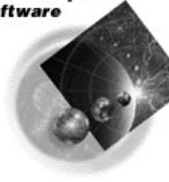


**IBM WebSphere
Software**



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

WSADMIN

Scripting Interface

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

(This page intentionally left blank)

Presentation Based on White Paper



If you're interested in going deeper still, refer to white paper WP100421 on the "Techdocs" website

<http://www.ibm.com/support/techdocs/atσμαstr.nsf/WebIndex/WP100421>

Includes a ZIP file with dozens of exercises.

This presentation won't go into nearly as much detail; rather it'll pick up the key points.

If you want more detail, pull the white paper off Techdocs.

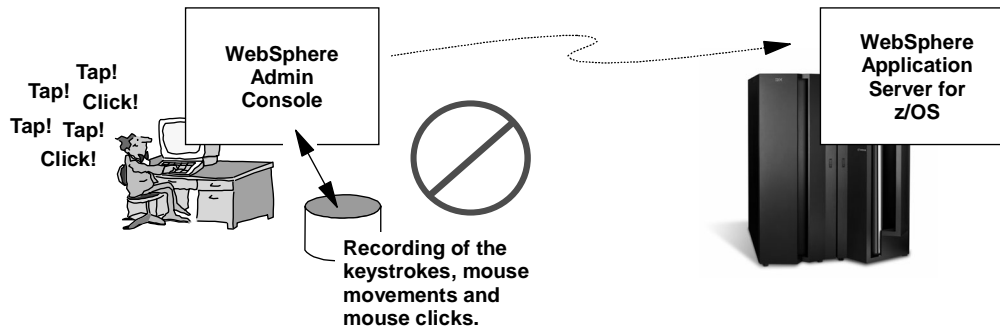
■

To set your minds at ease, this presentation is based on a far more detailed and comprehensive white paper on the WSADMIN topic. That white paper is available at the URL you see on the chart above. The white paper is in the form of a "primer," which is meant to be a step-by-step instruction, or tutorial, on the topic. Included with the white paper is a ZIP file with dozens of exercises to help you see how WSADMIN scripting is done.

So as we go through this presentation, do not worry if all the details are not present ... the white paper has plenty of details.

What WSADMIN is NOT

WSADMIN is not a keyboard activity record-and-playback mechanism.



A common question is whether it's possible to record Admin Console work and use it to create a WSADMIN script.

The answer is "no" ... but that's not necessarily a bad thing. As you'll see, many WSADMIN commands are far simpler than the steps you'd take in the Admin Console to achieve the same thing.

Some, however, are more complex. It's a tradeoff.

What is WSADMIN?

■

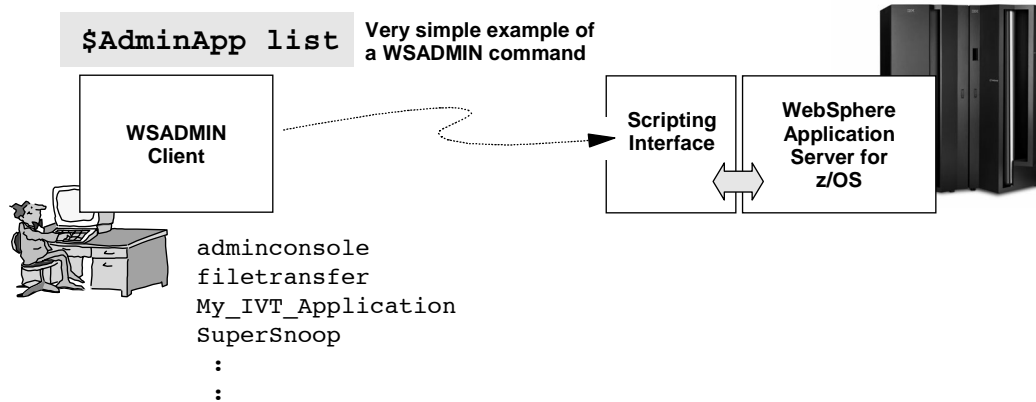
We'll start by dispelling with a misunderstanding many have regarding WSADMIN. What WSADMIN is *not* is a keyboard recorder/playback mechanism. Many people ask if it's possible to "record" the things done on the Admin Console and use that as a WSADMIN script. The answer is "no" because WSADMIN is fundamentally not about "playing back" something. WSADMIN is really more of a programming interface.

As the chart indicates, this is not necessarily a bad thing. Capture-and-playback tools can be quite confusing at times. WSADMIN can be very simple for certain things (though, admittedly, difficult for others).

So what exactly is WSADMIN?

WSADMIN is Scripting Interface

WSADMIN is an interface to WebSphere that allows commands issued to modify some aspect of the runtime environment:



What sort of things can be accomplished?

- Install or uninstall applications
- Modify an existing application
- Start or stop servers
- Initiate node synchronization
- Create new servers, clusters, virtual hosts, etc.

Without realizing it, you may have already used WSADMIN ...

■

What WSADMIN is a programming interface into WebSphere Application Server. Commands executed against this interface may then modify some aspect of the runtime environment. There's quite a few things you can do with WSADMIN, as the chart indicates. An example of a very simple WSADMIN command is the one on the chart:

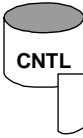
```
$AdminApp list
```

Which will then list out all the applications installed in the cell.

Note: There's quite a bit more we have yet to explain, such as how to execute that command, and what cell it acts against to list out the applications. That's coming.

The chances are good that if you built a WebSphere for z/OS environment, you've used WSADMIN.

You've Probably Used WSADMIN



BBOWIAPP

```
//INST1 EXEC PGM=IKJEFT01,REGION=0M
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
BPXBATCH SH +
/wasv5config/g5cell+
/AppServer+
/bin/wsadmin.sh -conntype none +
-c '$AdminApp install +
/wasv5config/g5cell+
/AppServer+
/installableApps/adminconsole.ear +
{-appname adminconsole +
-MapRolesToUsers {"administrator" ...
{"monitor" No No G5ADMIN G5CFG} +
{"operator" No No G5ADMIN G5CFG} +
{"configurator" No No G5ADMIN G5CFG}} +
-server g5sr01c +
-node g5nodec +
-cell g5cellc +
-copy.sessionmgr.
g5sr01c}' +
1> /tmp/bbowiapp_26921.out +
2> /tmp/bbowiapp_26921.err
/*
```

Submitted before you started the server ... don't need server running to install applications

BPXBATCH invocation of shell script

WSADMIN command, attributes and options

When configuring WebSphere initially, the BBOWIAPP job installed the Admin Console into your new server using WSADMIN

Some interesting things:

- Server wasn't up when you installed application
- Simple BPXBATCH invocation of `wsadmin.sh` shell script
- WSADMIN command and its attributes/options contained in the JCL

We'll explore all of these things in this presentation.



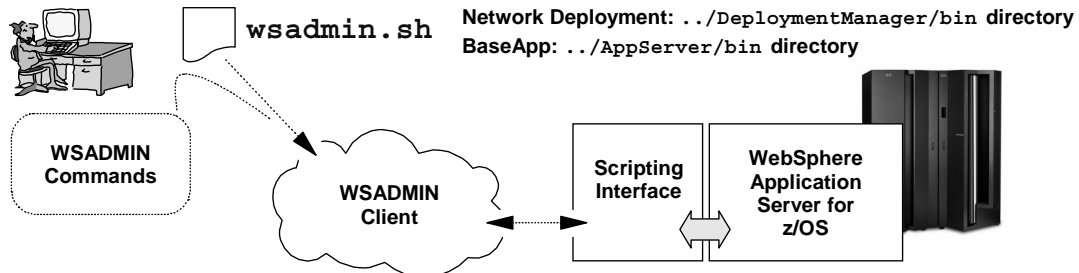
The BBOWIAPP job -- one of the customized jobs generated by the ISPF dialogs -- installs the Admin Console into the newly created server. That job has within it a really good example of WSADMIN at work. Some things of note:

- When you ran BBOWIAPP (or BBODIAPP for the Deployment Manager), your server was *not* up. Yet the application was installed anyway. This helps illustrate the point that a server does not need to be up and running to install an application. For that matter, it's possible for every server in an environment to be down and yet still install an application. How? WSADMIN knows enough about the configuration repository environment to make changes directly to it without requiring a server to be operational.
- The way BBOWIAPP invoked WSADMIN was through BPXBATCH. This helps illustrate a key point about WSADMIN: in its "native" form it's a shell script. That means it can be invoked from OMVS or a Telnet session or in JCL as BBOWIAPP does it.
- BBOWIAPP uses the `$AdminApp` object and `install` method to install a file called `adminconsole.ear`. What follows is a long string of options to the `install` method. Not all the options are required for every application you install -- the `-MapRolesToUser` option is fairly complex and not something you'll probably do when you're first working with WSADMIN.

We have a world of WSADMIN commands to explore ... that's coming in a bit.

WSADMIN "Client"

To exercise the scripting interface you use the WSADMIN client. On z/OS the client comes in the form of a shell script:



This is the bare-bones basics of it. There are a lot of variations on how this is done, which we'll cover. For now, understand three key points:

- WSADMIN client is shell script
- WSADMIN commands passed into client
- Client operates against WebSphere

Where can you run client? ■

The first key building block to this thing is the WSADMIN "client." The client is the `wsadmin.sh` shell script. It is this shell script which accepts the commands you program, and it is this shell script which acts against the interface of the WebSphere runtime environment.

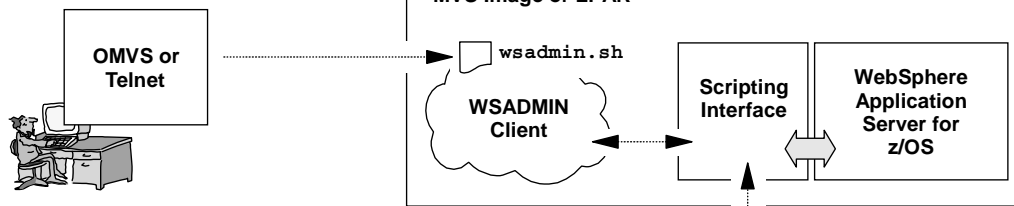
There are actually multiple copies of the shell script found in the HFS. There's a copy under the `/DeploymentManager/bin` directory, and there's a copy under the `/AppServer/bin` directory for each application server.

Where can you run this client? That's covered next.

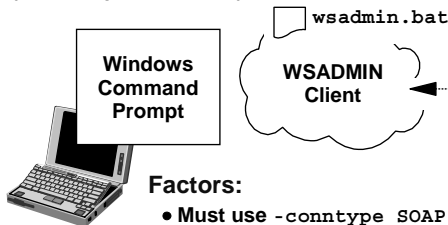
Where You May Run Client

WSADMIN is provided on all WebSphere Application Server platforms. So it's possible to run the WSADMIN client in different places:

On z/OS System:



From Distributed Platform (for example, Windows)



Factors:

- Must use `-conntype SOAP`
- Target server process must be up
- When security on then need to coordinate certificates

Next: two "modes" of operation ...

■

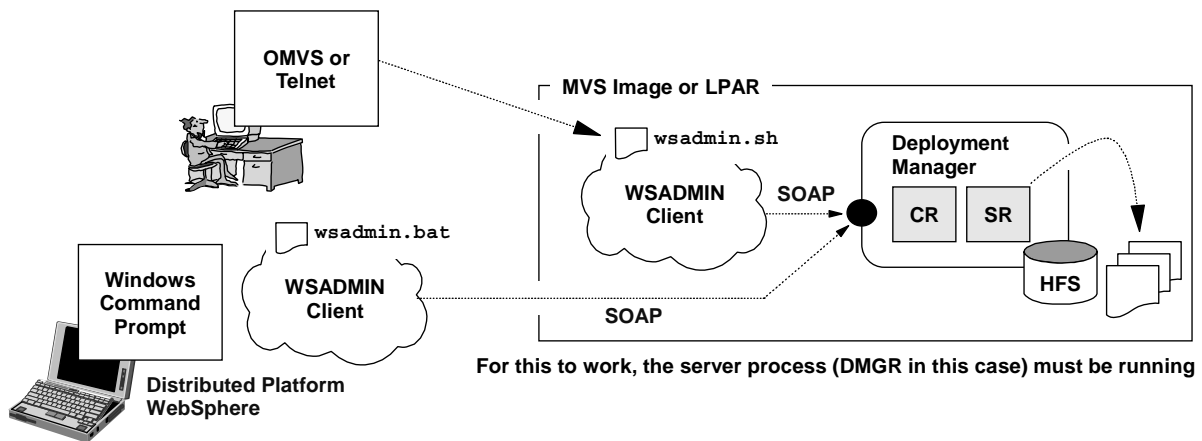
It turns out that WSADMIN is provided on all platforms where WebSphere Application Server runs -- both z/OS and the distributed platforms. It turns out further that you may actually run the WSADMIN client anywhere ... but if you run it on a distributed box there are some restrictions.

Note: What kind of restrictions? We'll go over them in a bit. For now, simply know this: if you invoke WSADMIN on a distributed platform and you want to make changes to the configuration repository on the z/OS box, then you need to go across the network. That means something on the z/OS box needs to be there to receive the requests and act upon them. That "something" is a running server process ... for example, the Deployment Manager. What that means is you must use `-conntype SOAP` (we've not yet talked about that, but will). The key point is this: only when you invoke WSADMIN on the z/OS box can you operate directly against the configuration repository. Invoke WSADMIN on another box and you have to come across the network.

To understand this better we need to explain the two "mode" of operation: local and remote.

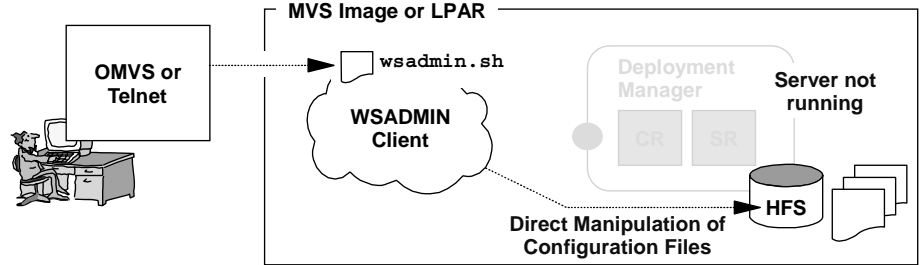
"Local" Mode vs. "Remote" Mode

"Remote" Mode -- Connect via SOAP to server; let server modify configuration files



"Local" Mode --
WSADMIN changes
configuration files
directly

- WSADMIN must run on z/OS
- BBOWIAPP did this
- Some functions not available (\$AdminControl)



There are two basic "modes" in which WSADMIN may run: "local" and "remote." The difference is whether the client -- the shell script -- operates directly against the configuration repository (local) or connects to a server process (remote).

Remote

Remote mode operations is where you instruct the WSADMIN client to issue SOAP messages to a server process, and allow that server process to manipulate the configuration repository. It may seem like that method is only applicable to distributed platform boxes, but in truth you can use remote mode even when invoking the WSADMIN shell script on the z/OS box.

Note: Why would you want to do that? Two reasons: you invoke `wsadmin.sh` on one system in a Sysplex but you want to affect a configuration repository on another system; and because the `$AdminControl` object of WSADMIN is only available in remote mode.

In either case the shell script will issue a SOAP message and aim it at the SOAP port of the server process you designate.

Local

Here is where you invoke the shell script and inform it that it will be operating directly against the configuration repository. No SOAP messages are sent; no network communications are involved.

This mode has some limitations:

- You can't run in local mode on a distributed box and expect to update a configuration repository on the z/OS box. Local mode can only be accomplished when the configuration repository and the shell script client reside on the same system.

Note: What about network mapped drives and things like that? Here's the rule of thumb: if the configuration files appear to be "local," then you can use local mode. If you use a network mapped drive to make a distant repository "appear" local, that's good enough for WSADMIN.

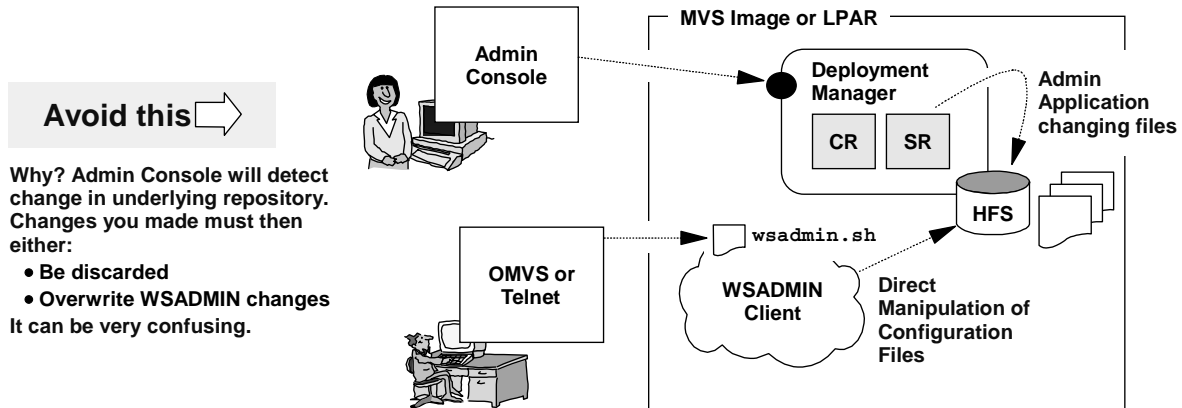
- Much of the `$AdminControl` functionality is not available to you if you invoke in local mode.

The BBOWIAPP job used "local" mode to install the Administrative Application. It had to ... the server wasn't yet running, so it had no server process to send SOAP messages to even if it wanted to. But local allowed it to "install" the application -- add directories and files to the HFS and update key XML files to reflect the new application -- even when the server was down.

There are some things to be aware of when operating in local and remote mode ...

When "Local" vs. "Remote"

Rule of Thumb: If Deployment Manager (or AppServer if BaseApp) is available, connect to it ("Remote"). If server process not available, then use "Local."



Avoid this →

Why? Admin Console will detect change in underlying repository. Changes you made must then either:

- Be discarded
 - Overwrite WSADMIN changes
- It can be very confusing.

If you come in "remote," the server running the administrative service can handle (to some degree) two different forces working against the configuration repository. But it has to know about WSADMIN doing it, and it can't if WSADMIN is operating in "local" mode.

"...to some degree..." -- Some configuration buffering does occur. Based on timing, it's possible changes in one environment won't be "seen" in the other.

Generally speaking, even in remote mode you should avoid having the Admin Console working against repository at the same time WSADMIN is doing it.

■

There's a rule of thumb you should employ when considering whether to use "local" or "remote" mode. It is this:

If the Deployment Manager server is up and running (or application server for BaseApp configuration), then you should use "remote" mode and connect to the server using `-connType SOAP`. In particular, what you want to avoid is the case where someone is using the Admin Console to make changes to the configuration at the same time someone else is using WSADMIN in "local" mode to directly change the configuration repository. The reason for this is because the Admin Console will have no idea that WSADMIN is working on stuff beneath the covers. Two things may happen:

- WSADMIN's changes may fail if the configuration has been changed by the person operating the Admin Console, or
- The person at the Admin Console may be alerted to the fact that the underlying configuration has changed. The Admin Console will then ask the user what they want to do: overwrite changes in repository or discard changes made in the Admin Console.

In either case it's a very disconcerting message to receive.

When you connect to the server's SOAP port using `-connType SOAP`, you at least give the administrative management function of the running server process some degree of coordination. But you should be careful even in this mode ... having both forces working against the configuration can result in problems, so in general you should not have both operating at the same time.

On z/OS, Run Under "WAS Admin ID"

In order to have access to the configuration directory structure, `wsadmin.sh` must run under the authority of the "WebSphere Administrator ID"

If Telnet or OMVS:

```
EZYTE27I login: USER1
EZYTE28I user1 Password: xxxxxxxx
:
:
USER1:/u/user1-> su g5admin
Enter the password for g5admin: xxxxxxxx
USER1:/u/user1-> cd /wasv5config/g5cell/DeploymentManager/bin
USER1:/wasv5config/g5cell/DeploymentManager/bin-> ./wsadmin.sh ...
```

Switch users to the
WebSphere Admin ID

If JCL

```
*****
//WSADMIN JOB (ACCTNO,ROOM), 'USER1',
// USER=G5ADMIN, PASSWORD=xxxxxxx
//*****
//INST1 EXEC PGM=IKJEFT01,REGION=0M
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
BPXBATCH SH +
  /wasv5config/g5cell+
  /DeploymentManager/bin/wsadmin.sh ...
```

Provide authority
via JOB card

Any UID=0 ID? It'll work,
but it may affect file
ownership. Better to use
WAS Admin ID.

This is different from the
issue of authentication
when "Global Security"
enabled. More on that at
end of presentation.

■

When running WSADMIN on the z/OS box (as opposed to invoking it on a remote platform), you should take care to make sure the process runs under the authority of the "WebSphere Administrator ID" of the cell against which you'll work. This is to give the WSADMIN process sufficient authority to read and write into the directories.

Note: A UID=0 ID would work -- it would have sufficient authority -- but you run the risk of modifying a file ownership, which may cause problems later. Better to use the "WebSphere Admin ID."

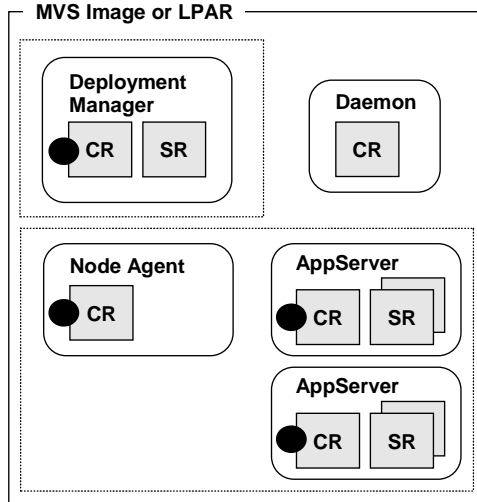
How you accomplish this depends on how you invoke WSADMIN:

- Via Telnet or OMVS -- the thing to do here is "switch user" to the WebSphere Admin ID prior to invoking WSADMIN. That will provide the shell script with the proper authority to do its work.
- Via JCL -- here you simply make sure that the job itself runs under the WebSphere Admin ID authority.

Be careful -- this is not the same thing as authenticating the WSADMIN client to the SOAP port when global security is enabled for the server. We'll cover that topic later. Here what we're referring to is simply the authority to access files and directories.

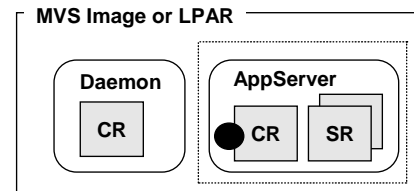
Network Deployment vs. BaseApp

In a Network Deployment configuration, there are many different SOAP ports to which WSADMIN could connect:



Rule of Thumb: Connect to Deployment Manager. That'll then update "master configuration"

A Base Application Server node has only the application server, so that's what you'd connect to in "Remote" mode:



Message:

- If ND, connect to DMGR
- When it comes to basics of WSADMIN, Network Deployment or BaseApp are essentially the same
- Going forward in this presentation we'll assume ND

■

Let's stay on the topic of connecting via the `-connntype` SOAP option, and explore what server processes we have available to us.

Network Deployment Configuration

In a ND configuration we have several server processes that have SOAP ports: the Deployment Manager, all Node Agents, and each application server.

Even though all those server types have SOAP ports, you should only connect to the Deployment Manager's port. The other servers will permit you to connect, but the functionality will be limited. Only the Deployment Manager has the full range of administrative capabilities, and only the Deployment Manager has the "master configuration" under its control.

Base Application Server Node Configuration

There's only one SOAP port available: the application server.

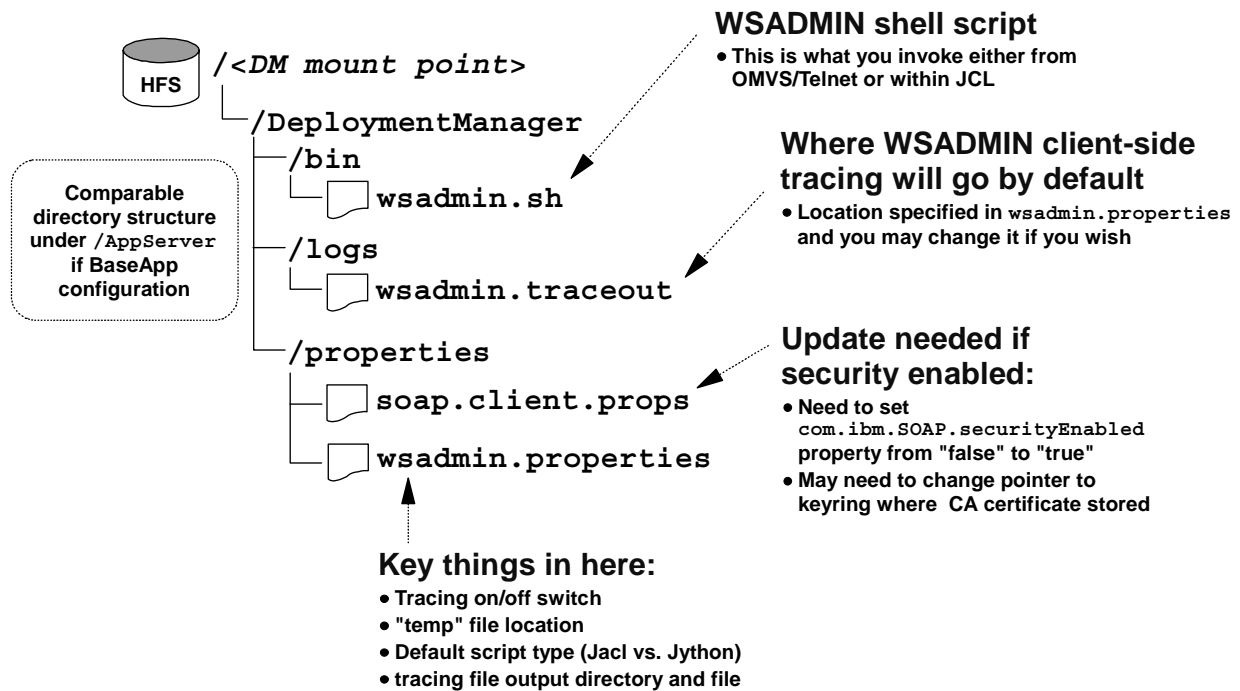
There are some key messages this chart is trying to deliver:

- If Network Deployment, connect to the Deployment Manager

Note: If there's a good reason to connect to one of the other server types when the Deployment Manager is available, we've not heard of it.

- When it comes to discussing WSADMIN, there's really little difference between an ND configuration and a BaseApp (except for things like synchronizing nodes). So we're going to assume that from this point forward we're talking about an ND configuration, unless explicitly stated otherwise.

Other Files To Be Aware Of



Besides the shell script itself, there are some other files you should be aware of in the HFS heirarchy:

- **wsadmin.sh** -- the shell script itself
- **wsadmin.traceout** -- this file is where tracing will go by default. The location of the tracing directory and file is specified in the wsadmin.properties file. By default it'll be this file. The *amount* of tracing is determined in the wsadmin.properties file as well.
- **soap.client.props** -- this file comes into play if you're looking to access the SOAP port when global security is enabled.
- **wsadmin.properties** -- this file has a bunch of things in it, but most importantly the four things shown on the chart above.

This directory structure is found on all the platforms, so you should not think this is just a z/OS thing. All the platforms utilize the same WSADMIN architecture.

What About Security?

Do you have Global Security enabled?

This has no impact when WSADMIN run in "local" mode

Some things *do* change:

- Need to pass `-user` and `-password` in on invocation of remote WSADMIN
- Need to make sure WSADMIN has access to keyring with proper CA certificate
- If WSADMIN on distributed platform, you'll need to make sure trust file there has CA certificate

More on this at end of presentation

Key Message: scripting *itself* is not affected when security enabled -- only access to scripting interface

If you enable the "global security" function of the WebSphere Application Server runtime environment, that affects the ability of WSADMIN to gain access to the runtime. We're going to cover this topic in more detail at the end of this presentation, but a few notes are worthwhile right here:

- Enabling global security has no impact on WSADMIN when WSADMIN running in "local" mode. Global security is going to impact the ability to access services through the SOAP port. Running in "local" mode means WSADMIN is operating against the configuration HFS directly. There's no SOAP port being accessed. In fact, the server doesn't even need to be up for "local" mode to work.
- When global security is enabled, it'll mean you have to pass in `-user` and `-password` on the command used when connecting to the SOAP port.

- Notes:**
- There is a way to code this userid and password in the `soap.client.props` properties file located in the `/properties` directory. If you do that, then you don't have to pass `-user` and `-password` on the invocation.
 - Remember that this provides a way to authenticate the WSADMIN client to the SOAP port. This is something different from running the process under the WebSphere Admin ID -- that was done to provide read/write access to the HFS. Keep separate access/authentication separate from read/write permissions. They're two different things.

- Because SSL is involved with global security being enabled, the topic of "keyrings" and "certificates" comes into play. To enable the SSL handshake it is important to make sure that WSADMIN has in its keyring the proper CA certificate. If you're running WSADMIN on a distributed machine then it involves exporting that CA certificate from WebSphere's keyring and importing it into the trustfile on the distributed platform.

This whole "global security" thing involves authentication and access into the "front door" of WebSphere. Once inside, the scripting is the same as when global security is disabled. In other words: the same.

Syntax of WSADMIN Invocation

```
./wsadmin.sh -?
```

```
wsadmin
```

```
[ -h(elp) ]
[ -? ]
[ -c <command> ]
[ -p <properties_file_name> ]
[ -profile <profile_script_name> ]
[ -f <script_file_name> ]
[ -javaoption java_option ]
[ -lang language ]
[ -wsadmin_classpath classpath ]
[ -conntype
  SOAP
  RMI
  JMS <jms_parms> |
  NONE
]
[ script parameters ]
```

Used to indicate WSADMIN commands follow. We illustrate that in a few charts.

Used to tell WSADMIN that script file is in EBCDIC rather than default ASCII

Used to point to a file in which the commands are held. We illustrate that after -c switch.

Note: if connecting via SOAP and global security is enabled, provide the WebSphere Admin ID and password on the invocation.

Used to indicate the type of connection -- "Remote" (-conntype SOAP) or "Local" (-conntype NONE)

Finally we get to the point where we can show the command syntax used to invoke WSADMIN. We know that the client comes in the form of a shell script. The shell script accepts parameters, one of which is `-?`, which provides the syntax shown here.

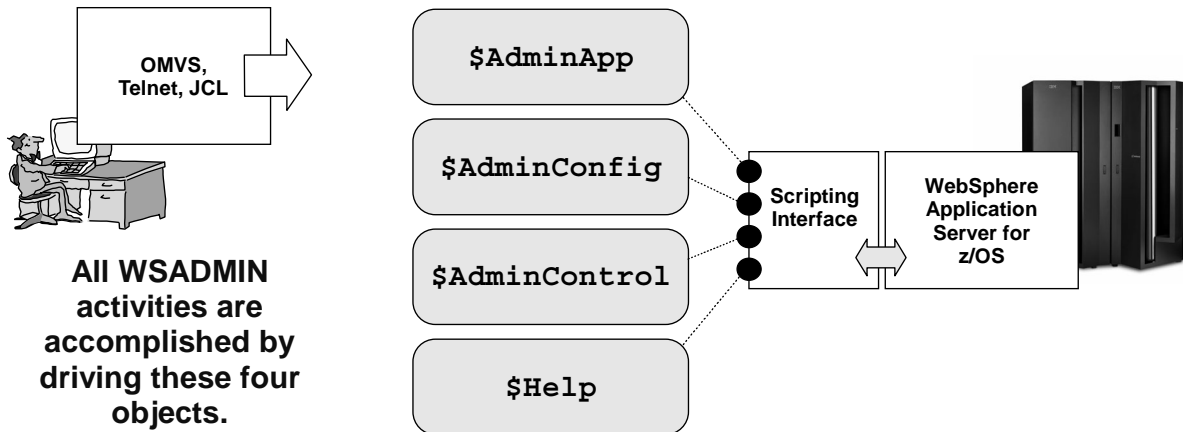
- The `-c` switch is what's used to pass WSADMIN commands "inline" with the invocation of the shell script. The alternative way to process commands is in a separate file, which is done with the `-f` switch.
- The `-f` switch points to a file that contains the WSADMIN commands. When your command set is long and complicated, this is the way to go. Short commands are easily done with the `-c` switch.
- The `-javaoption` switch is needed when invoking WSADMIN on the z/OS system and the file you're pointing to is in EBCDIC. WSADMIN by default assumes an ASCII file, and the `-javaoption` switch is used to indicate the EBCDIC file encoding.
- The `-conntype` switch is used to indicate if the mode is "local" (value of `NONE`) or remote (value of `SOAP`). If `-conntype SOAP`, then additional parameters are needed, such as `-host` and `-port` to indicate *what* SOAP port to connect to, and `-user` and `-password` if global security is enabled on that SOAP port.

So, for example, a command to invoke WSADMIN might look like this:

```
./wsadmin.sh -conntype SOAP -host www.myhost.com -port 9200 -c '$AdminApp list'
```

That would imply "remote" mode, connected to port 9200 on host `www.myhost.com`, and issue the command `$AdminApp list`.

Four WSADMIN Program "Objects"



Each has many different "methods," attributes and options:

Object	Method	Attribute	Options
\$AdminApp	uninstall	My_IVT_Application	
\$AdminApp	install	/u/user1/MyIVT.ear	{-server G5SR01C -node G5NODEC}

Good deal of the learning curve is discovering the syntax of these methods

More examples coming

Let's start to explore the command set of WSADMIN. At the heart of it are four "objects" -- in other words, four primary commands. (The term "objects" is technically accurate, as the WSADMIN scripting interface is *object-oriented*.)

- **\$AdminApp** -- This object is used to affect applications, such as installing them, modifying them, or removing them from the configuration.
- **\$AdminConfig** -- This object is used to affect the configuration, such as creating new servers or cluster, or modifying the attributes of a configuration object.
- **\$AdminControl** -- This object is used to control the objects, such as starting or stopping servers or applications. This object is limited in function when in "local" mode.
- **\$Help** -- This object is used to provide online help.

Each object has a bunch of "methods" that are used to tell WSADMIN exactly what you're doing. Those methods in turn have "attributes," which provide more detail about what action is desired. Finally, "options" on the attributes provide the finest level of detail. A command may have just the object and a method (for example, \$AdminApp list), the object, method and attribute (\$AdminApp uninstall My_IVT_Application), or object, method, attribute and options (example shown at the bottom of the chart above).

We'll have lots more examples later in this presentation, plus the WP100421 white paper has dozens and dozens of examples.

"Inline" Commands

Interactively at WSADMIN prompt



```
./wsadmin.sh
:
WASX7029I: For help, enter: "$Help help"
wsadmin> $AdminApp list
```

Passed in as parameter on shell script invocation



```
./wsadmin.sh -c '$AdminApp list'
```

Parameter for shell script, but processed in JCL



```
BPXBATCH SH +
/wasv5config/g5cell+
/AppServer+
/bin/wsadmin.sh +
-c '$AdminApp list' +
:
/*
```

All three of these are more or less the same thing

Great for relatively simple things, such as:

- listing installed applications
- uninstalling an application
- installing an application with a small set of options
- Exploring the "help" option -- getting information about an option, etc.

But as the input gets more complex, you want to keep things in a separate file ...

Let's show a couple of ways you could provide the commands just illustrated on the previous chart "inline" when invoking WSADMIN. (By "inline" we mean not in a separate file pointed to by the `-f` switch.) The chart above illustrates three different ways the simple `$AdminApp list` command could be passed in:

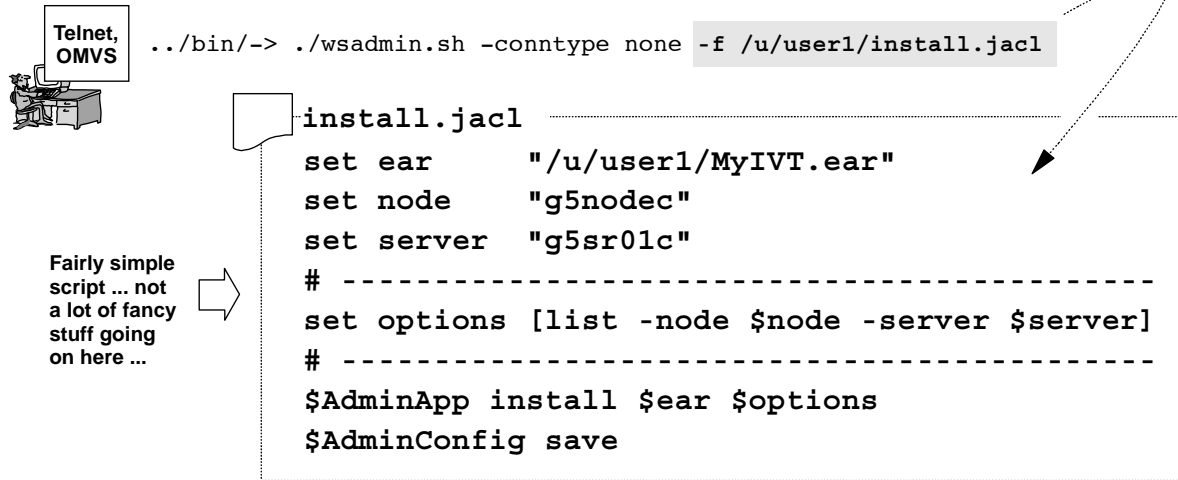
- **Interactively** -- if you invoke WSADMIN and *do not* provide `-c` or `-f`, it will provide a WSADMIN command prompt at which commands can be entered.
- **With `-c` switch** -- here the `wsadmin.sh` shell script is invoked, and the `-c` switch is used to indicate that a command is following. WSADMIN will come up, execute the command, and then stop.
- **In JCL** -- the example here is essentially the same as the previous one, except that WSADMIN is invoked with BPXBATCH out of JCL.

Note: All of those are essentially the same thing ... the only difference is how WSADMIN is invoked.

Issuing command "inline" like this has its purpose -- its great for relatively simple things. But as the command set gets more complex, you'll find yourself wanting to put things in a separate file.

Files Containing Script

It's called a "scripting interface" because scripting languages like "Jacl" and "Jython" can be used to drive the WSADMIN commands:



Script processing allows:

- passing in parameters
- logic tests (if-then-else)
- built-in functions (count, length, string, etc.)
- error checking and handling

Notes:

- "Jacl" is Java-based version of "Tcl" scripting language
- "Jacl" is default script-type expected
- Support for "Jython" in WebSphere for z/OS Version 5.1

■

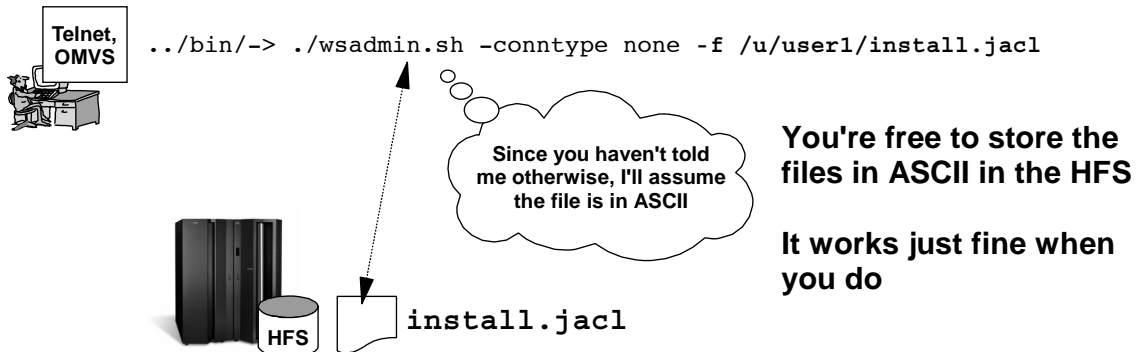
The -f switch of WSADMIN will allow you to point to a separate file containing commands. The -f switch takes as a parameter the path and file name of the file containing the commands. By default, WSADMIN expects to see a scripting language known as "Jacl."

Note: "Jacl" is a form of the older "Tcl" language. WSADMIN also supports "Jython," which is a more "object-oriented" language. That support was made available with WAS V5.1.

The chart above illustrates an example of a script file used to install an application. It's a relatively simple script file ... the sample scripts in the InfoCenter are much more complicated than this. We'll analyze this script in a few charts. For now, simply understand that the -f switch points to the file, and the file contains script used to install the application.

WSADMIN Expects ASCII Script File

Be aware that WSADMIN client on z/OS expects -- by default -- for script files to be in ASCII encoding:



If file is really in EBCDIC and WSADMIN expects ASCII, it'll fail. But there is a way to tell WSADMIN that the file is in EBCDIC:

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047 -conntype none -f /u/user1/install.jacl
```

The `-javaoption` switch is used to pass in the type of encoding used by the script file

■

One quick note about using a separate script file with WSADMIN. By default, WSADMIN expects that file to be in ASCII. On a distributed platform box that would be the default encoding for files. But on z/OS, the default encoding for files in the HFS is EBCDIC. Therefore, you have two options:

1. Create the script files on your workstation and then upload them to the z/OS HFS in binary mode so they hit the HFS as an ASCII file. Then simply point to them with the `-f` switch and WSADMIN is happy.

Note: Of course, it makes editing those files on the z/OS box a bit more cumbersome. But there are ASCII editors for files resident on z/OS HFS.

2. Use the `-javaoptions` switch to indicate to WSADMIN that the file pointed to by the `-f` switch is an EBCDIC file. The parameter supplied on the `-javaoption` switch will be `-Dscript.encoding=Cp1047`.

Then you can have the file in the HFS in EBCDIC, and use the standard editor to make minor modifications as needed.

Note: There's an issue with square bracket encoding you need to be aware of. See WP100421 for details.

The \$AdminApp Object

`$AdminApp help`

```
edit
editInteractive
export
exportDDL
help
install
installInteractive
list
listModules
options
publishWSDL
taskInfo
uninstall
updateAccessIDs
deleteUserAndGroupEntries
```


`$AdminApp help install`

WASX7096I: Method: install

Arguments: filename, options

Description: Installs the application in the file specified by "filename" using the options specified by "options." All required information must be supplied in the options string; no prompting is performed.

The AdminApp "options" command may be used to get a list of all possible options for a given ear file. The AdminApp "help" command may be used to get more information about each particular option.

How can you know what options are valid? 

Simple Example:

```
$AdminApp install /u/user1/MyIVT.ear {-node g5nodec -server g5sr01c}
```

Object Method Filename Options

Let's look at the \$AdminApp object. It has a number of methods associated with it, and you can see those methods by using the help method. That'll spit back a list of the methods on the object. (All four objects -- \$AdminApp, \$AdminConfig, \$AdminControl and even \$Help -- have a help method, by the way.) Further, you can use the help method to give you information on any of the other methods. The chart shown an example of this ... help is used to give information about the install method.

The example shows how this pieces together:

- \$AdminApp is the object
- install is the method
- We see from the help that the syntax of the install method that two parameters are passed in: the filename of the EAR file being installed, and any options

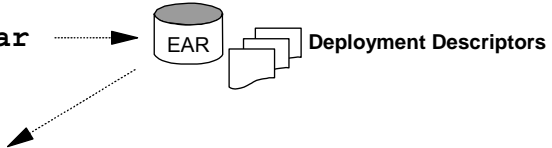
The chart shows an option list of:

```
{-node g5nodec -server g5sr01c}
```

Where in the world did that come from? How can we know what options are valid for a given EAR file? We can use the options method ...

\$AdminApp Options

The `options` method of `$AdminApp` can be used to list back the tasks (or options) that are valid for a given EAR file:

`$AdminApp options /u/user1/MyIVT.ear` → 

WASX7112I: The following tasks are valid for "/u/user1/MyIVT.ear"

```
BindJndiForEJBNonMessageBinding
MapEJBRefToEJB
MapWebModToVH
MapModulesToServers
```

```
:
server
cluster
cell
node
:
appname
verbose
contextroot
:
defaultbinding.force
defaultbinding.strategy.file
```

Two shown
on the
previous
chart

You can use `help` to list back general information on each of these:

```
$AdminApp help appname
```

WASX7232I: "appname" option; use this option to specify the name of the application. The default is to use the display name of the application.

The InfoCenter is helpful in determining syntax of these options.

Let's look at a simple example and start the discussion on Jacl scripting

■

The `options` method can be used to interrogate an EAR file and have it tell you what options are applicable. The syntax is fairly simple and is shown in the chart above.

Note: Where would you issue this command? This gets back to the topic of "inline" commands. Here's a good example of simple, ad-hoc type of command where invoking WSADMIN in "local" mode (no SOAP connection) and getting a command prompt would be handy. If all you're doing is exploring the `help` function for various things, this'll work just fine.

What happens when you do this is this: WSADMIN will rummage around in the XML files of the EAR file and determine what's in the EAR. Then it'll display back the options that apply to the EAR.

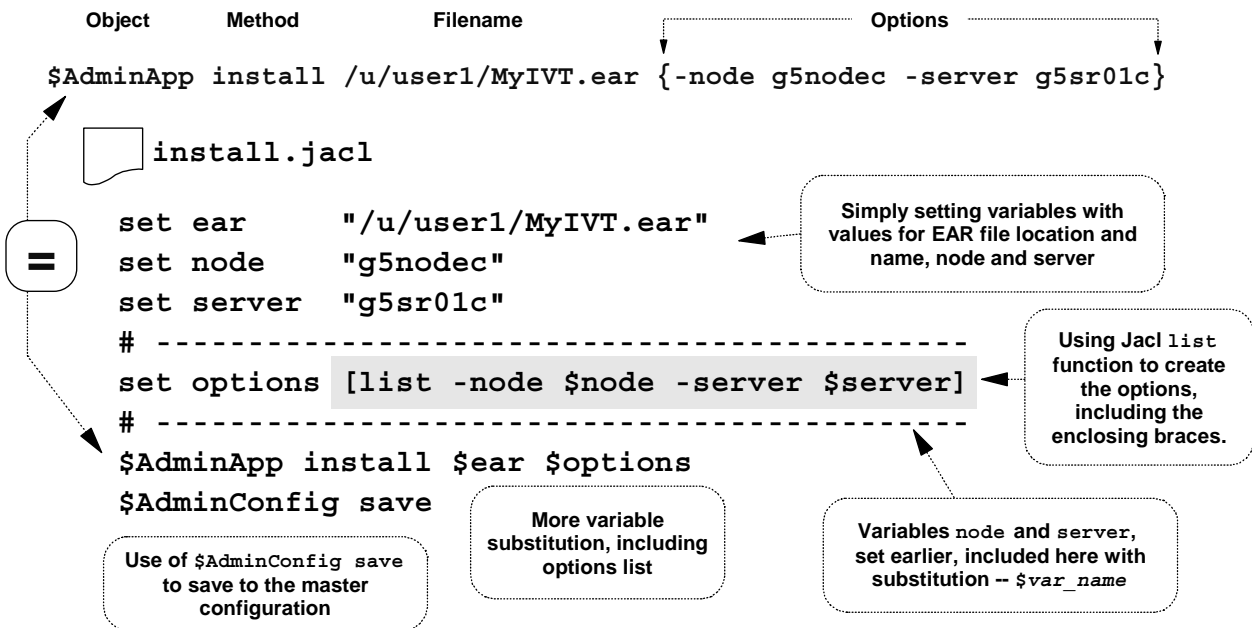
Note: Not all the options are required ... this is just a list of what *may* be used with that particular EAR file.

Here's where the WebSphere Application Server InfoCenter comes in real handy. There's a wealth of examples in there, and the whole thing is searchable. We would strongly encourage you to make use of the InfoCenter for specific coding examples for the methods of WSADMIN.

Let's now turn to Jacl scripting in particular. There are some fundamental things that you must know to do this stuff.

Jacl Scripting Basics

Let's show how simple `$AdminApp` command can be coded in Jacl:



Now simply point to this file either on command line or from JCL. Change variables to install different application or install into different server ...

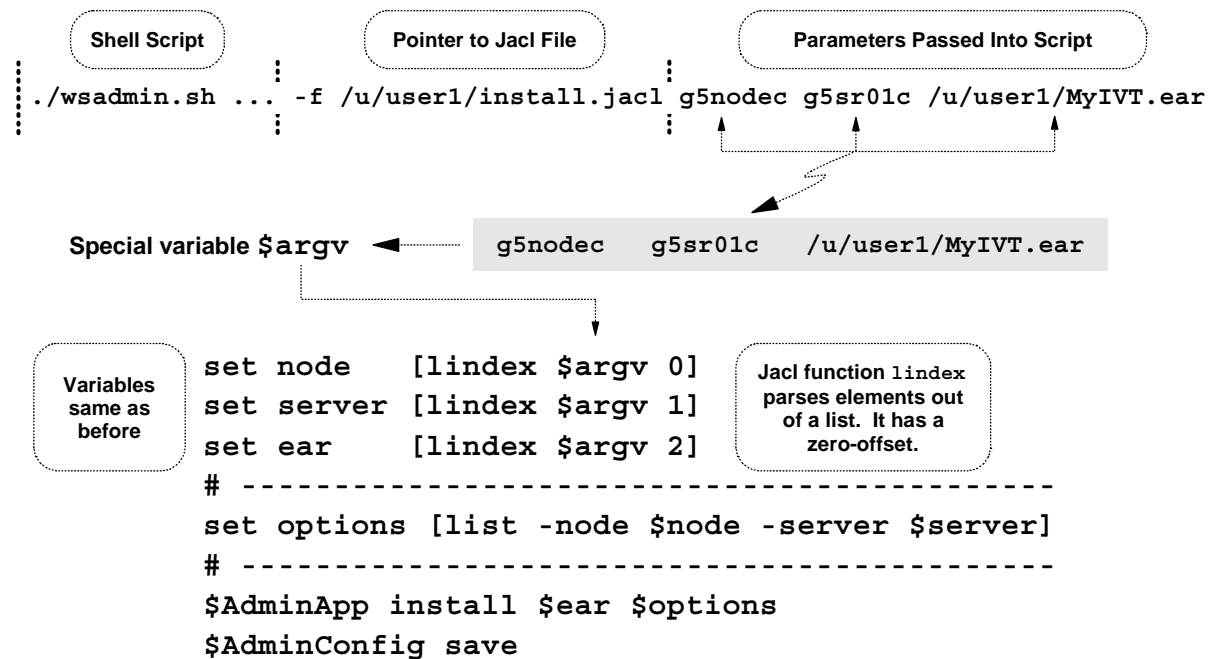
Let's take a closer look at what's going on inside this Jacl script file. At the top of the chart you see the command we're looking to code inside the file. Below is Jacl that shows how that command could be coded more flexibly.

Note: You *could* simply code the exact same string in the file. You don't *have* to use the variables and other things we're showing you here. But the power of Jacl script is in the variables, logic, etc.

- It sets three variables at the top of the file: `ear`, `node` and `server`. Those will be options needed on the `install` method, and rather than hard-code those values down in the command, we'll set them as variables at the top. That'll allow changes to be made more easily.
- The Jacl `list` function is used to construct a special kind of variable. The variables for the `node` and `server` are used in the construction of this special variable. A "list" variable -- in this case we're creating a variable called `options` that'll be a list variable -- is one where the elements in the variable are delineated and easily extractable, unlike a string variable which is just a stream of characters. We'll see more of the `list` function later.
- The `$AdminApp install` command is constructed, with the EAR file and options being passed in as variables.
- Finally, the `$AdminConfig save` command is executed to save the changes.

Next we'll explore passing parameters into the Jacl script. This will make it even more flexible.

Passing Arguments into Jacl Script



Jacl script is now "generic" and can be used to install any EAR file into any server ... simply by passing in parameters.

Here we have something very similar to what we had on the previous chart. But there's a difference: on the invocation of the `wsadmin.sh` shell script, we're passing in parameters after the `-f` switch and the file we're pointing to. When we do that, those values are passed into the Jacl script as parameters.

Note: Go back to the chart titled, "Syntax of WSADMIN Invocation" and look at the very bottom of the syntax chart. You see the following: `[script parameters]`. The three strings that follow the Jacl script file on the invocation line -- `g5nodec`, `g5sr01c` and `/u/user1/MyIVT.ear` ... separated by spaces -- are passed in as parameters.

How do we handle those parameters? First you must understand that once inside the Jacl script, those parameters are held inside a special variable (a "list" variable, by the way) called `$argv`. So all we need to do is parse those parameters out and then use them elsewhere in the Jacl. We do that with a Jacl function called `lindex`. `lindex` parses out based on an offset number. Jacl list functions have a zero offset, so the first parameter in the list is parsed out with a 0.

Once parsed out, what do you do with the parameters? Place them into other variables, which have the same names we used before. From there the script is exactly the same. But the Jacl script is now generic ... you can use it to install a simple application into any server on any node.

Notes:

- The script above does not do any error checking for validity of the passed-in parameters
- The script above does not do any checking for the validity of the sequence of the parameters
- The script above does not synchronize the changes to the nodes, which is necessary in a Network Deployment configuration

All those things are *possible*, just not shown in this example. See WP100421.

Nested Options

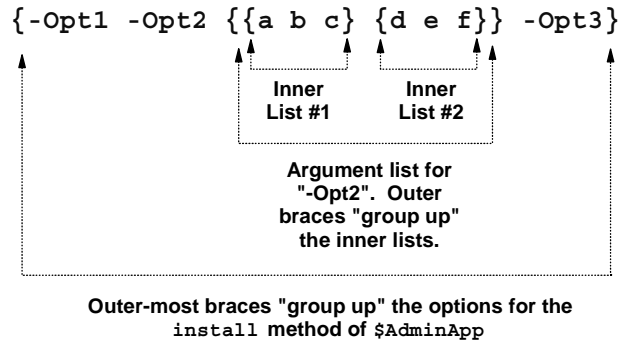
Some options have their own options ... which means it becomes necessary to nest option lists inside of other options:

```

BBODIAPP
(Installs Admin Console into DMGR)
:
{-apptime adminconsole
-MapRolesToUsers {
  {"administrator" No No G5ADMIN G5CFG}
  {"monitor" No No G5ADMIN G5CFG}
  {"operator" No No G5ADMIN G5CFG}
  {"configurator" No No G5ADMIN G5CFG}
}
-server dmgr
-node g5dm
-cell g5cell
}
:

```

Simplified Schematic Diagram:



Two ways you can build this with Jacl:

```

set inner_1 [list a b c]
set inner_2 [list d e f]
1 set Opt2_arg [list $inner_1 $inner_2]
set options [list -Opt1 -Opt2 $Opt2_arg -Opt3]

2 set options [list -Opt1 -Opt2 [list [list a b c] [list d e f]] -Opt3]

```

Build up the nested lists, starting from the inner-most and working outwards

Nesting list functions inside one another

This is one of the most challenging aspects of WSADMIN and Jacl -- understanding exact structure of option syntax, and matching up the braces.

■

Now things get a bit more complex. It turns out that some options have their own options. That means that the Jacl coding needs to *nest* the options lists. If you look closely at the BBODIAPP job, you'll see that nested options are occurring. `-MapRolesToUsers` is an option, and it is supplied with four more options -- one to define the "administrator" role, one to define the "monitor" role, etc.

Look at the chart to the right of the BBODIAPP example ... it shows a simplified example of a nested option list. The focus is on the braces -- or "squiggle brackets" -- and how they must match up. When analyzing these things, it's best to work from the inside and then to the outside:

- Look at the center of this ... we have the option `-Opt2`. It has two option *lists*, `{a b c}` and `{d e f}`. (This is similar to the actual example of `-MapRolesToUsers`.) The braces around those delimit each option list, and then another set of braces are used to delimit all the options lists for `-Opt2`.
- As we move outward, we see that `-Opt1` and `-Opt3` do not have option themselves ... they're simple options that stand all by themselves. But we do need to delimit the extent of the options overall, which is done with the outer-most braces.

There are three ways you could code that up in Jacl. One is the code the whole string literally, without any variable substitution. This gets awkward quickly, as it's difficult to break lines in Jacl. The better way is to construct the string using the `list` function. Even here we have two approaches:

1. Construct the list step-by-step. This is perhaps the easiest method. Start with the inner-most lists, build those, then imbed those in outer lists.
2. Nest the `list` function inside other `list` functions on the same line. This works, but it can very quickly get confusing.

\$AdminConfig Object

`$AdminConfig` is used to create, modify or delete things in the configuration. This object has quite a few methods:

`$AdminConfig help`

```

attributes          required
checkin             reset
convertToCluster   save
create              setCrossDocumentValidationEnabled
createClusterMember  setSaveMode
createDocument      setValidationLevel
installResourceAdapter  show
createUsingTemplate  showall
defaults            showAttribute
deleteDocument      types
existsDocument      validate
extract
getCrossDocumentValidationEnabled
getid
getObjectName
getSaveMode
getValidationLevel
getValidationSeverityResult
hasChanges
help
list
listTemplates
modify
parents
queryChanges
remove

```

Further, these methods operate against configuration "types" -- specific configuration objects such as server, clusters and many more.

`$AdminConfig types`

```

AdminService
Agent
:
WASQueueConnectionFactory
WASTopic
WASTopicConnectionFactory
WebContainer
WebModuleConfig
WebModuleDeployment
WorkloadManagementServer

```

255 Total!

When you create or modify part of the configuration, you'll be working against a "type"

Let's switch to another object -- `$AdminConfig`. This object is used to affect configuration elements that are not applications. There's quite a few methods to this object, and using the help function lists them. But you won't see names there that look like configuration elements ... the names of the methods are *verbs* -- things you *do* to things in the configuration. The target of the configuration change is something known as a "type". To see all the types that are available you can issue the command `$AdminConfig types`. You get a long list ... 255 items long.

There's far too many "types" to cover in this presentation, so we'll focus on one ... `VirtualHost`.

Exploring VirtualHost Type

First, use `attributes` method to list out the possible attributes for `VirtualHost`:

```
$AdminConfig attributes VirtualHost
```

```
"aliases HostAlias*"
"mimeTypes MimeEntry*"
"name String"
```

Three attributes:

- `aliases` -- asterisk on "HostAlias" indicates there's more to this
- `mimeTypes` -- asterisk indicates there's more
- `name` -- no asterisk: this is lowest level. "name" is attribute, a text string is its value

Next, drill down on the `HostAlias` type with `attributes`:

```
$AdminConfig attributes HostAlias
```

```
"hostname String"
"port String"
```

Two attributes:

- `hostname` -- a string value
- `port` -- a string value

Finally, use `required` to determine minimum settings:

```
$AdminConfig required VirtualHost
```

Attribute	Type
name	String

You can get away with only the "name" attribute. `VirtualHost` won't actually work, but `WSADMIN` will allow it be created.

Let's see example of actual `$AdminConfig` command to create a new `VirtualHost` ...

■

We're focusing on `VirtualHost` because it's relatively easy to understand. It's as good as any type to use to demonstrate the way in which you explore and learn more about how to construct your command structure. We know that the string `VirtualHost` is a "type" because it's one of the 255 or so that were spit out when we ran `$AdminConfig types`.

To determine what attributes are on this configuration type, we use the `attributes` method of `$AdminConfig`.

Note: Again, this where operating in "local" mode from a `WSADMIN` command prompt proves useful ... it's quick, interactive, and great for drilling down to find information like this.

This method will take as an argument the "type" you want information on. So in the example shown in the chart, we provide `VirtualHost`. What comes back is a three item list:

- `aliases HostAlias*` -- the attribute is `aliases`, and the asterisk after "HostAlias" is telling you that there are more attributes under this. In other words, "HostAlias" is not itself an attribute, but is comprised of lower-level attributes. We'll drill for those in a bit.
- `mimeTypes MimeEntry*` -- the attribute is `mimeTypes` and again, "MimeEntry" with the asterisk is telling us there's more under this.
- `name String` -- the attribute is `name` and "String" is a key word with *no asterisk*. That means `name` is a low-level attribute and has as its value a simple string of characters.

To drill down under `HostAlias` we once again use the `attributes` method. It turns out that `HostAlias` is itself a "type," and therefore it has its own attributes. Doing this yields two attributes --

`hostname` and `port` -- both with values of `String`. That means `hostname` and `port` are low level attributes and `string` implies the value is a string of characters.

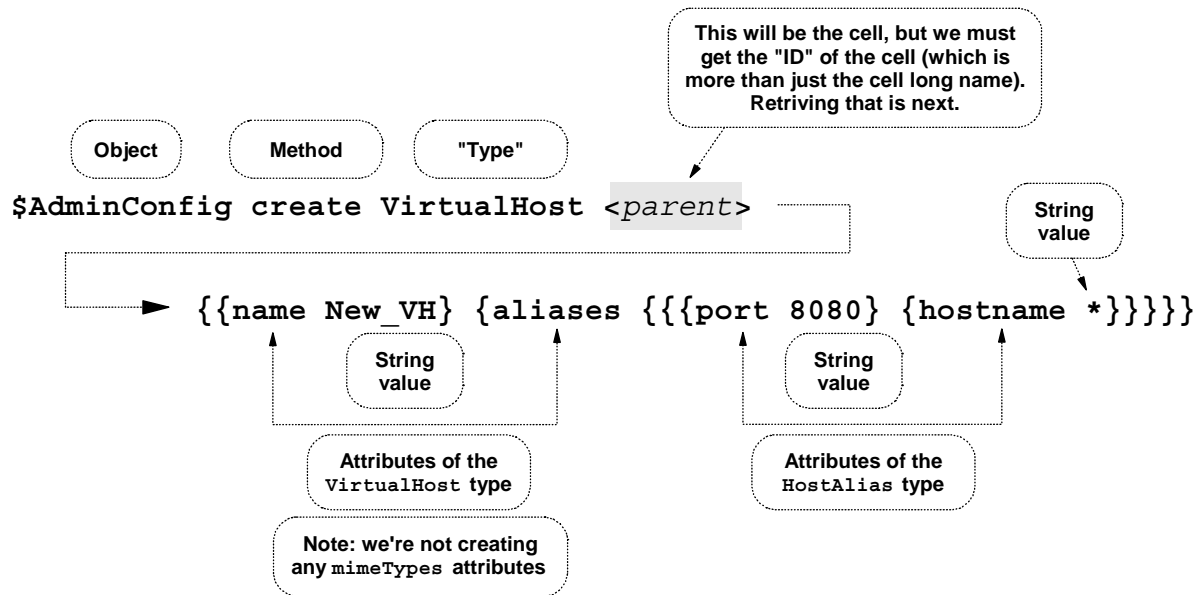
That's a lot of information. What happens if all you want to know is what the absolute minimum is required to create a `VirtualHost` type? You use the `required` method. That yields only those attributes that are required to create the configuration type.

Note: Be careful ... as the chart implies, just because you can create a configuration type doesn't mean that configuration type is then fully usable and functional. A virtual host created with simply a name but no host alias or `mimetype` will simply not work. You can map an application to it, but if you try to drive that application it'll fail.

With this under your belts, let's look at what an actual `WSADMIN` command to create a virtual host would look like.

Creating a New Virtual Host

The following shows the syntax of a `$AdminConfig create VirtualHost` command used to create a new virtual host called `New_VH`. It'll have one port (8080) with a hostname of " * "



Yes, it would be easier to do this through the Admin Console. One-off things like this won't be what you use WSADMIN for. Repeatable things ... yes.

This chart illustrates the command used to create a `VirtualHost` configuration type. What it will do is create a single `VirtualHost` with port 8080 and a hostname of the wildcard value asterisk.

Let's walk through the command:

- `$AdminConfig` is the object
- `create` is the method
- `VirtualHost` is the "type" we're looking to create
- `<parent>` is a special string of characters that uniquely identifies the configuration object to which this new type will belong. It's unfortunately not as simple as providing the cell long name. We'll show the format of this and how to retrieve it next.
- What follows in the braces is the option list. This is where the notion of "nested options" gets important. We saw the attributes of the `VirtualHost` type earlier. We also saw that the `aliases` attribute has itself two attributes. Hence, we're going to have to nest. As we mentioned earlier, the best way to approach this is from the inside working out:
 - The `port` attribute is provided along with its "string" value -- the port number 8080. That is enclosed in braces.
 - The `hostname` attribute is provided along with its "string" value -- a single asterisk. That is enclosed in braces.
 - To "group up" the `port` and `hostname` attributes we enclose those in braces.

- Because the `aliases` attribute might include *multiple* name/port pairs -- though we don't show that here, we have to provide another set of braces to indicate that the starting and ending of all the possible pairs.
- We next enclose the `aliases` attribute in braces.
- The `name` attribute is a peer of `aliases`. It has only one value -- the name of the `VirtualHost` being created. Therefore, `name` and the string value provided are enclosed in single braces, but outside the `aliases` braces.
- Finally, we group the whole thing up with one final set of outer braces.

You're probably looking at that and thinking, "Why in the world would I do that if I'm only looking to create a single virtual host?" That highlights what is an important point: this WSADMIN thing is most powerful for things done repetitively. One-off configuration changes are probably most easily done through the Admin Console.

Using `getid` to Get `<parent>` Value

The `getid` method will return the unique "ID" value for a configuration object. You must supply a "containment path":

`g5cell(cells/g5cell:cell.xml#Cell_1)`

"Containment Path"
of the cell long name

```
set cell_id [$AdminConfig getid /Cell:g5cell/]
$AdminConfig create VirtualHost $cell_id ...
```

```
set cell      "g5cell"
set vh_name  "New_VH"
set host1    "*"
set port1    "8081"
# -----
set cell_id  [$AdminConfig getid /Cell:$cell/]
# -----
set name      [list "name" $vh_name]
set p1       [list port $port1]
set h1       [list hostname $host1]
set pair1    [list $p1 $h1]
set alias_attrs [list $pair1]
set aliases  [list aliases $alias_attrs]
set VH_attrs [list $name $aliases]
# -----
$AdminConfig create VirtualHost $cell_id $VH_attrs
$AdminConfig save
```

Notion of "ID" of configuration object becomes critical when `$AdminConfig` is used to modify an existing object.

Let's now turn to the `$AdminControl` object ...

■

On the previous chart we saw the create method called for something called a `<parent>` -- that's the configuration object under which this new virtual host will be created. What it's looking for a unique string of characters that, for my test cell called `g5cell`, was:

```
g5cell(cells/g5cell:cell.xml#Cell_1)
```

Note: That's actually a reference to an XML tag inside the file `cell.xml`. If you were to look inside that file you'd see a "stanza" of XML labeled with `Cell_1`, and in that stanza is all sorts of information about the cell.

You could simply hand-code that, but there's two problems with that:

- It's awkward. The parent string for the cell is simple compared to other configuration parents.
- You might not have any idea what XML tag number to use. You can probably be pretty assured that `Cell_1` is the reference to the one and only cell, but what about nodes, where multiple nodes might be present? Better to use the `getid` method to extract the information.

The `getid` method will return the unique ID string for a configuration object if you provide what's known as a "containment path" for the object. For the cell long name the containment path is `/Cell:g5cell/`. For a server it would be `/Server:g5sr01c/`.

The best thing to do is to use `getid` to extract the information, place that information into a variable, then use the variable on the command to create the virtualhost (or whatever object you're looking to create). The chart shows the command from the previous chart in Jacl, with variable substitution.

There's quite a bit more to the `$AdminConfig` object. See WP100421 for more example. Let's now turn to the `$AdminControl` object.

The \$AdminControl Object

```
$AdminControl help
```



```
getMBeanInfo_jmx
getNode
getPort
getType
help
invoke_jmx
invoke
isRegistered_jmx
isRegistered
makeObjectName
queryNames_jmx
queryNames
reconnect
setAttribute_jmx
setAttribute
setAttributes_jmx
startServer
stopServer
testConnection
trace
```

The \$AdminControl object is useful only in "Remote" mode where WSADMIN is connected to a server process

If `-conntype NONE` used, \$AdminControl considerably hobbled

Further, WSADMIN must be connected to a server in which the Admin Application is running

Possible to connect to Node Agent or AppServer in ND configuration, but \$AdminControl won't work.

Examples:

```
$AdminControl startServer g5sr01c g5nodec
```

```
$AdminControl stopServer g5sr01c g5nodec
```

A very important \$AdminControl method is `invoke ...` that's used to synchronize to the nodes in a Network Deployment configuration ...

■

The \$AdminControl object is used to control the runtime behavior of the object. For example, you can use it to start or stop servers. But it has some restrictions:

- You can't use it when you use `-conntype NONE`. If WSADMIN isn't connected to a SOAP port, you'll find that most of the \$AdminControl object is restricted.
- WSADMIN must be connected to the SOAP port of the server process in which the Admin Application is running. Here's where we get into the issue of connecting to the DMGR port versus a Node Agent's port versus a server's port. All are possible, but only the DMGR SOAP port will yield the ability to control stuff through \$AdminControl.

Perhaps the most useful aspect of \$AdminControl -- or at least the one you'll likely use the most -- is the ability to synchronize nodes in a ND configuration. If, for example, you install an application using \$AdminApp and you save it with \$AdminConfig save, the application will be installed only in the "master configuration." If you want to use the application in an application server, you'll need to "synch" to the node in which that application server belongs. We'll show that next.

Using `invoke` Method to Sync Nodes

Updates made to the "master configuration" are not usable until they are "synchronized" to the nodes. This is done with the `invoke` method:

Synchronizing with a single, specific node

```
WebSphere:platform=common,cell=g5cell,version=5.0,name=nodeSync,
mbeanIdentifier=nodeSync,type=NodeSync,node=g5nodec,process=nodeagent
```

```
set var [$AdminControl completeObjectName type=NodeSync,node=g5nodec,*]
$AdminControl invoke $var sync
```

Synchronizing with multiple nodes

All nodes
in a cell:

```
set node_ids [$AdminConfig list Node]
foreach node $node_ids {
  set node_name [$AdminConfig showAttribute $node name]
  set nodeSync [$AdminControl completeObjectName type=NodeSync,node=$node_name,*]
  if { !($nodeSync=="") } then {
    $AdminControl invoke $nodeSync sync
  }
}
```

"If" structure checks to make sure node is not DMGR node. If not, then synchronize.

All nodes
across which
cluster is
defined:

```
set c_id [$AdminConfig getid /ServerCluster:g5sr02cluster/]
set c_membs [$AdminConfig list ClusterMember $c_id]
foreach m_id $c_membs {
  set node_name [$AdminConfig showAttribute $m_id nodeName]
  set nodeSync [$AdminControl completeObjectName type=NodeSync,node=$node_name,*]
  set work [$AdminControl invoke $nodeSync sync]
}
```

■

This gets really complicated really quickly. To synchronize a node you must first extract from WebSphere the "completeObjectName" of the mBean of the type `NodeSync`. How do you pull that information out of WebSphere? You provide a pointer to the node in which you want to synchronize. So in this case we simply pointed to the node using its long name. The result is a very long string that's best handled in a variable.

??? If none of that made any sense, just take it on faith. Honestly, at some point in working with WSADMIN you'll have to simply accept the examples without fully understanding what's going on. I know I have on more occasions than I care to admit. ☺

To actually synchronize with the node, you use the `invoke` method of `$AdminControl`, and you drive against the "completeObjectName" you captured in the variable, and you pass in the keyword `sync`. Provided that the Node Agent is up and running (a critical piece of this), the synchronization will take place.

Here's the thing you have to understand regarding node synchronization: there's no way to synch to all the nodes with a single command. You can't do that in WSADMIN, and the Admin Console doesn't do it with one command. The way it's done is to programmatically synch to each node, one after another. For this we use the `list` method of `$AdminConfig` to list out the nodes, then we cycle through them. There's two flavors of this:

- Synchronizing to all the nodes in a cell. The example of that is shown above. The "if" logic is testing to see if the node being worked on is the Deployment Manager node. We don't want to try to synch to it because that operation will yield an error. So we test to see if the

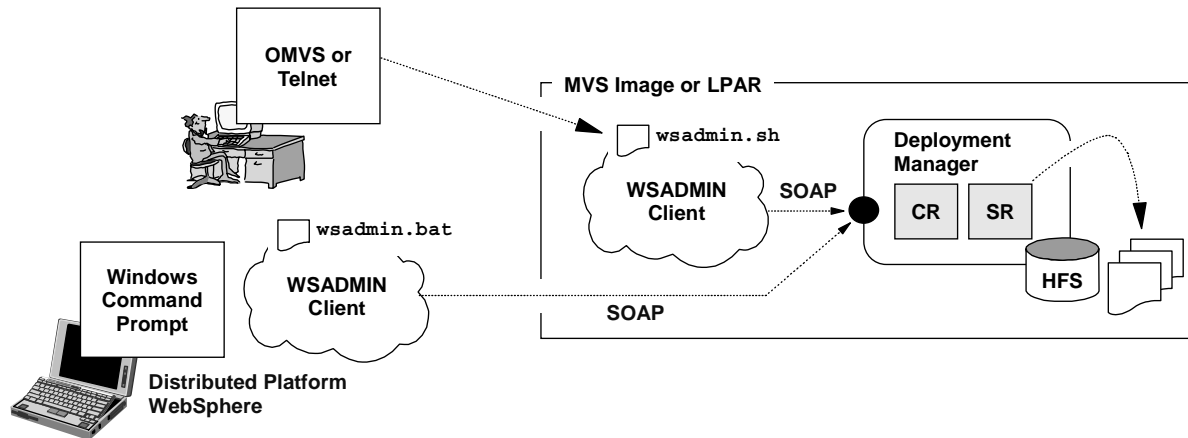
"completeObjectName" value is *not null*, and if not null then we synchronise. (The Deployment Manager's node will yield a null "completeObjectName" value.)

- Synchronising to only those nodes in a given cluster. Here we use a few methods to extract a list of nodes represented in the cluster, then cycle through the list. We don't need to worry about the Deployment Manager here because we know that a cluster will not have one of its members in the Deployment Manager node.

Let this information bubble around in your mind a bit. Cut-and-paste this stuff into your Jacl script and simply make use of it without knowing exactly how it all works. Later you'll start figuring out how to use the more powerful aspects of this.

If Global Security Enabled

Affects how you invoke WSADMIN in "remote" mode. ("Local" mode is unaffected by global security because it doesn't go through server.)



Two things:

- **Pass `-user` and `-password` in on invocation of WSADMIN:**

```
./wsadmin.sh -conntype SOAP ... -port 15510 -user g5admin -password #####
```

- **Insure ID under which WSADMIN runs has proper CA Certificate in keyring**
Must have CA certificate used to sign default certificate of the DMGR controller ID's keyring

■

The enablement of "Global Security" is like a big red switch on the wall: turn that switch on and all sorts of things change. It's no different with WSADMIN.

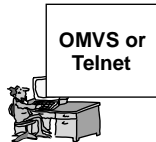
Note: If you're running WSADMIN in "local" mode you don't have to worry about this. Global Security will only affect the ability to access and be accepted into the SOAP port. Local mode operations involves no server process, so it doesn't matter if security is turn on or not. Besides, it's possible to use WSADMIN in local mode when all the servers are stopped, thus security on or off wouldn't affect WSADMIN.

What does change is the manner in which you invoke WSADMIN in remote mode. There are two things you must do:

1. You must pass in a userid and password so the server process owning the SOAP port can authenticate you. There are two ways to do this:
 - By passing `-user` and `-password` on the invocation command, as shown in the chart
 - By coding user and password in the `soap.client.props` file.
2. You must insure that the id under which WSADMIN is running has the "certificate" for the "Certificate Authority" that signed the default certificate used by the server. This is necessary so the WSADMIN client can properly determine that the certificate it receives from the server when the SSL handshake is taking place is valid.

On the next chart we'll focus on the userid and password issue, and then we'll turn our attention to the keyring/certificate issue.

User/Password Passed In



```
./wsadmin.sh -conntype SOAP -host wsc3.washington.ibm.com
-port 15510 -user G5ADMIN -password XXXXXXXX -f /u/user1/test.jacl
```

Couple of points:

- The userid and password you send in needs to have **READ** access to the **EJBROLE** profile defined for the WebSphere cell
 - Does not *have* to be the "WAS Admin ID," but that will by default have access
- You can hard-code this into the `soap.client.props` file and avoid having to send it in on each command line:

```
# JMX SOAP connector identity
com.ibm.SOAP.loginUserId=G5ADMIN
com.ibm.SOAP.loginPassword=XXXXXX
```

**No user
passed in**

```
BBOO0222I SECJ0305I: Role based authorization check failed for 506
security name <null>, accessId NO_CRED_NO_ACCESS_ID while invoking
method getProcessType on resource Server and module Server.
```

**User passed
in, not in
EJBROLE**

```
BBOO0222I SECJ0305I: Role based authorization check failed for 507
security name <plex/ID>, accessId user:<plex/ID> while
invoking method getRepositoryEpoch on resource ConfigRepository and
module ConfigRepository.
```

■

What userid do you provide to authenticate you to the SOAP port? The "WebSphere Admin ID" will by default have the proper authorities, but it's not required that you use that ID. What is required is that the ID/password sent in be a valid RACF ID, and have **READ** access to the **EJBROLE** profile defined for the WebSphere cell. There are four **EJBROLE** profiles created by default when you run the "Security Domain" customized jobs: `administrator`, `monitor`, `configurator` and `operator`. By default the "WebSphere Admin ID" will have access to all four.

But your personal userid may not have **READ** access to any of them. So to use your personal ID with **WSADMIN**, you'd need to have someone grant you **READ** access to one of the profiles ("administrator" gives you full authority).

Things get a little more complicated when the issue "Security Domains" come into the picture. What a "security domain" provides is a way to qualify the **EJBROLE** profiles with a string of characters that is applicable to only one cell. So the `administrator` profile might be qualified with `G5CELL.administrator`, which means that only IDs with **READ** access to that could get in. You must know what the profile structure for your cell looks like if you want to grant an ID **READ** to the profile.

You don't have to provide the user and password on the invocation of the command ... you can code those values into the `soap.client.props` file as shown. It results in the same thing.

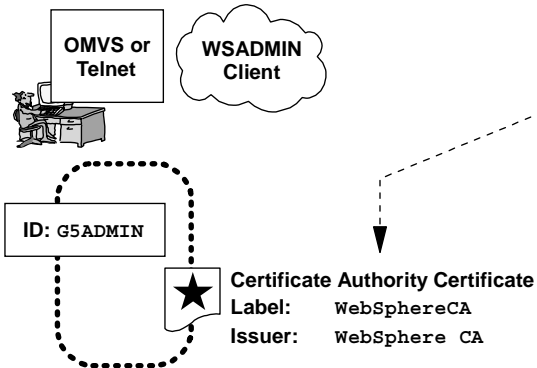
What are the error symptoms you can expect? Two cases are represented: when no userid/password is provide (for example, you simply forgot to provide it), and when the userid passed in does not have **READ** to a **EJBROLE** profile. The errors you'll see on the MVS console (if RACF auditing is on) are shown on the chart.

WSADMIN and CA Certificates

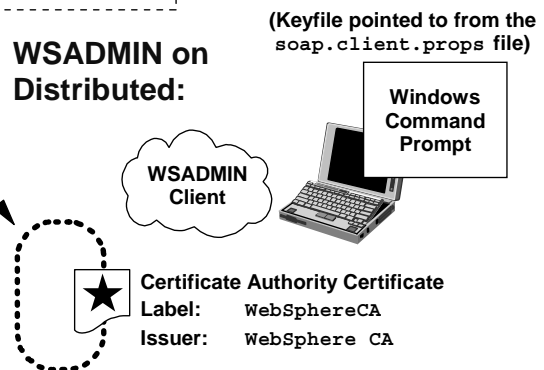
Nutshell:

The Certificate for the CA who signed the DMGR's default certificate must be present in the keyrings of the client:

WSADMIN on z/OS:



WSADMIN on Distributed:



Error symptom:

```
WASX7023E: Error creating "SOAP" connection to host "<host>"; exception information:
com.ibm.websphere.management.exception.ConnectorNotAvailableException
```

Whenever a WSADMIN client connects to the SOAP port when Global Security is enabled, an SSL connection will be established. In order to establish the SSL connection, the server is going to pass a "certificate" to the client, and the client will use that certificate to set up the encrypted link. But before the client can safely use the certificate passed it by the server, it needs to make sure the server is who it says it is. This is done programmatically by comparing the "signature" on the server's certificate -- put there by a "Certificate Authority" (CA) -- with a copy of that CA's certificate. If the signature on the server's certificate properly compares to the client's copy of the CA's certificate, then the client can be reasonably assured that the server's certificate is okay.

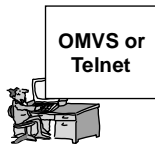
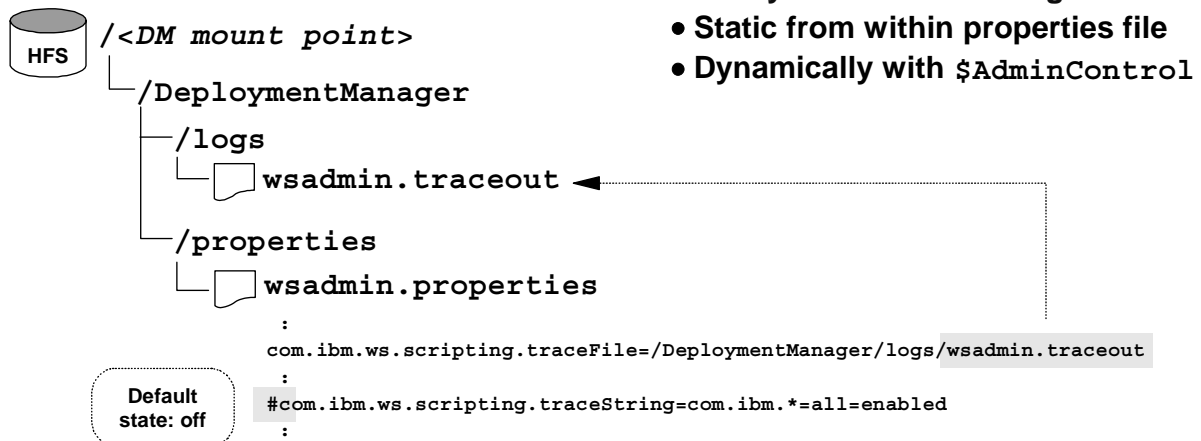
In order for all this to work, the client needs to have the CA's certificate in its "keyring." If the CA's certificate isn't in the client's keyring, the client won't be able to validate the signature on the server's certificate. You'll get the error shown at the bottom of the chart.

There are two scenarios we need to talk about:

- WSADMIN client is on z/OS -- here you're likely to already have the CA's certificate in the client's keyring because it would have been put there by default when customizing the environment. But that's only the case if you're running WSADMIN under the same "WebSphere Admin ID" as the server to which you're connecting. If you're running WSADMIN on a completely different z/OS system you may not have the CA's certificate in your client's keyring.
- WSADMIN client is on distributed platform -- here you're unlikely to have the CA's certificate in the client's keyring by default. But it is possible to put it there. It involves exporting the CA's certificate from the RACF database and then bringing it down to the client's system and importing it into the keyring for the client.

The important point here is the concept: having the CA's certificate in the client's keyring. The steps needed to put the CA Certificate there is beyond this presentation's scope.

Tracing of the WSADMN Activities



```

$AdminControl trace com.ibm.*=all=enabled
$AdminControl trace com.ibm.*=all=disabled
  
```

■

There'll come a point when you have to turn on tracing to debug a problem. There are two ways to do this:

- You can un-comment the `traceString` line in the `wsadmin.properties` file to enable tracing. Tracing will then go to the file specified on `traceFile`.

Note: Even with `traceString` commented out, some -- but not much -- tracing will still occur.

- You can dynamically turn it on or off with the `$AdminControl trace` function shown.

How to read and make good use of the trace files is a subject well beyond the scope of this presentation. But if anyone asks you to trace a problem, here's how you'd do it.

End of Document