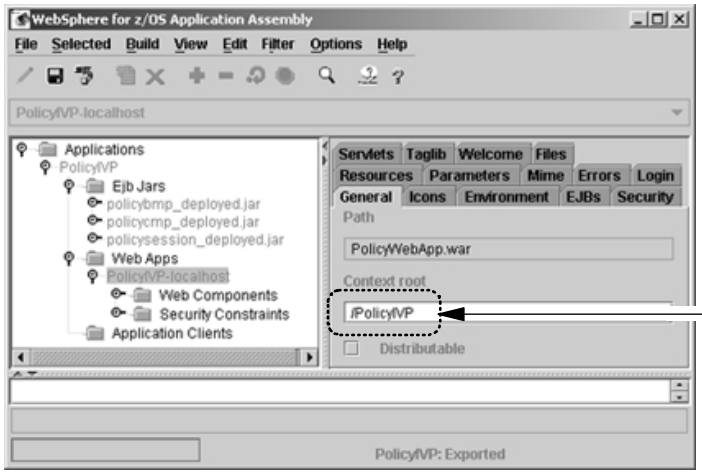
A "servletmapping" component -- a Web Application may have any number of servlets packaged inside of it. The "servletmapping" portion of the URL is used by WAS to know which of the potentially many servlets to invoke. The servletmapping value is set in another deployment descriptor, which we'll illustrate in a few charts.

Note: The URL may have quite a bit more that follows the "servletmapping" portion. These would be parameters passed into the application. You'll typically see these separated from the rest of the URL with a question mark (?), and each parameter separated with an ampersand (&).

The key message is this: the URL must have information in it to help the WAS system route the URL to the proper servlet for execution. Since you may have multiple servers, multiple webapps in a given server, and multiple servlets within a webapp, some way to getting the request to the correct servlet is required.

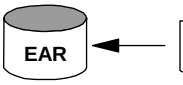
Context Root Setting for WebApp

http://wg31.washington.ibm.com:8080/PolicyIVP/PolicyServlet



- **WebApp's Context Root set in AAT**
- **Property set in `application.xml` deployment descriptor**
- **Connects URL to WebApp in which servlet is packaged**

Not the whole story ... "virtual hosts" come into play as well. See next chart.



`application.xml`

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

The "context root" for a WebApp is set in the AAT tool. It falls under the "General" tab of the web application object under the "Web Apps" folder. The AAT tool will take the value you supply and generate a portion of the `application.xml` file (the "deployment descriptor" for the application) which will assign the context root value to the WAR file in which the web application is packaged.

Take a look at the example shown above. That is an actual clipping from the `application.xml` file generated by AAT. The XML portion delimited by the `<web>` tag provides the WAR file name and the context root for that WAR file. In the case of the previous lab, the context root value was `/PolicyIVP`.

So that's how the context root is set. When a WAS application server starts up, it reads in and knows about all the web applications deployed into it, and the context roots for each. When a URL is passed to the application server, the server may then route the request to the proper web application based on the context root.

But that's not the whole story ... the other half of this equation is the "virtual host" value, which is explained next.

Concept of Virtual Hosts

http://police.state.md.us/crime/ReportTheft

http://fire.state.md.us/safety/seminars

DNS Entries

police.state.md.us	59.1.1.8
fire.state.md.us	59.1.1.8

Network

- **You may have multiple hosts pointing to one server, and wish to keep applications separated by host**
- ***Virtual hosts* allow WAS to isolate the requests based on the URL host**
- **This allows WAS to keep applications separated**
- **Key is "binding" application to virtual host**

URL for Fire Webapp

URL for Police Webapp

Web Container

Fire Dept. WebApps

Police Dept. WebApps

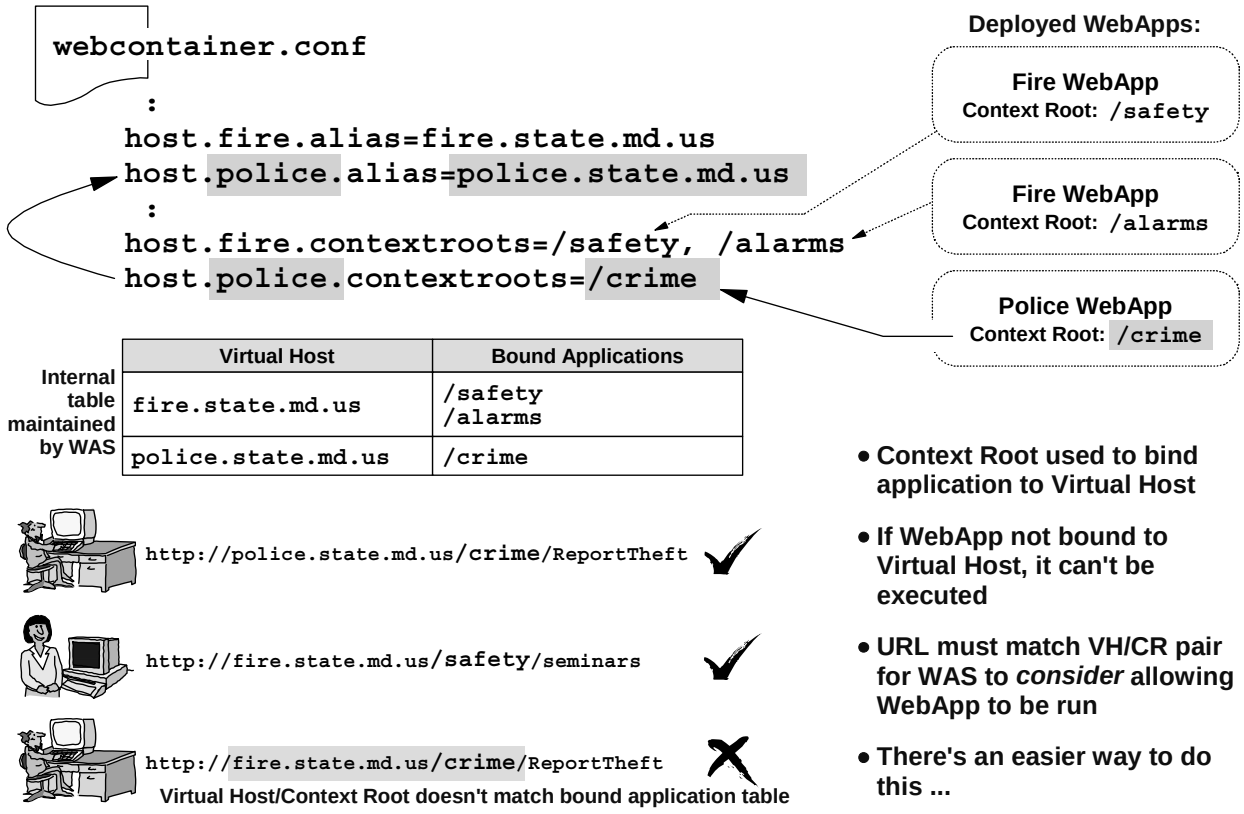
Suppose you have two different people on the web, one pointing his browser to the police department of the State of Maryland, and one pointing her browser to the fire department of the State of Maryland. Would those two URLs go to different web servers?

Not necessarily. The Domain Name Service (DNS) for the network might point both host names to the same IP address, which would mean both URLs would flow to the same adapter on the same machine. When those URLs flowed into the web container, you would want a way to make certain that those applications written for the police department are only run for URLs that came in with the police department host name, and those applications written for the fire department isolated to fire department URLs.

To do this, WAS implements something called a "virtual host." What the virtual host does is provide a mechanism by which WAS may compare the received URL against a setting in the `webcontainer.conf` file, and see whether the application being requested (based on the context root) is associated with that host. In other words, it provides a way of making sure that *police* department applications aren't run accidentally when a `fire.state.md.us` URL is received.

The key to this is the "binding" of applications to virtual hosts. *Binding is nothing more than associating a context root with a host name*. The next chart shows how this is done.

Binding Application to a Virtual Host



Here's where we introduce some of the definitions in the `webcontainer.conf` file:

`host.<name>.alias=<host name[:port]>` -- this statement is used to name a URL host value that you expect to be received by this application server, and to tag this host value with a "name" of your choosing. This "name" portion is used to connect this statement in the `webcontainer.conf` file with other statements in the file that have the same `<name>` value.

`host.<name>.contextroots=<context root>` -- this statement is used to name context roots that will be associated with the host value. What host value? The host value provided on the `alias=` statement that has the same `<name>` value as this statement.

This is best illustrated with an example. Imagine you have three web applications deployed into a server. The have context roots of `/safety`, `/alarms` and `/crime`. Further, imagine your server will service both the fire department and the police department. The first two webapps are the fire department's, and the last webapp is the police departments.

The `host.fire.alias=fire.state.md.us` statement defines the host value for the fire department, and ties that value to the string "fire," which is the `<name>` portion of the statement. The `host.police.alias=police.state.md.us` does the same thing, but with a string of "police."

Further down in the webcontainer.conf file you have two more statements. The `host.`**`fire`**`.contextroots=` statement defines the context roots of the applications that are associated with -- or *bound* -- to the host name specified on the `host.`**`fire`**`.alias=` statement. Notice how the <name> value "fire" is the same on both of those statements. That's what ties one statement to the other. When this application server is started up, WAS will read the context root values specified in the `application.xml` files of each deployed application, and match the context root definition found there with the values found on the `contextroots=` statements in the

`webcontainer.conf` file. When a match is found, it'll then bind the application to the host name specified on the `alias=` statement with the same `<name>` value. In this example, the police department's `contextroots=` statement has one definitions on it: `/crime`. Of the three applications deployed into the server, one has a context root definition of `/crime`. That application matches matches this statement. WAS then reads backwards on the statement, finds the `<name>` value of "police," then goes and looks for the `alias=` statement with the same `<name>` value of "police." In this example, that statement names a host value of `police.state.md.us`. WAS then *binds* the application -- `/crime` -- to the host value of `police.state.md.us`. *This application has been bound to the virtual host `police.state.md.us`*.

The fire department's two applications are bound in the same manner. The difference there is the fire department's `contextroots=` definition names two values. The fire department has two applications, each matching with a string found on the `contextroots=` definition.

WAS builds an internal table that lists all the context root values associated with each virtual host defined.

Here's what happens next. A URL is received with the police department's host name on it and is requesting the `/crime` application. WAS checks it's internal table and finds that sure enough the virtual host of `police.state.md.us` is associated with a context root of `/crime`. That application is permitted in for execution.

The same would apply to second example URL, which is requesting the `/safety` application and has the `fire.state.md.us` host name on the URL. WAS checks its table and finds that this is permitted.

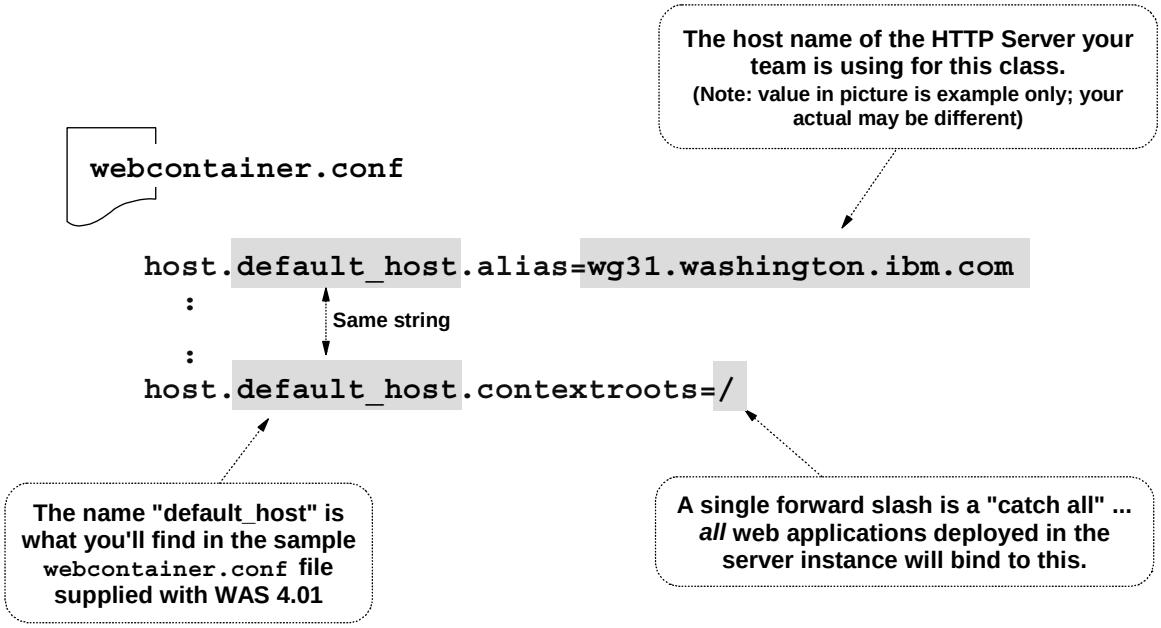
The third example illustrates a case that won't work. A URL is received with `fire.state.md.us` as the host and an application context root value of `/crime`. WAS checks its table and finds that this is an *invalid combination*. It rejects the request.

So the match is really on the "Virtual Host / Context Root" pair, not just the context root value.

What we've shown you here is how the mechanics of this thing works. Take a deep breath ... there's an easier way to do this. That's illustrated next.

The Context Root "Catch All"

For this class, we can simplify this process quite a bit ...



Let's see some variations on this "Virtual Host / Context Root" theme ...

■

It turns out there's a way to make this whole thing a lot simpler. The contextroots= definition allows the use of a single slash, which acts as a universal "catch all." All context roots will match with a single slash. If you read backwards on the `contextroots=` definition, you find the `<name>` value of `default_host`. This is what you'll find in the supplied sample `webcontainer.conf` file you copy over from the `/usr/lpp/WebSphere401/bin` directory.

Going up in the `webcontainer.conf` file you'll find the `host.default_host.alias=` definition, and this will tie to the `contextroots=` definition by virtue of the `default_host` value for `<name>`. It is on this definition that you code the IP host name of your team's system. For this class, that's `wg31.washington.ibm.com`.

That's it. All applications you now deploy, regardless of context root value, will bind to your `wg31.washington.ibm.com` virtual host. This works perfectly well when you have only one host name. It's when you start thinking of supporting multiple hosts that you need to worry about which application binds to which host.

Note: If you only have one host supported by your server, you still need to define a virtual host. *You can't get away with leaving the `alias=` definition blank*.

Now let's look at a few variations on how this stuff works.

Virtual Host/Context Root Variations

Binding multiple Context Roots to a Virtual Host:

```
host.default_host.alias=wg31.washington.ibm.com
  :
host.default_host.contextroots=/PolicyIVP,/Sample1,/TestABC
```

Virtual Host	Bound Applications
wg31.washington.ibm.com	/PolicyIVP /Sample1 /TestABC

Binding Context Root to multiple Virtual Hosts:

```
host.default_host.alias=wg31.washington.ibm.com,www.wscenter.com
  :
host.default_host.contextroots=/PolicyIVP
```

Comma!

Virtual Host	Bound Applications
wg31.washington.ibm.com	/PolicyIVP
www.wscenter.com	/PolicyIVP

Here we illustrate two common variations on this:

Multiple Context Roots to a Single Virtual Host -- this is the same things as was illustrated earlier with the fire department's two applications of `/safety` and `/alarms`. The `contextroots=` definition permits the coding of multiple strings. You separate them with a comma as shown. Deployed applications that match those values will then be bound to the virtual host of, in this example, `wg31.washington.ibm.com`. When the server is started, the "internal table" WAS will build showing context root to virtual host binding will be as shown.

Context Root to Multiple Virtual Hosts -- this is used when you want to be able to access the *same application, but use different host values*. In this example the two virtual host values are `wg31.washington.ibm.com` ***and*** `www.wscenter.com`. The internal table WAS builds will show the application `/PolicyIVP` bound to both virtual hosts. That means you could come into WAS with either host name on your URL and be permitted access to the `/PolicyIVP` application.

You've probably guessed that there's an opportunity to introduce ambiguity into the picture with all these `alias=` and `contextroots=` combinations. You're right. The next chart illustrates how WAS will sort it all out.

How WAS Resolves Ambiguity

```
host.aaa.alias=www.aaa.com
host.bbb.alias=www.bbb.com
host.ccc.alias=www.ccc.com
   :
host.aaa.contextroots=/
host.bbb.contextroots=/Pol/*
host.ccc.contextroots=/PolicyIVP
```

This is a wildcard. You must precede all uses of a wildcard asterisk with a slash, as shown.

`/Pol/*` will allow `/PolicyIVP` to bind, as well as `/Police` as well as `/Polly_want_a_cracker`

Class WebApp
Context Root: `/PolicyIVP`

???

All three "contextroots=" definitions will allow WebApp with `/PolicyIVP` to bind.

Which will WAS choose?

Order of binding:

1. Exact match
2. Wildcard match
3. Single slash match

In this example, WAS would bind WebApp to host `www.ccc.com`

Don't Do This!

```
host.aaa.alias=www.aaa.com
host.bbb.alias=www.bbb.com
   :
host.aaa.contextroots=/
host.bbb.contextroots=/
```

Ambiguity that can't be resolved results in unpredictable bindings. Don't introduce ambiguity with single-slashes across different hosts, or with the same contextroots= across different hosts

We've seen how a `contextroots=` definition can be coded explicitely, and how a single slash can be a universal catch-all. It turns out that WAS also permits the use of wildcards in the `contextroots=` string. The wildcard is the asterisk (*), but in order to use it you must precede the asterisk with a slash. So the string `/Pol/*` provides the matching for any context root that starts with `/Pol`.

In the example above we have three different `contextroots=` statements, each tying back to a different virtual host value. The strings on each would permit the context root of `/PolicyIVP` to match. To which virtual host will WAS bind the application? The answer is *not* "all of them." (We saw earlier how to bind an application to multiple virtual hosts; that was done with a single `contextroots=` definition.) WAS doesn't like to match on multiple contextroots= definitions, so it'll choose one. Which will WAS choose?

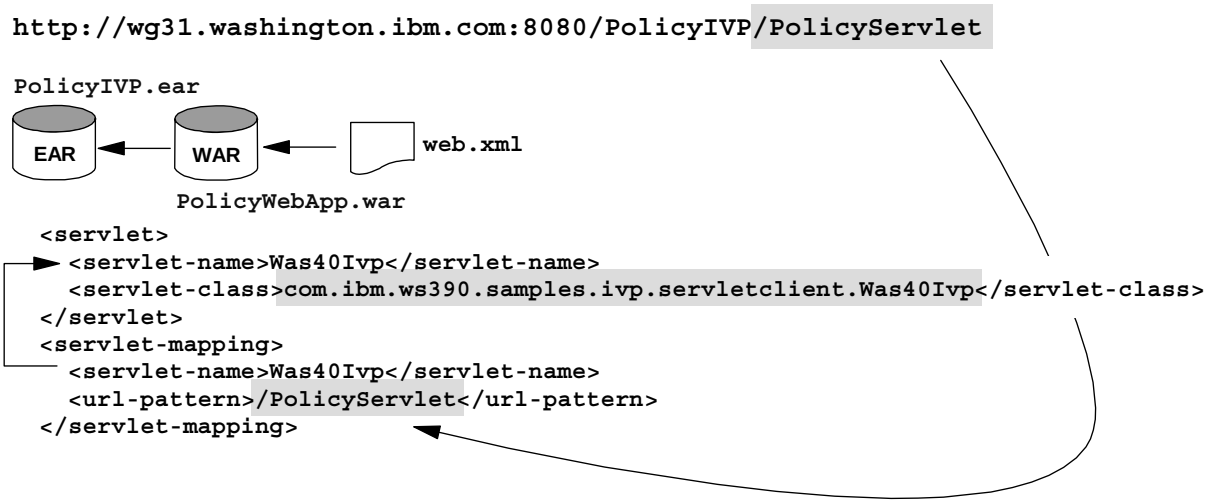
It goes in the order shown in the chart above: an exact match first, then any wildcard matching (with a more explicit wildcard match taking precedence over a less explicit wildcard match), and then finally the single slash. In this case, WAS would match the application to the explicit coding of `/PolicyIVP`, which would bind the application to the `www.ccc.com` virtual host.

What you want to avoid is what's illustrated in the box in the lower right of the chart. That's where you have the single slash on multiple contextroots= definitions, with the hope of binding all applications to the two virtual hosts. This results in unpredictable binding by WAS (if there's a pattern to it, we've not been able to find it ... it appears random). The message here is this: you have some responsibility to avoid ambiguity in your webcontainer.conf file.

Let's say you have a match on the virtual host and context root. How does WAS isolate down to the servlet? With the servletmapping component of the URL, which is discussed next.

Servletmapping Setting for Servlet

Final piece of the puzzle is the Servletmapping, which allows WAS to resolve request to a particular servlet class file and execute that servlet:



- **Servlet's Servletmapping set in tool used to create webapp (Studio, WSAD, by hand like we did in lab)**
- **Property set in `web.xml` deployment descriptor in WAR file**
- **Connects URL to servlet that is to be executed**

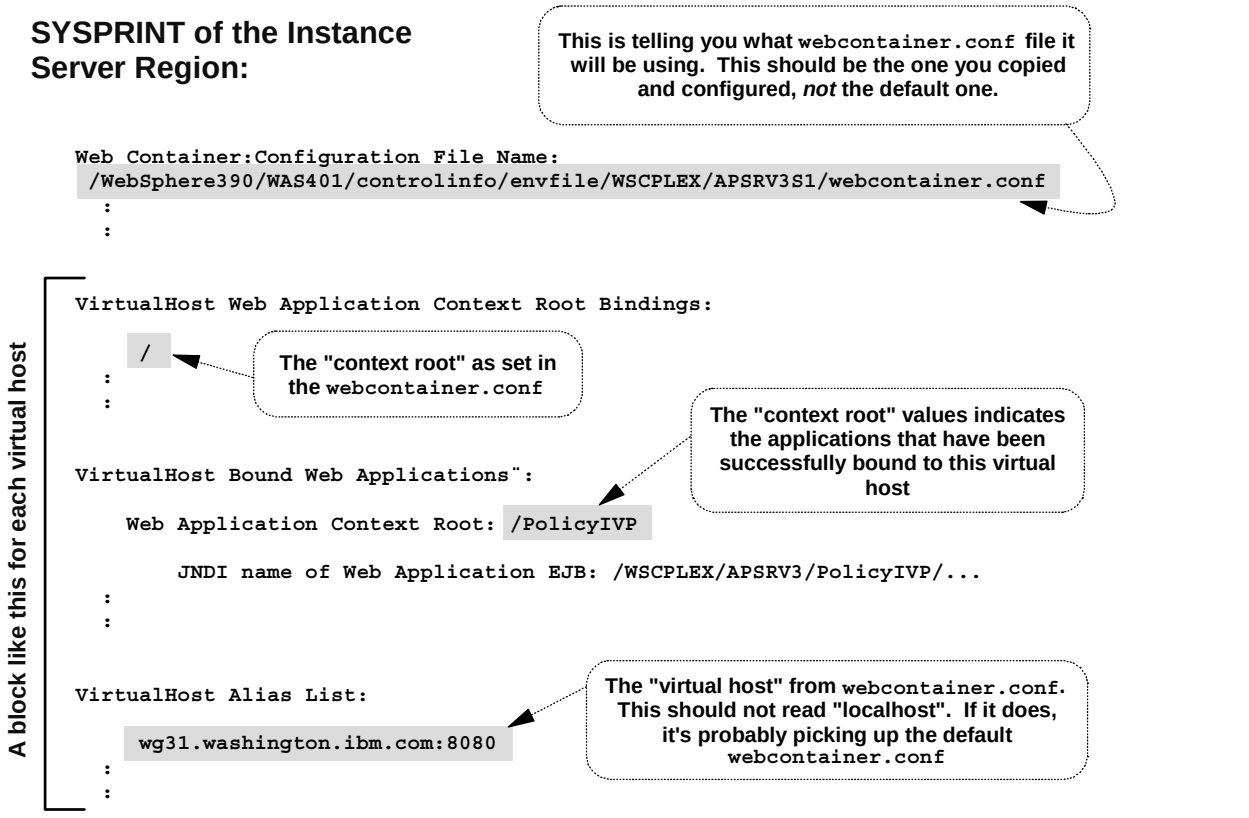
Recall that the third and last component of the URL is the "servletmapping" string. This is used by WAS to resolve the request down to one of what may be many servlets packaged in the web application.

The servletmapping value for a servlet is defined in the `web.xml` file, which is a deployment descriptor found in the WAR file. This value is coded into the `web.xml` file by whatever tool you might use to create the web application, such as WebSphere Studio, or the new WebSphere Application Developer. (For this class, we simply gave you a completed `web.xml` file, which you downloaded from the host and packaged into your hand-constructed WAR file.) Take a look at the chart above, and note where the XML `<url-pattern>` tag defines the string that needs to match what's coming in on the URL. What we need to do is tie that string to an actual Java class file that represents the servlet. That's done with the `<servlet-name>` definition. In this example, that value is `Was40Ivp`. WAS then goes looking for a `<servlet>` stanza in the `web.xml` file with the same `<servlet-name>` value ("Was40Ivp" in this example), then reads the `<servlet-class>` value. That specifies the actual Java class file that is to be executed. In this example, that's a long Java package name.

You've completed the puzzle: host component gets the URL to the proper server; context root (in combination with the virtual host) resolves the request down to a given web application; servletmapping resolves the request down to a given servlet class file.

Initial Verification: Server SYSPRINT

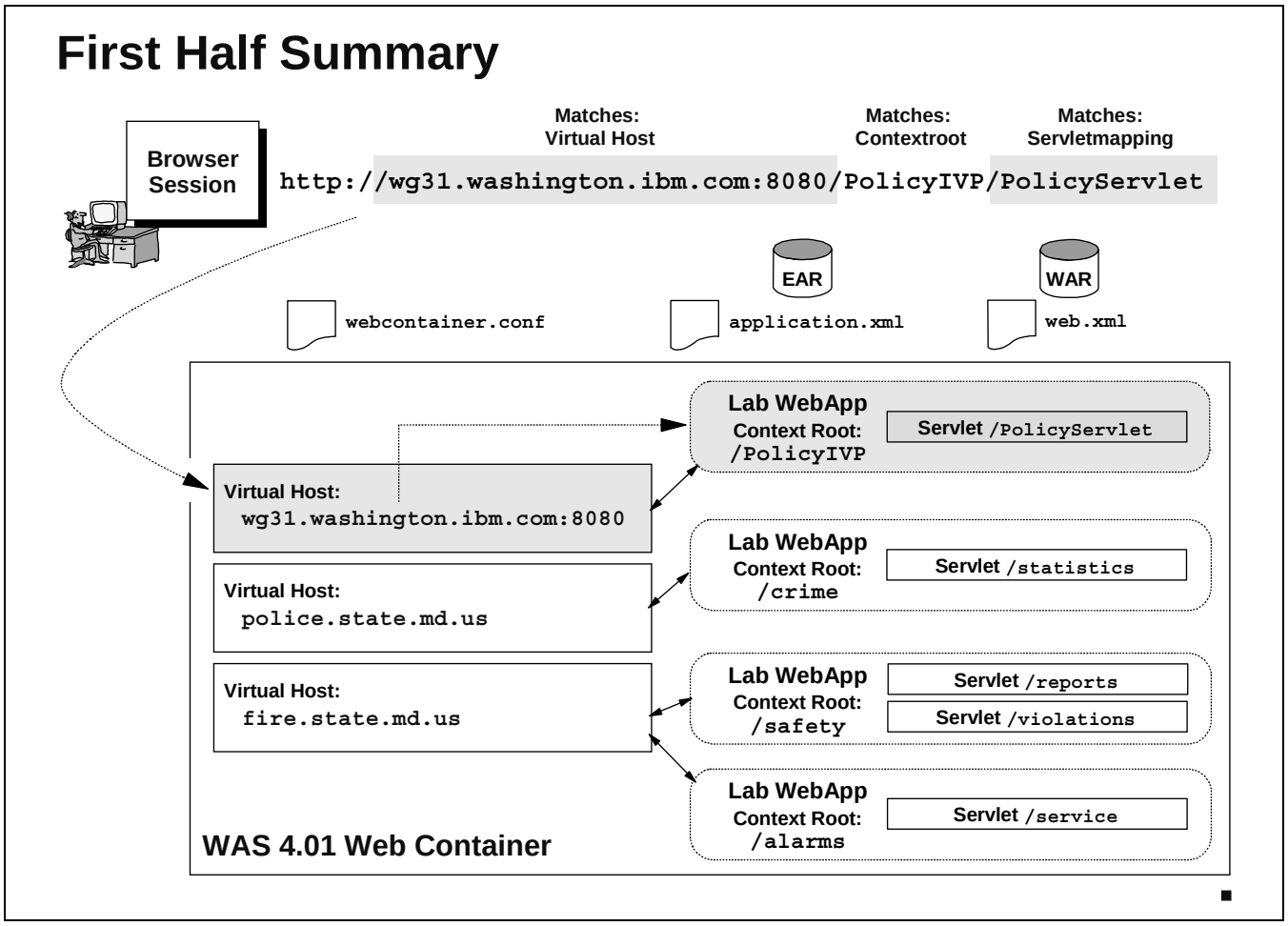
SYSPRINT of the Instance Server Region:



How can you tell if your applications bound to the virtual hosts the way you wished them to? By inspecting the SYSPRINT of the *server region* (not control region) of your application server. This will tell you what's happening when the application server is coming up:

- The first thing that's important is the `webcontainer.conf` file that's being used by the server. Look at this very carefully. If you mis-typed anything in the pointer to the file in the `jvm.properties` file, WAS will look for but probably to fail to find the (erroneous) reference to the `webcontainer.conf` file. It will then fall back and take the default `webcontainer.conf` file, which is located in the `/usr/lpp/WebSphere401/bin` directory. If you see that default copy in use, it means you've made a mistake in your `jvm.properties` update. The default `webcontainer.conf` won't have your `contextroots=` and `alias=` updates, so your applications won't be bound properly.
- WAS will then report on the applications that are bound to each virtual host specified in the `webcontainer.conf` file. You'll see a block of lines similar to what's shown above for each virtual host you define. In this example, WAS is telling you that `/PolicyIVP` matched with the `contextroots=` value of `/` (single slash), and that resulted in it being bound to `wg31.washington.ibm.com:8080`.

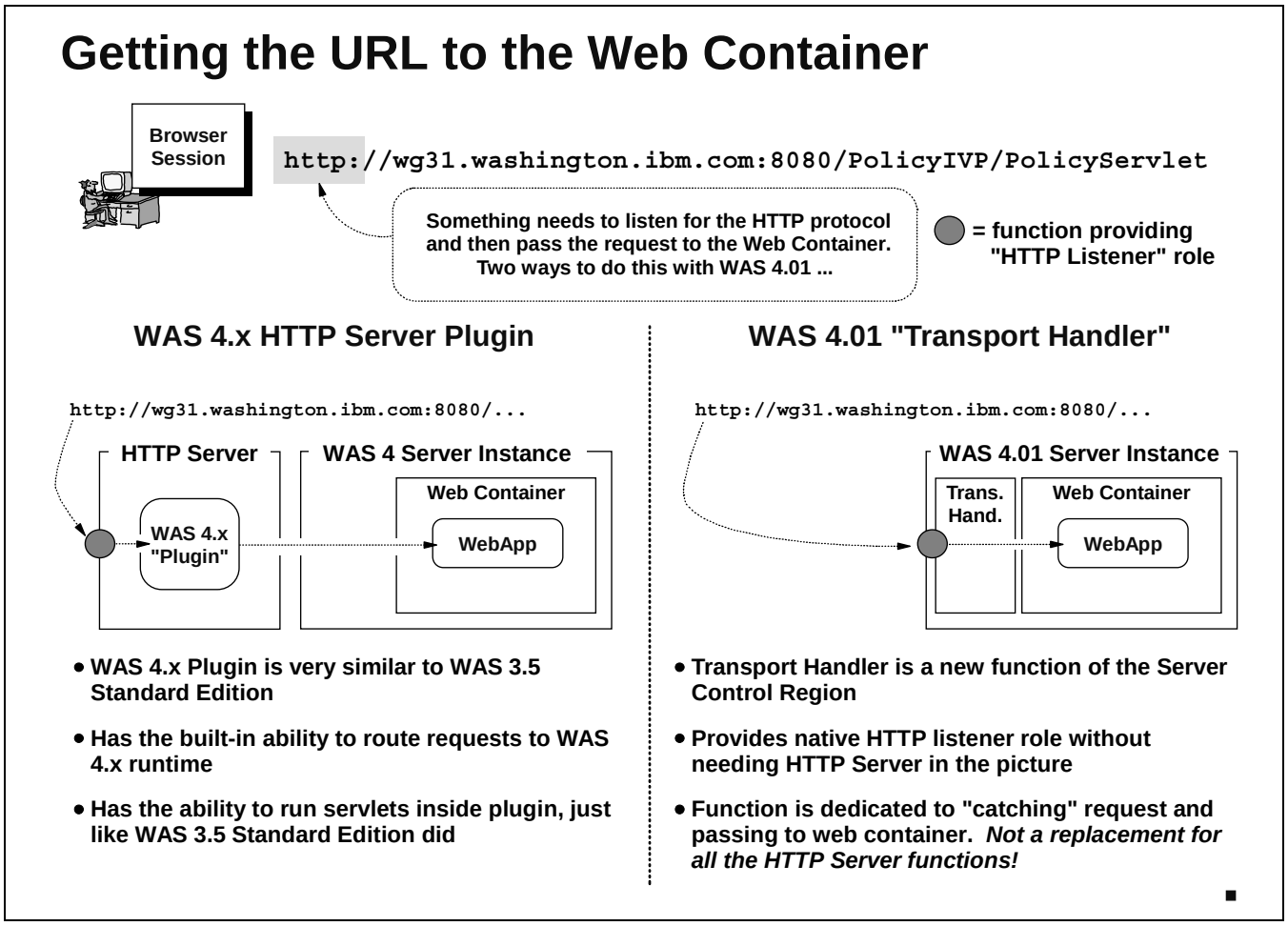
Make it a habit of looking at the SYSPRINT each time you make an update to `jvm.properties` or `webcontainer.conf`. You'll avoid spending hours debugging problems that are really nothing more than typographic errors.



And here's the summary of the past dozen or so charts:

- A URL has three major components to it: the host portion, the context root portion and the servletmapping portion.
- The host portion is used to get the URL to the right server, and is what matches up against the virtual host definition you provide on the `host.<name>.alias=` statement in `webcontainer.conf`.
- The context root portion is used to resolve the request down to one of what may be many web applications deployed in the server. The context root value for an application is specified in the AAT, and ends up as a definition in the `application.xml` deployment descriptor. WAS uses this context root value to "bind" the application to a virtual host, based on your definitions in the `webcontainer.conf` file. If the host portion and context root portion of the URL match the "bound application" table of WAS, WAS will accept the URL for potential execution.
- The servletmapping portion is used to resolve the request down to one of what may be many servlets packaged in the web application. The servletmapping value for any given servlet is specified in the `web.xml` deployment descriptor for the webapp. The `web.xml` file is found inside the WAR file that represents the web application. If the servletmapping portion of the URL matches a servletmapping string WAS knows about, it'll go in search of the Java class file and load it for execution.

You have the issue of binding applications to virtual hosts mastered. Now comes the issue of how to get the URL to the WAS application server in the first place. To do that, you need *something* in the picture that will listen for HTTP requests and pull them off the wire. There are two ways to do that, and each is discussed next.



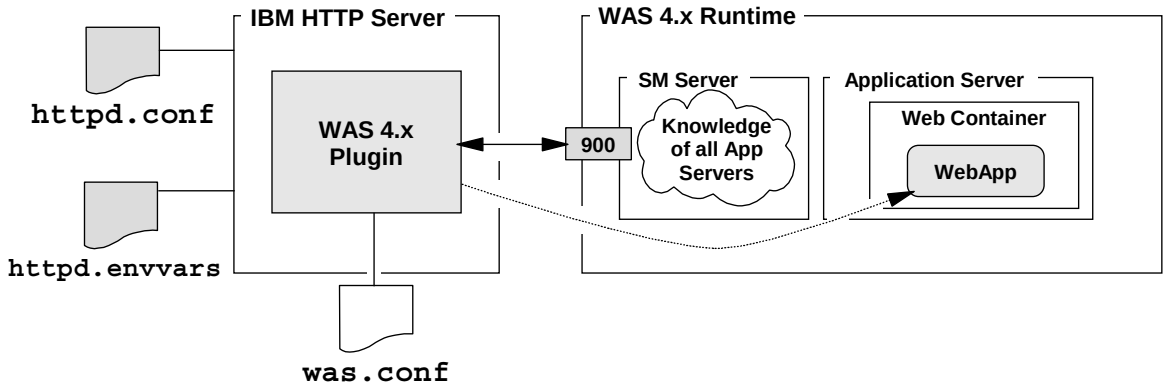
For a URL coming out of a browser to get to the Web Container, something needs to listen for the HTTP protocol and pull the request off the network. The web container itself can't do that.

There are two ways to do this with WAS 4.01:

- With WAS 4.00 (and continued with WAS 4.01) there is an HTTP Server "plugin" that provides the ability to catch requests and route them to the web container. This is called a "plugin" because the code makes use of the HTTP Server's API, and the code is said to "plug in" to the API. The WAS 4.x Plugin is in reality the WAS 3.5 Standard Edition plugin with some added stuff to allow the plugin to know what servers and applications exist over in the WAS 4.x runtime environment. Because it still has the WAS 3.5 SE product in it, servlets can still run in this WAS 4.x plugin. That provides a nice migration path, which will be discussed later.
- With WAS 4.01 there is a new integrated HTTP listening agent known as the "Transport Handler." It is a function of the WAS 4.x application server itself, and because of that the HTTP Server is no longer needed. This new function is dedicated to the rather narrow task of catching the HTTP request and routing it to the web container. As such, it is not a replacement for the IBM HTTP Server and does not have all the bells and whistles of the HTTP Server.

Let's explore each in turn.

Configuring the WAS 4.x Plugin



To configure Plugin's initialization and routing of requests to Plugin:

```
  :
ServerInit /<install path>/was400plugin.so:init_exit /usr/lpp/WebSphere401,/etc/wasweb/was.conf
Service    /PolicyIVP/*   /<install path>/was400plugin.so:service_exit
ServerTerm /<install path>/was400plugin.so:term_exit
  :
```

<install path> = `/usr/lpp/WebSphere401/WebServerPlugIn/bin`

To configure connection to SMS:

```
RESOLVE_IPNAME=wg31.washington.ibm.com
RESOLVE_PORT=900
```

If you leave this out, it defaults to "localhost" and port 900. Probably will work, but coding it lessens ambiguity

■

The configuration of the Plugin involves updating the HTTP Server's `httpd.conf` and `httpd.envvars` files. The Plugin is a piece of code that runs inside the Webserver's address space, and *the configuration is very, very similar to that for the WAS 3.5 Standard Edition code*. Three updates to the `httpd.conf` file are required:

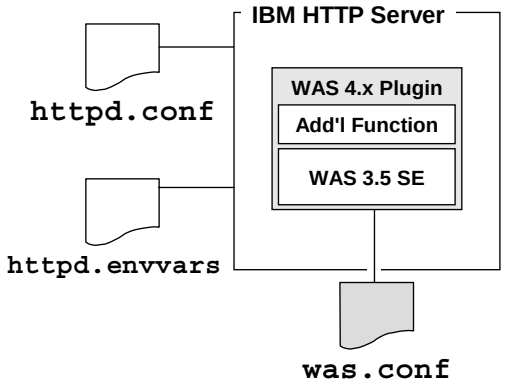
`ServerInit` -- this statement is used to instruct the Webserver to initialize the Plugin when the Webserver itself comes up. You will have only one `ServerInit` statement for the plugin code. It has the format as shown in the chart above. Two parameters are coded on the statement: the WebSphere install root, and the directory and file name for the `was.conf` configuration file for the Plugin.

`Service` -- this statement is used to catch URL requests and pass them from the Webserver realm into the Plugin. It has a mask, or template, that if matched to a received URL, instructs the Webserver to pass the entire URL over to the Plugin . You will have been 1 and "n" number of `Service` statements; the exact number depends on the number of different URLs you want to throw over to the Plugin.

`ServerTerm` -- this statement is used to gracefully shut down the Plugin when the Webserver is coming down. You will have only one of these statements.

Two updates are required to the `httpd.envvars` file, and they define the SMS to which the Plugin will connect and communicate. This connection is necessary because the Plugin needs to understand what applications are deployed where. The two updates to the `httpd.envvars` are shown above, and they simply name the IP host and port on which the SMS is listening. You could leave these values uncoded and the Plugin would try the default port 900 on "localhost." That would work in this class, and might work back home. But it's better to simply code them and avoid any ambiguity.

Will Servlets Still Run in 4.x Plugin?



Yes! They WAS 4.x Plugin is the WAS 3.5 SE code with some additional function added

(to provide connectivity to WAS 4.x runtime)

If you want to run servlets in the Plugin, code definitions in `was.conf` just as you did in WAS 3.5 SE.

Can you use your WAS 3.5 SE `was.conf` with this new Plugin? Yes, with only a minor change.

`http://wg31.washington.ibm.com/PolicyIVP/PolicyServlet`

`was.conf`

```
host.default_host.alias=wg31.washington.ibm.com
 :
deployedwebapp.hello.host=default_host
deployedwebapp.hello.rooturi=/PolicyIVP
deployedwebapp.hello.classpath=/u/team##/servlets
deployedwebapp.hello.documentroot=/u/team##
webapp.hello.servletmapping=/PolicyServlet
```

In this example, Plugin will try to run the request locally. If you don't want it to run locally, remove these definitions from `was.conf`

If Plugin see's "hit' on virtual host and rooturi in `was.conf`, it'll try to run request *locally*

Otherwise, it'll check with SMS and route the request to the WAS 4.x runtime

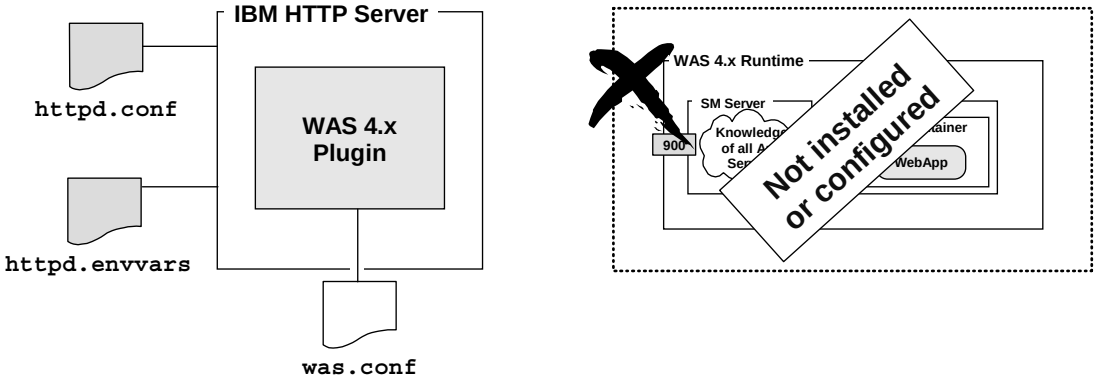
■

We have stated that the WAS 4.x Plugin is a superset of the WAS 3.5 Standard Edition code. The SE code had the capability to run servlets in the Plugin itself. Can the new WAS 4.x Plugin do the same? Yes, it can. The WAS 3.5 Standard Edition servlet execution runtime is still part of the Plugin, so servlet can run inside the plugin just as they did in WAS 3.5. The function is so similar to WAS 3.5's that the 3.5 `was.conf` configuration file can be used with the WAS 4.x Plugin with only a minor one-line change.

The key to whether the Plugin will run a request locally or route it to the WAS 4.x web container is whether the Plugin see's a "hit" on a virtual host and rooturi definition in the `was.conf`. (The notion of virtual hosts in the Plugin is nearly identical to that found in the web container, and "rooturi" is analogous to the "context root" in the web container.) If the Plugin gets a hit, it'll try to run the request locally as if the application was deployed in the Plugin itself. The example above shows just such a case. If no match is found in the `was.conf`, the Plugin then checks to see if the host and context root on the URL matches any it knows about in the web container. If so, it'll pass the request over.

If you want to make sure no requests are run locally in the Plugin, remove all your custom definitions from the `was.conf` so that all requests are routed to the web container.

Can The Plugin Be Run All By Itself?



Yes. This is known as the "simple configuration."

It is essentially the WebSphere 3.5 Standard Edition (SE) environment.

Just servlets and JSPs right now? Use this configuration until you're ready to use EJBs.

A logical next question is whether you can use the Plugin all by itself, without installing and configuring the WebSphere 4 runtime environment. The answer is yes ... and the configuration is known as the "simple" configuration.

If all you have right now is servlets and JSPs, but EJBs are in your project plans, this might be just the ticket for you. It gives you an opportunity to establish the plugin environment now and run your servlets. When you want to start running EJBs, you can configure up the WebSphere 4 runtime and deploy EJBs just like you did throughout this class.

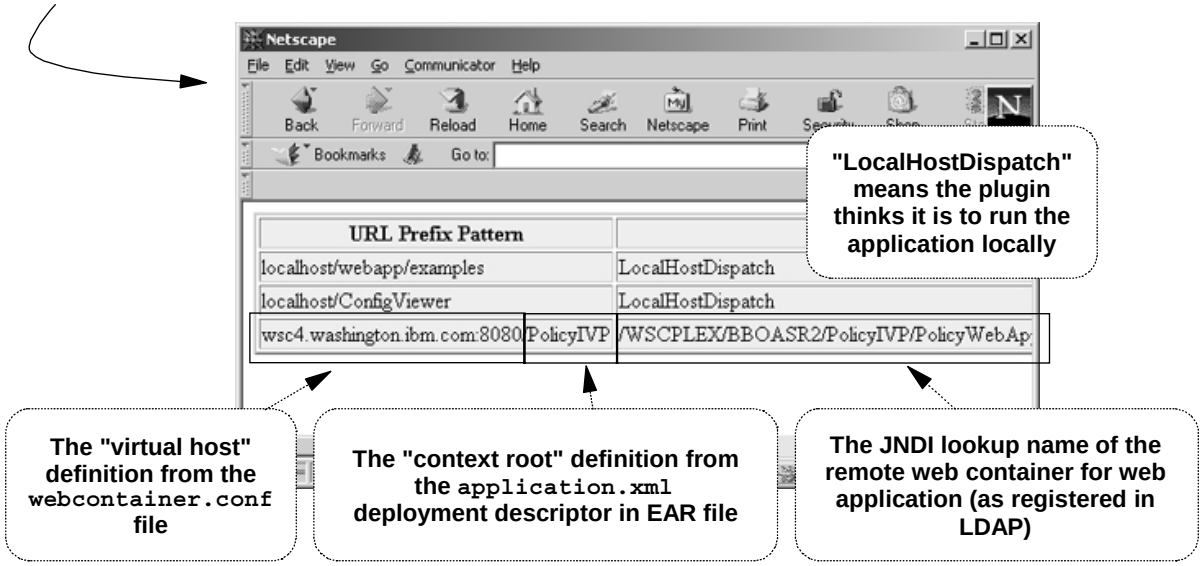
The only real difference between this and the WebSphere Application Server V3.5 product is the directories in which the code is installed. Beyond that, it's essentially the same thing as WebSphere V3.5.

Verifying Plugin-SMS Handshake

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

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

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



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

■

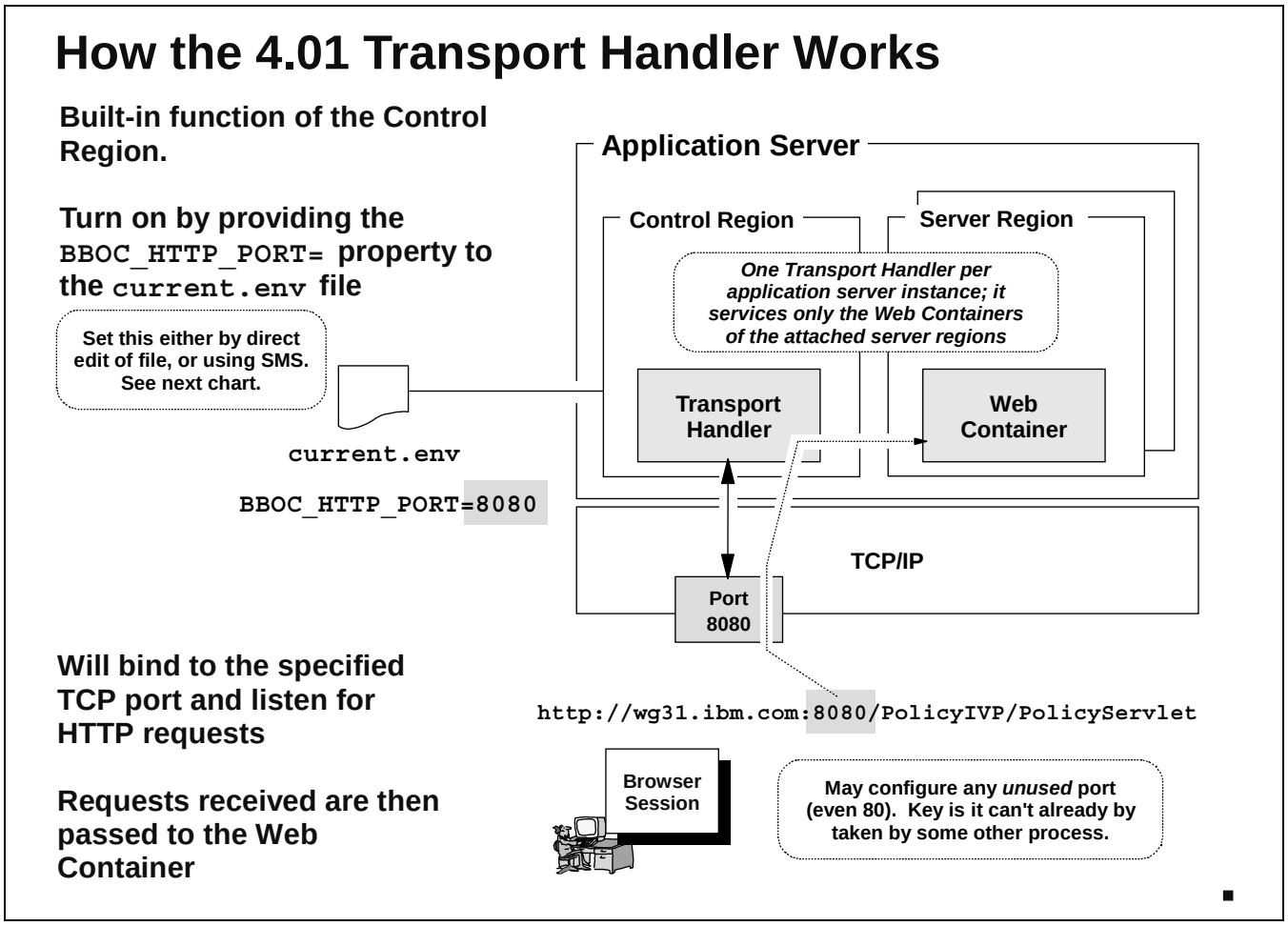
How can you easily tell if your Plugin is communicating with the SMS and knows about deployed applications? It turns out the Plugin comes with a servlet that will report what the SMS has told it. The servlet is invoked with the URL you see on the chart above, and if you follow the two instructions also provided above, you'll get a screen that is known as the "Application Dispatching" table.

Note: On the previous page we mentioned removing definitions from the `was.conf` file to allow all requests to flow over to the WAS 4.x runtime. The definition for this configuration reporting servlet is an exception. Leave in place the block of definitions that has a `rooturi=` value of `/webapp/examples`.

What this Application Dispatching table tells you is what applications are available to be executed -- or "dispatched" -- and where the application is deployed. The column "URL Prefix Pattern" tells you the "Virtual Host / Context Root" pair and the next column tells you where it will be dispatched. The first two rows in the table are for the servlet that provides this Application Dispatching report. It's virtual host is "localhost," which is the default virtual host setting in the default was.conf file. The value of "LocalHostDispatch" means the Plugin will execute this application *in the Plugin itself*.

The last line shows an application that's deployed over in the WAS 4.x runtime. The virtual host is `wsc4.washington.ibm.com:8080`, and the context root is `/PolicyIVP`. Where the application will be dispatched is the JNDI name of the remote web container, which is the web container in your application server. This is telling you that if a URL comes in with virtual host and context root of `wsc4.washington.ibm.com:8080/PolicyIVP`, it'll be passed over to the webcontainer for execution.

If the Plugin receives a URL that doesn't match any virtual host and context root pairs, it'll simply reject it. This illustrates the point that requests won't flow over to the WAS 4 web container unless the SMS has told the plugin that the application is deployed.



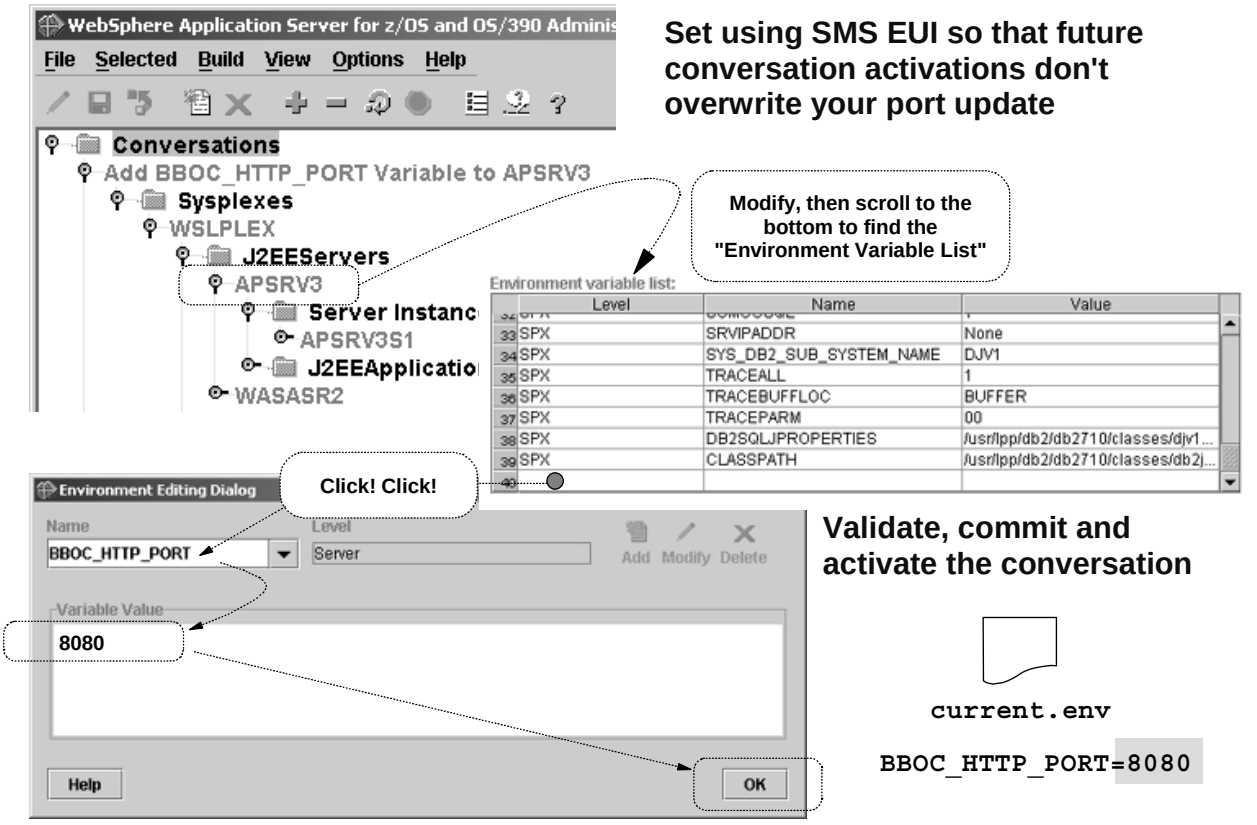
Now we'll discuss the other way in which you can get HTTP requests to the web container. It involves using what's known as the "Transport Handler." The Transport Handler is a built-in function of the application server control region of WAS 4.01 and above. *This won't work if your level of WAS is 4.00*. With the Transport Handler you don't need the IBM HTTP Server at all (though you may still have it in the configuration if you wish, and we'll talk about that in a bit).

You turn on the function of the Transport Handler by coding an additional environment variable in the control region's `current.env` file. That variable is `BBOC_HTTP_PORT=`, and it defines the TCP port on which the Transport Handler will listen for HTTP requests. If that port is available (in other words, it hasn't be grabbed by some other process), the Transport Handler will bind to the port. The port may be any unused port, including the default HTTP port of 80.

If a URL hits your TCP stack destined for the port on which the Transport Handler is listening, it'll grab the request, check the virtual host and context root values against what it sees deployed in the web container of the attached server region, and if it matches, it'll pass the request over to the web container.

Note: The Transport Handler is a function of the control region. Each application server you define -- and you may define any number of those -- has its own Transport Hander. The Transport Handler is designed to route HTTP requests to the Web Container of the server regions of *that application server, not the server regions of other application servers*. If you have multiple application servers defined, and each has deployed webapps, that would mean enabling the Transport Handler of each. With multiple Transport Handlers defined, you get into the question of whether to use different port numbers or use TCP port sharing. That topic is beyond the scope of this presentation.

Using SMS to Update Listening Port



There are two ways in which you may set the `BBOC_HTTP_PORT` environment variable in the `current.env` file: by hand-editing the `current.env` file and adding the property manually, or by using the SMS EUI and setting the property there. Hand-editing the file is quick and easy, but the changes you make will be lost the next time an SMS EUI conversation is activated. So the preferred method is to set it in the SMS EUI.

The SMS EUI allows environment variables to be set at different "levels" of the system heirarchy: at the SYSPLEX level (these values cascade down to all the lower-level components, such as servers and server instances); at the the Server level, or at the Server Instance level. The recommendation is to set this as the Server level. Here's why:

- If you set the property at the SYSPLEX level, then all application servers will try to bind to the same HTTP port. There is a way to accomplish port sharing in TCP, but the way TCP/IP routes a received request when port sharing is in effect is more or less a round-robin approach. If you have multiple application servers all listening on the same port, that implies all applications are deployed in all application servers. That's awkward.
- If you set the property at the Server *Instance* level, that gives you the opportunity to set different ports for the instances under a given server. The design of Server Instances is that one instance is indistinguishable from another from the client's perspective. Different port numbers violates that principle.
- Therefore, the recommendation is to set this at the Server Level. That means the value will cascade down to all the server instances you have configured under the server.

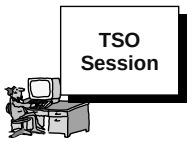
To set the value using the SMS EUI, you create a new conversation, modify the server itself, then scroll to the bottom of the settings for the server. There you'll find the environment variable list. If you double click on the open field after the last variable, it'll pop up a dialog box that'll allow you to set the property name, the level and the value. Then validate, commit and activate the conversation.

Verifying Tran. Handler is Listening

```
 BROWSE -- current.env
 Command ===>
******************************** Top ******
# ENVIRONMENT FILE FROM CONVERSATION ....
#-----------------------------------------
BBOC_HTTP_PORT=8080
```

The variable is now set in `current.env`

Issue the TSO NETSTAT command to see who's listening on what TCP ports:



`TSO NETSTAT`

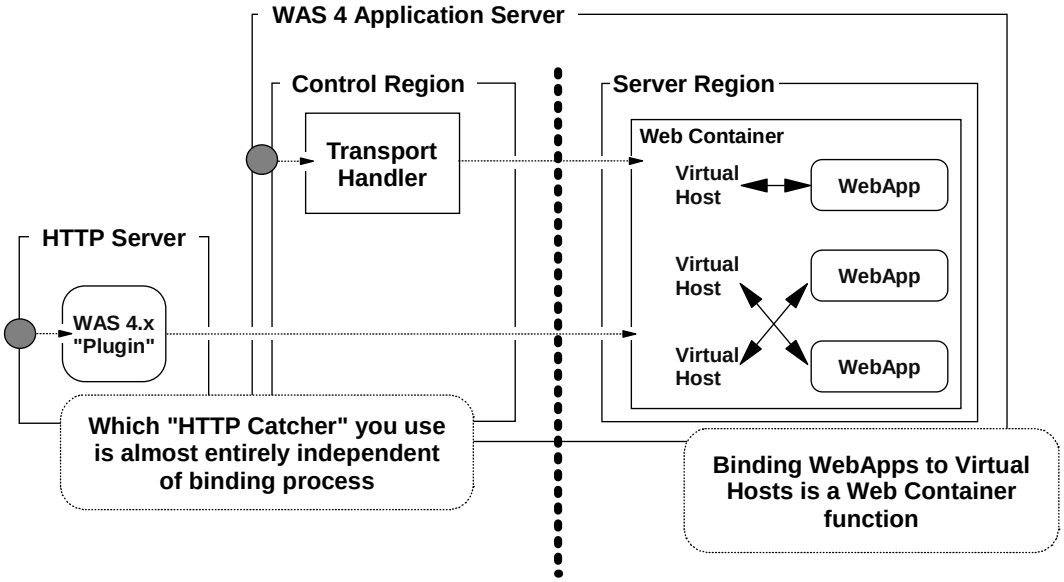
```
EZZ2350I MVS TCP/IP NETSTAT CS V2R10       TCPIP NAME: TCPIP1          02:28:47
EZZ2585I User Id  Conn     Local Socket           Foreign Socket         State
EZZ2586I -------  ----     ------------           --------------         -----
EZZ2587I APSRV3C  000017FB 0.0.0.0..1089          0.0.0.0..0             Listen

EZZ2587I APSRV3C  000017FC 0.0.0.0..8080          0.0.0.0..0             Listen

EZZ2587I APSRV3C  000017FA 0.0.0.0..1088          0.0.0.0..0             Listen
```

The Transport Handler is part of the Control Region ... look for the UserID for the Control region, and then check to see if the `BBOC_HTTP_PORT` port is in "Listen" state

■

Once you have the Transport Handler configure, you can check to see if has properly bound to the TCP port by issuing a `TSO NETSTAT` command. The results show all the processes bound to the various ports. Look for your ID assigned to your application server control region. This example is showing `APSRV3` bound to port `8080`, and it is in `Listen` mode. All that is a good sign: the control region is up and the Transport Handler is listening on port `8080`.

Still have to Bind Applications?



Yes. The process of binding WebApps to Virtual Hosts is a function and requirement of the Web Container. The Web Container doesn't really care which "HTTP Catcher" you use out front.

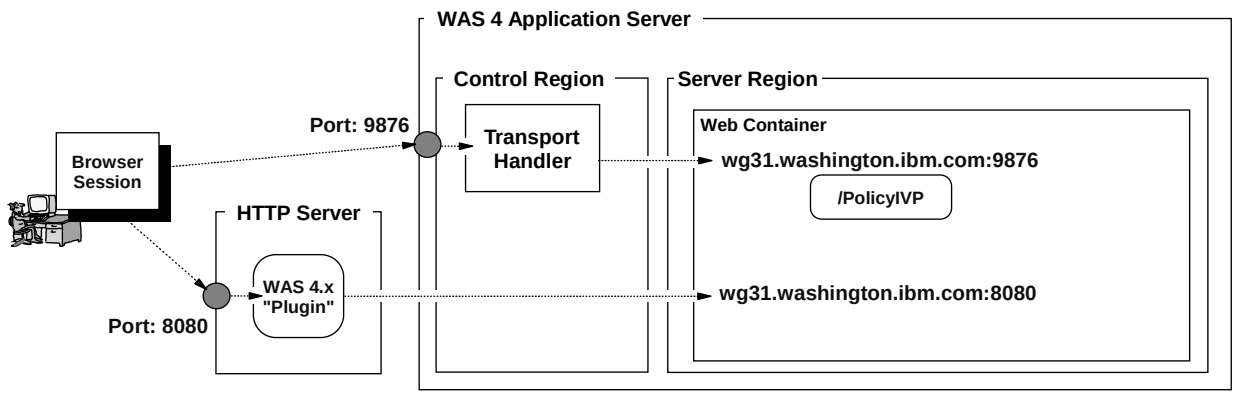
You do need to make sure the host (*and port*) used by whatever "HTTP listener" you choose to use will match your defined virtual host. See next chart ...

■

Some may wonder whether or not it is still necessary to bind applications to virtual hosts when the Transport Handler is in use. The answer is yes. This chart is in the presentation to illustrate a point: the act of binding applications to virtual hosts is pretty much independent of which of the two HTTP listening devices you use. However, you do need to be careful in the coding of your virtual hosts when port numbers other than 80 are in play. The next chart illustrates this.

Plugin and Tran. Handler at Same Time?

Is this possible? Yes! But you have to be careful about virtual hosts ...



Both URLs will get the request to the Web Container

`http://wg31.washington.ibm.com:9876/PolicyIVP/...`

- **This will execute PolicyIVP**

`http://wg31.washington.ibm.com:8080/PolicyIVP/...`

- **This will not! Application isn't bound to this virtual host!**

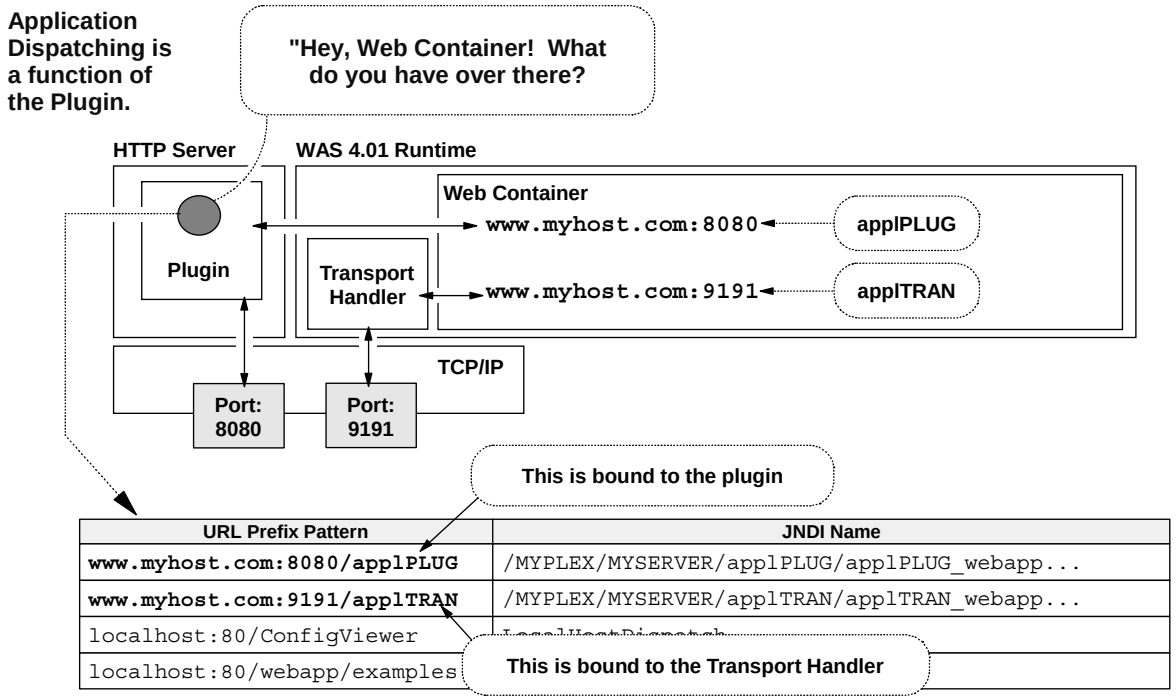
Can you bind application to different virtual hosts?	Can Plugin and Transport Handler share same port?	Can you use default port 80?
Yes!	**Yes!**	**Yes!**

A question that often comes up is whether the two HTTP listeners -- Plugin and Transport Handler -- can be used at the same time. The answer to that question is yes. However, if the two are listening on listening on different port numbers, then the virtual hosts you code over in the web container will be specific to the listener. That means you need to understand which virtual host your application is bound to. The picture above illustrates this:

- The Plugin is configured and bound to port 8080 and the Transport Handler is configured and bound to port 9876.
- Because the two are bound to different ports, their virtual hosts will be different, *even if they have the same host name*! `wg31.washington.ibm.com:9876` for the Transport Handler, `wg31.washington.ibm.com:8080` for the Plugin.
- In the example above, `/PolicyIVP` is bound to `wg31.washington.ibm.com:9876`, which is the virtual host related to the Transport Handler.
- A URL that hits the Transport Handler's port of 9876 will find its way to the Web Container and `/PolicyIVP` will be allowed to run: the virtual host and context root match.
- A URL that hits the Plugin's port of 8080 will be rejected: its virtual host / context root pair doesn't match what the web container is reporting.

There's nothing wrong with this ... it's working as designed. But it does mean you need to be careful, particularly when you first start out with this stuff, because the scenario pictured above can be confusing. Can this be made easier by binding your applications to both virtual hosts? Yes. Can the Transport Handler and the Plugin share the same port? Yes, if you make use of the TCP port sharing function. Can you use the default HTTP port of 80? Yes!

Application Dispatching, Part II



It'll report the bound applications as reported to it by the Web Container.

That holds for applications bound to Plugin virtual host or Transport Handler virtual host

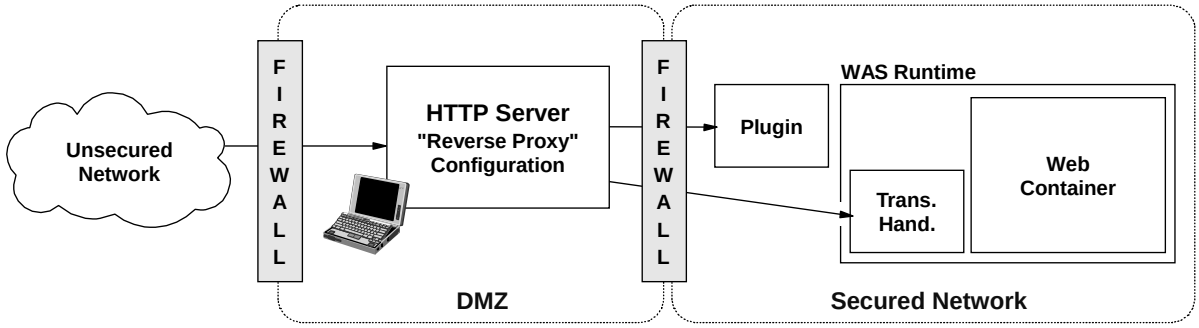
■

Let's revisit the issue of the Application Dispatching function of the Plugin. It turns out that function doesn't give a hoot whether an application over in the web container is bound to its virtual host or someone else's. The Application Dispatching function will simply ask the web container to report on *all* the bound applications. The Application Dispatching function will show them all, including those applications bound to the Transport Handler, if that's part of the configuration as well.

What this means is this: you can make use of the Plugin's Application Dispatching as a kind of monitor of the bound applications over in the web container. You don't have to have any application traffic go through the Plugin, but you can use the Plugin as a window into the web container.

Firewalls and DMZs

Recommended Method: "Reverse Proxy" in DMZ



***Not* Recommended:**

Plugin in DMZ

- **Flow from Plugin to WAS runtime is RMI/IIOP, and it's difficult to configure firewalls for IIOP**
- **Plugin must run in MVS image with running WAS 4.x daemon. That would imply stretching SYSPLEX into DMZ**

Transport Handler in DMZ

- **Transport Handler runs in Control Region and services Web Container in Server Region**
- **Both must run in the same MVS image. You can't separate the two**
- **Therefore, you can't get Transport Handler into DMZ without dragging Application Server region**

■

This is probably *the* question regarding the HTTP listening function of this WAS 4.x system. Many people are very interested in how to move the HTTP listener into the DMZ while maintaining the application server itself back in the secured network.

The recommendation is to configure a "reverse proxy" in the DMZ, probably on a distributed platform though it could be on a S/390 box, and use it to route requests to either the Plugin or Transport Handler behind the firewall.

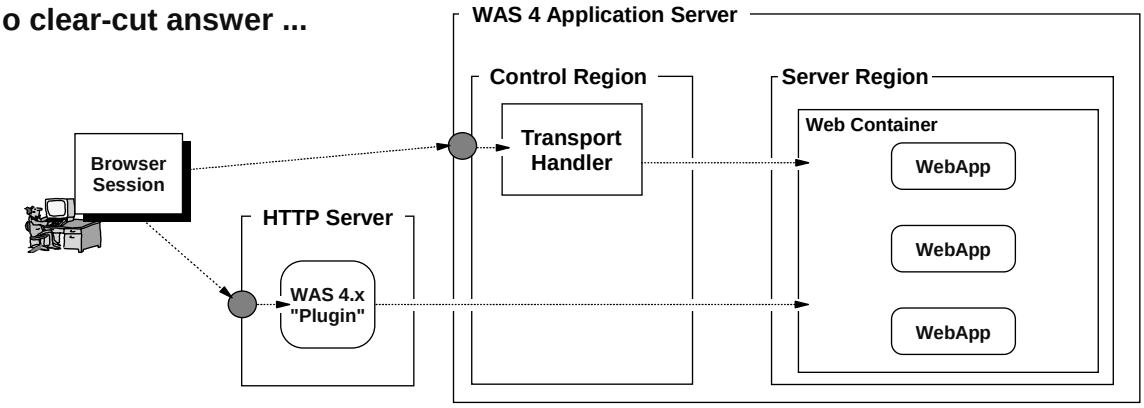
What's not recommend is placing the Plugin or the Transport Handler in the DMZ. Here's why:

- **Plugin** -- the flow out of the back of the Plugin to the WAS 4.x runtime is RMI/IIOP, and that's a very difficult protocol to configure a firewall around. Further, the Plugin must be running in an MVS image with a running copy of the WAS 4.x daemon. And the daemon will be part of your WAS 4.x SYSPLEX. The only way you could move the Plugin into the DMZ would be to stretch the SYSPLEX into the DMZ as well. That's generally not something you'd want to do.
- **Transport Handler** -- the Transport Handler is designed to provide HTTP listening services to the server regions connected to the control region of the application server. There's no way to separate the control region from the server regions: they must be in the same MVS image. Therefore, to drag the Transport Handler into the DMZ implies dragging the server region in as well.

Therefore, the best way to achieve a DMZ configuration is to use a reverse proxy. The firewall between the internet and the DMZ could be configured to allow through only HTTP on a given port, and the firewall between the DMZ and the secured network could be configured to allow through only HTTP from the reverse proxy's IP address.

Which to Use and Why?

No clear-cut answer ...



Plugin

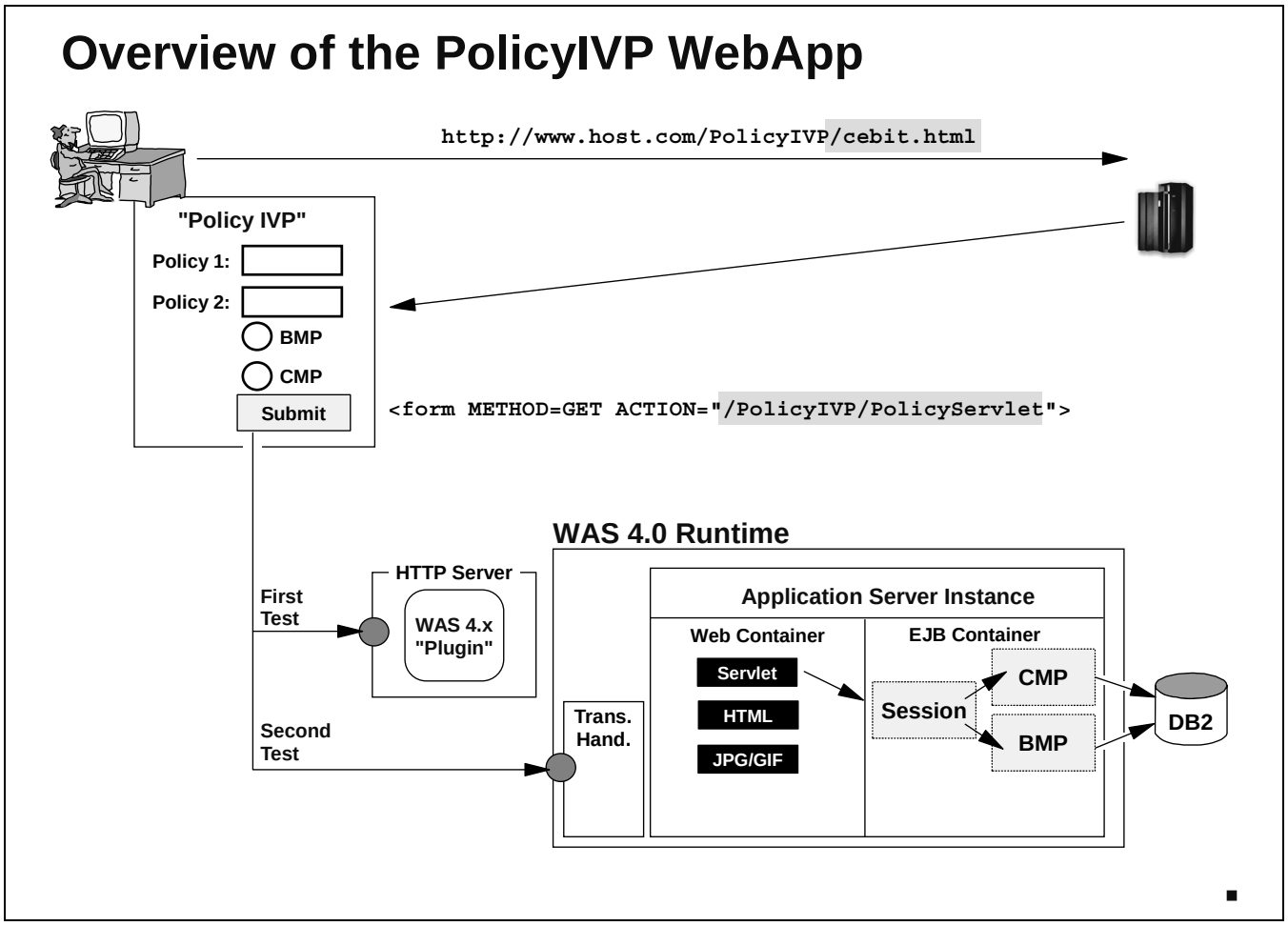
- **One HTTP listener to service any number of application servers**
- **Other HTTP Server functions available (FRCA caching, CGI execution, etc.)**
- **Additional processing overhead**
- **More complex configuration**

Transport Handler

- **Optimized code-path for fast execution**
- **Simple configuration**
- **Strategic direction for WAS 390 HTTP listener**
- **Provides access to only the server in which it's configured**
- **Not intended to be replacement of HTTP Server function**

■

Which HTTP listener you use depends somewhat up to what you're looking to do. Both will work, of course, but the Plugin is a bit more flexible if you have multiple application servers. That's because the Plugin can service all the application servers in a WAS node, while the Transport Handler is designed to support only the application server for which the Transport Handler is configured. The Plugin also has the benefit of all the features of the IBM HTTP Server, such as FRCA cachine and the ability to execute CGI programs or other GWAPI programs. The Transport Handler, on the other hand, is designed for speed; its code paths have been optimized to grab HTTP requests and pass them quickly to the Web Container. The Transport Handler is the strategic direction for HTTP listeners for the WAS web environment, and there will be enhancements in this arena in the future.



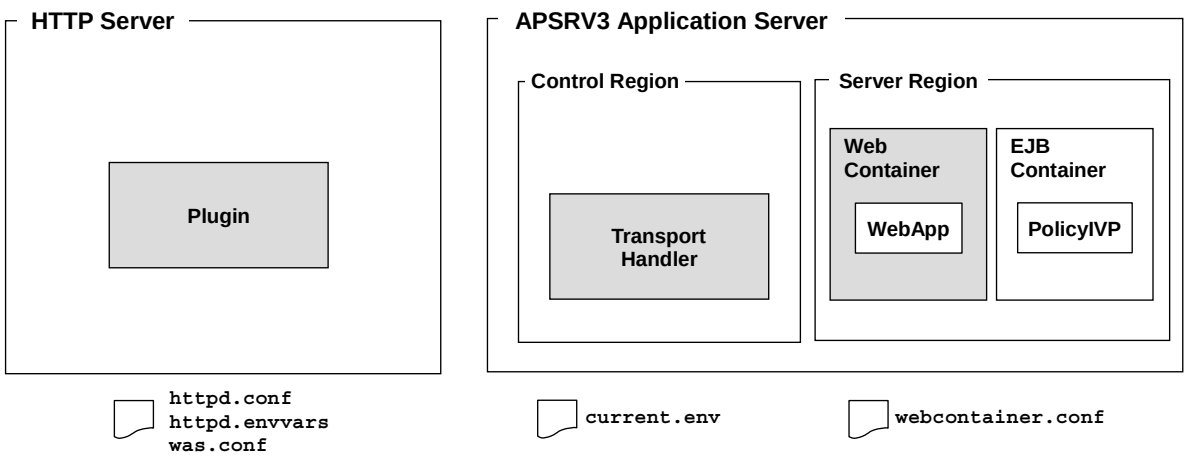
All of that discussion was a prelude to the lab we'll do next. The PolicyIVP application you constructed in the last lab had as one of its components a webapp called PolicyWebApp. That webapp will act as the client to the PolicyIVP application. The webapp has a servlet as well as a handful of static files. One of the static files is called `cebit.html`, and that's the input page on which you'll specify your two policy numbers and your choice of BMP or CMP.

To get the initial HTML page you issue a URL like what's shown at the top of the page. That URL must have the virtual host and context root values for the PolicyIVP application: the Web Container will serve out the static page, and to get to the application to do that you must have the correct virtual host and context root pair.

With the HTML page down on your browser, you may then specify the two policy numbers and your choice of BMP or CMP. In the earlier labs the "fat client" had hard coded values for the policy numbers, and the shell script you used invoked the fat client twice: once for BMP and then again for CMP. Here it's your choice.

With the click of the "Submit" button a URL goes back to the server with the context root and servletmapping string. The policy numbers and BMP/CMP choice go back as hidden variables. The servlet takes the input, then turns and drives the session. The rest goes just like what we discussed earlier. The only difference is the client used out front.

Overview of Lab



PolicyIVP is deployed into APSRV3

PolicyWebApp is deployed into Web Container (needs activating)

In this lab, you will:

- **Configure Plugin and make sure it initializes**
- **Configure Web Container and make sure PolicyWebApp binds to Virtual Host**
- **Test PolicyWebApp as client to PolicyIVP**
- **Configure Transport Handler, shut down Webserver, and test again**

■

And now it's time to go to lab and make all this stuff work.

WSC White Paper on WebApps

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

Configuring Web Applications

- Go to `www.ibm.com/support/techdocs`
- Click on "White Papers" link
- Search for WP100238 under "White Papers"
- 1.5MB PDF file
- 100 pages, table of contents, index, lots of pictures
- Everything covered here, and more

For much more information on configuring webapps, go get this PDF off the IBM technical support website.

End of Document