Objective of this Presentation

Provide an understanding of the application types supported by Liberty

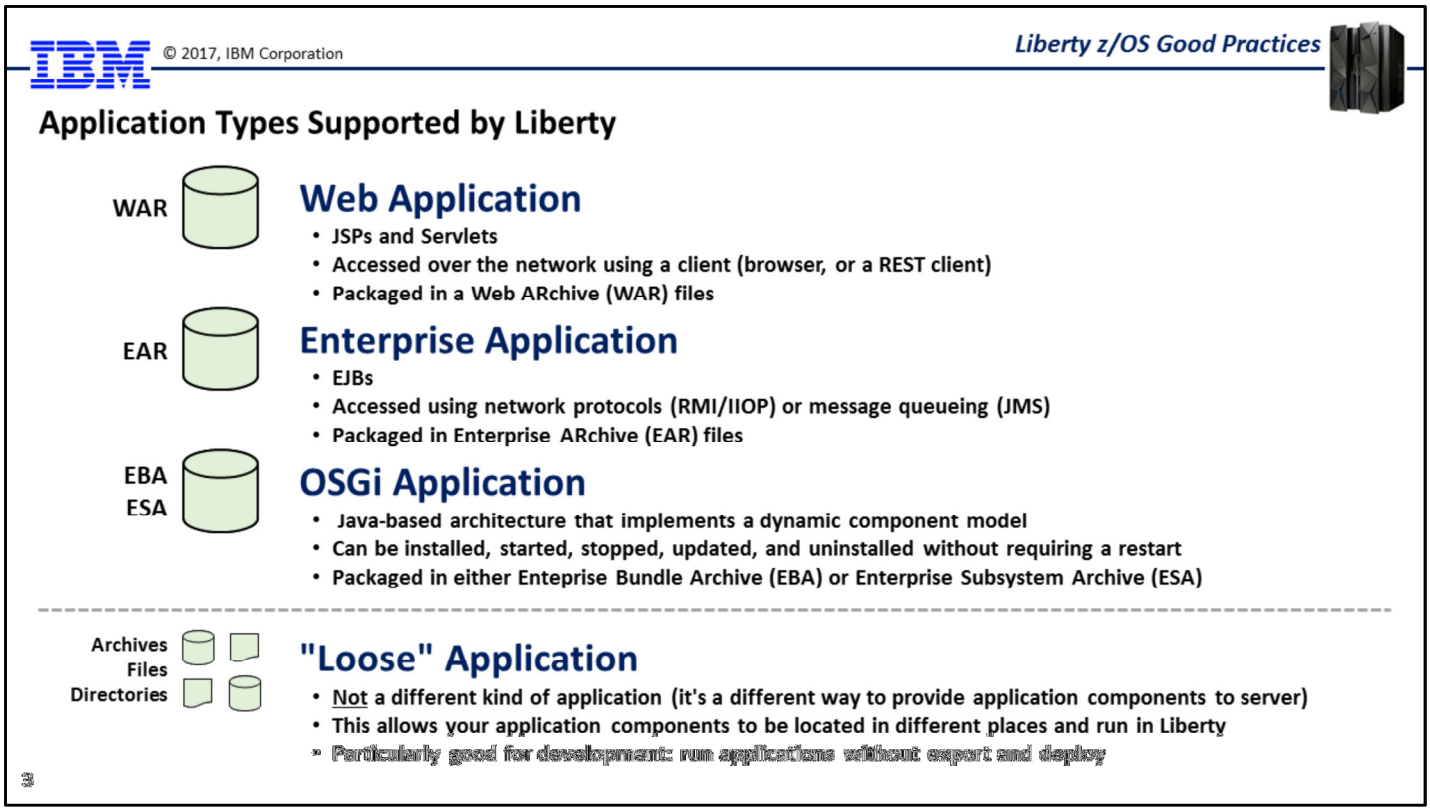
Provide a general understanding of the API model of Liberty, particularly as it relates to the API model of WAS traditional

Provide an understanding of the deployment model of Liberty

2

For this unit we will explore the topic of applications and application deployment within Liberty. Even though our focus is z/OS, this material relates to Liberty on all platforms, as the programming APIs are the same across all platforms. The deployment model is also the same across all platforms, with the possible exception of the ability to share a file system between LPARs in a Sysplex environment. (You could achieve something similar with distributed systems as well. The point here is that in a Sysplex environment the sharing facility may be present, as it commonly is, where on a distributed system it may not be.)

This unit does not go into programming specifics or examples, as that is outside the scope of what we're trying to accomplish with this collateral.



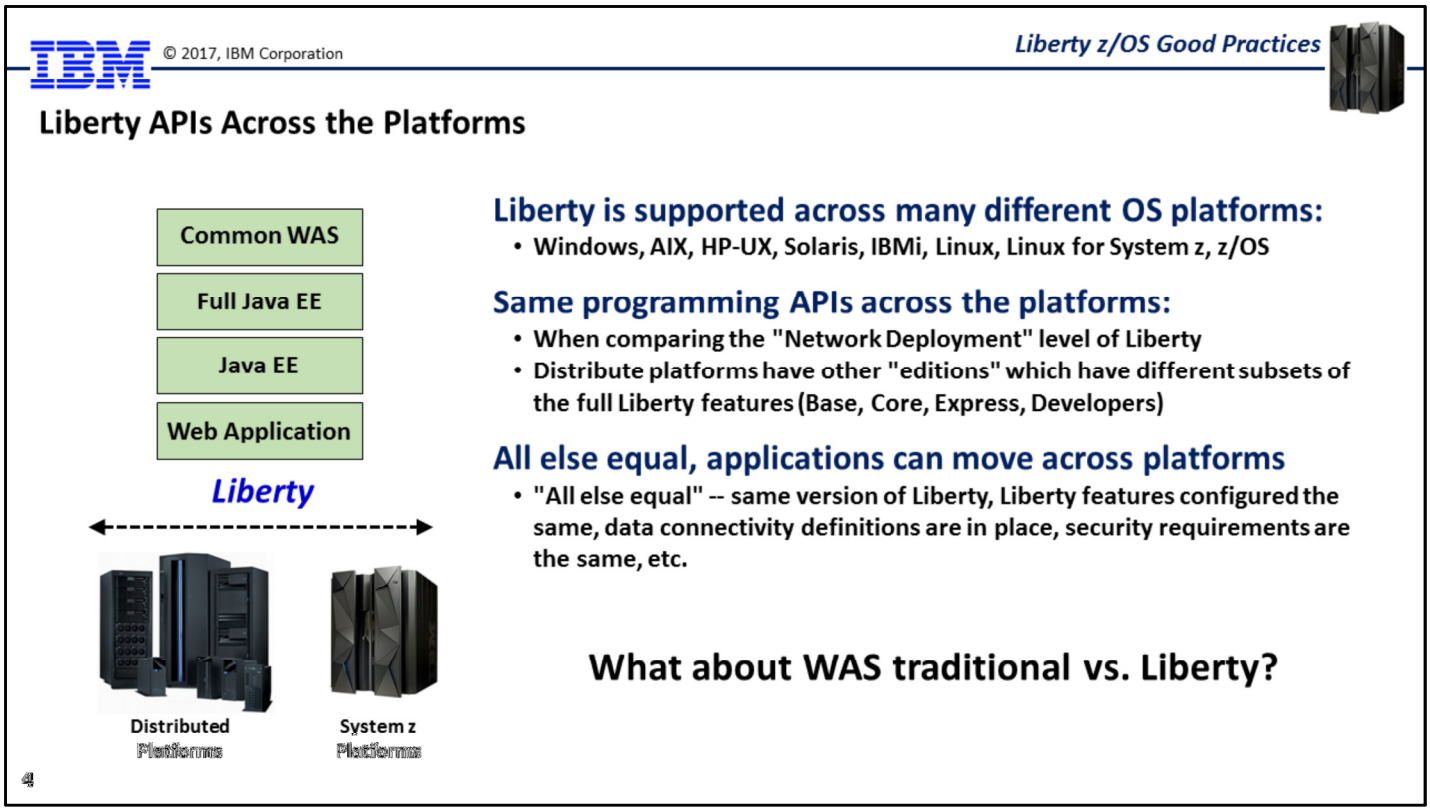
This chart is intended to provide a brief survey of the types of applications supported by Liberty, and the packaging models for each.

- **Web application** -- this includes JSPs and servlets, and is accessed over a network, typically with HTTP protocol. The packaging as a "WAR" file (Web ARchive). This type of application was supported by WAS traditional as well.
- **Enterprise Application** -- this includes EJBs, and is accessed with protocols such as RMI/IIOP, or message queuing. The packaging is an "EAR" file (Enterprise ARchive). This type of application was supported by WAS traditional as well.
- **OSGi Application** -- the acronym stands for "Open Service Gateway Initiative," and is based on the idea of a dynamic component model. Liberty itself is based on an OSGi design. OSGi applications are packaged in either EBA (Enterprise Bundle Archive) or ESA (Enterprise Service Archive) file formats.

  Note: as of the time of writing this, WAS traditional supported a few more OSGi APIs than does Liberty, but that gap is closing with each quarterly fixpack.

- **"Loose" Application** -- this is not really a different type of application, rather it is a different mechanism by which to deploy applications to Liberty. This is based on the idea of deploying the artifacts of an application without packaging into an archive file. They can be located at any location, and the location is specified in the server.xml. This is a good format for development environments were a quick change to some file can be done without doing a full repackage / export / deploy. This is probably not the mechanism to be used for production environments, where change control will be focused on the archive file tested and validated prior to deployment to production.

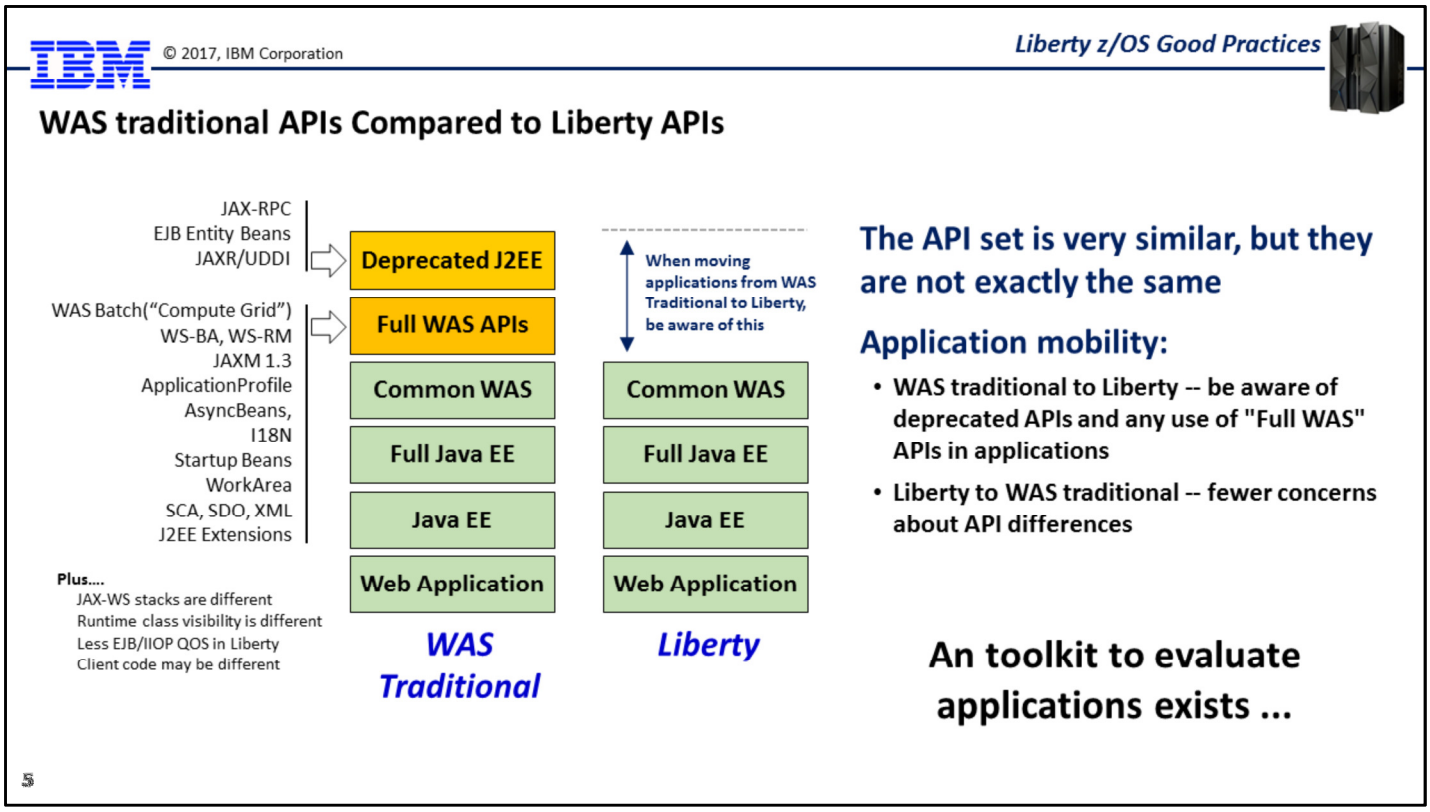
Your Liberty environment may run some or all of these application types.



A central design feature of Liberty is that the programming APIs are common across all platforms when comparing like-to-like releases ("editions") of Liberty. The only edition of Liberty supported on z/OS is what's called "network deployment." If you compare that edition on z/OS to that same edition on Windows, AIX, HP-UX, Solaris, IBMi, or Linux, you'll have the same API set (assuming the version and fixpack level is the same).

The value of that is it allows you to develop on one platform and deploy on another without having to worry about programming API incompatibilities. Now, you have to make sure the application has what it needs on the target platform for this to hold true. For example, you can't take an application that runs in Liberty on Windows and move it to a Liberty z/OS server that does not have the same configuration definitions for things like JDBC. Or if the application relied on some security artifact defined to Linux but not present on the target z/OS system. But it holds true that if we go from like-to-like, then the identical programming model provides portability.

That's Liberty-to-Liberty application portability; what about WAS traditional to Liberty?

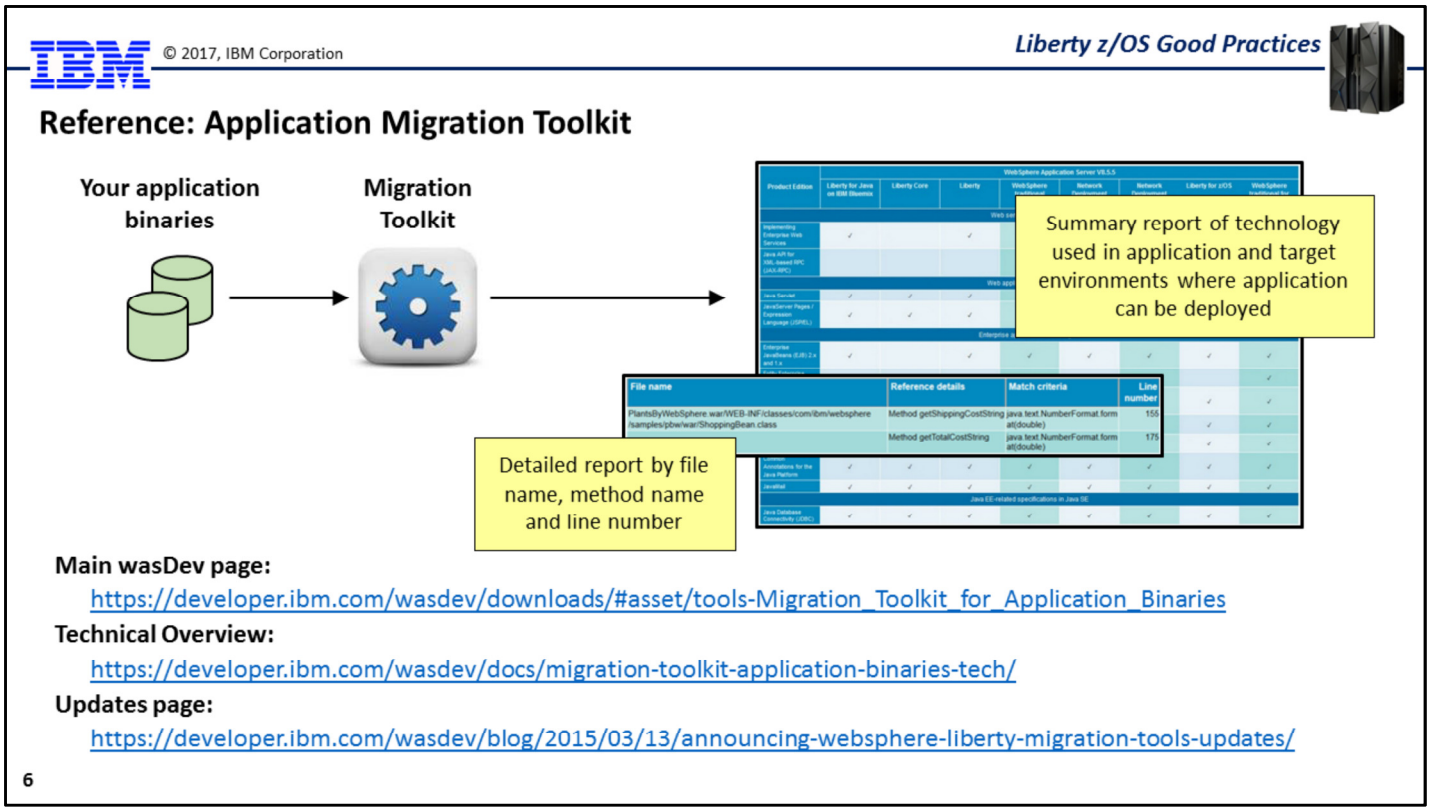


When comparing WAS traditional with Liberty (any platform), we see there is an "API gap" with Liberty. That "gap" is comprised of two categories of APIs:

- **Deprecated J2EE** -- these are APIs that are part of the open standard but are marked deprecated. They have been superseded by more current standard specified APIs. Applications use of deprecated APIs is not a good practice, and applications that do make use of those APIs should be investigated to bring up to current API standards. These applications may still work in WAS traditional (because the APIs are deprecated-but-not-yet-removed). But on Liberty the applications will throw an error since the API they are seeking to use is no longer present.
- **"Full WAS" APIs** -- these are APIs IBM supplied over and above the common open standard APIs. There is a set of those "above and beyond" APIs that are not in Liberty. The chart shows some examples of this. An application written to make use of any of those APIs would not work if carried to Liberty.

Note: there is an excellent write-up on this topic:
https://developer.ibm.com/wasdev/docs/was-classic-or-was-liberty-how-to-choose/

This begs the question: is there a way to evaluate applications to see what issues may exist? The answer to that is yes. We point you to that on the next chart.

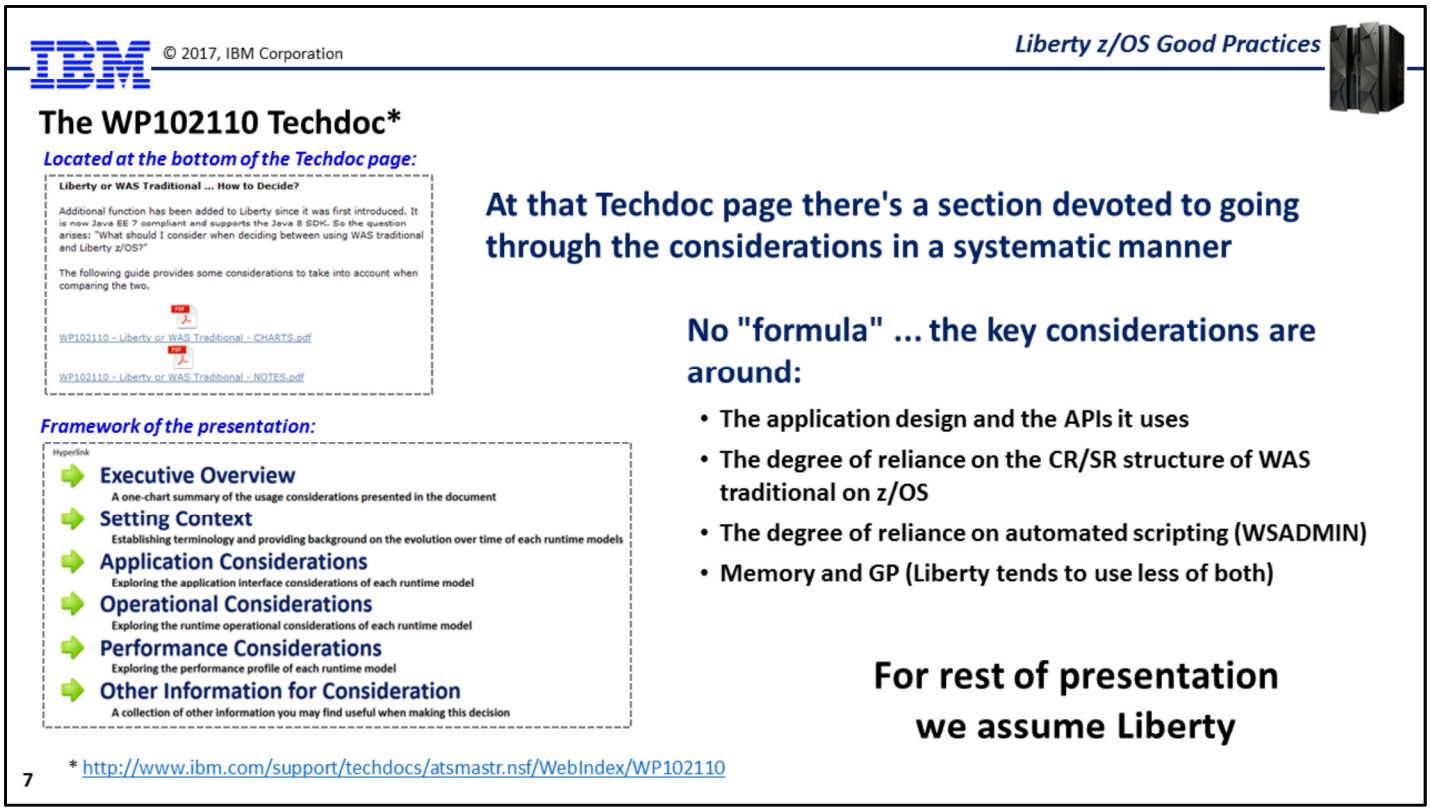


IBM has produced an application migration toolkit to assist you with analyzing applications and assessing the portability to the target platform you specify. This toolkit works on program source as well as program binaries The chart provides the URLs, which are reproduced here in case you can't use the hyperlink from a chart format of this deck:

https://developer.ibm.com/wasdev/downloads/#asset/tools-Migration_Toolkit_for_Application_Binaries

https://developer.ibm.com/wasdev/docs/migration-toolkit-application-binaries-tech/

https://developer.ibm.com/wasdev/blog/2015/03/13/announcing-websphere-liberty-migration-tools-updates/



To further assist with the broader question of which runtime to use -- WAS traditional z/OS or Liberty z/OS -- we have produced a Techdoc that takes you through a structured discussion of the considerations. The Techdoc comes in both chart format and speaker-note format. It can be found here:

http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102110

For the rest of this unit, we will assume the decision has been made to consider Liberty z/OS, and we will focus on that.

Liberty Application Development Good Practices

Liberty is a Java EE application server, so there's nothing unique about it compared to other Java EE runtimes

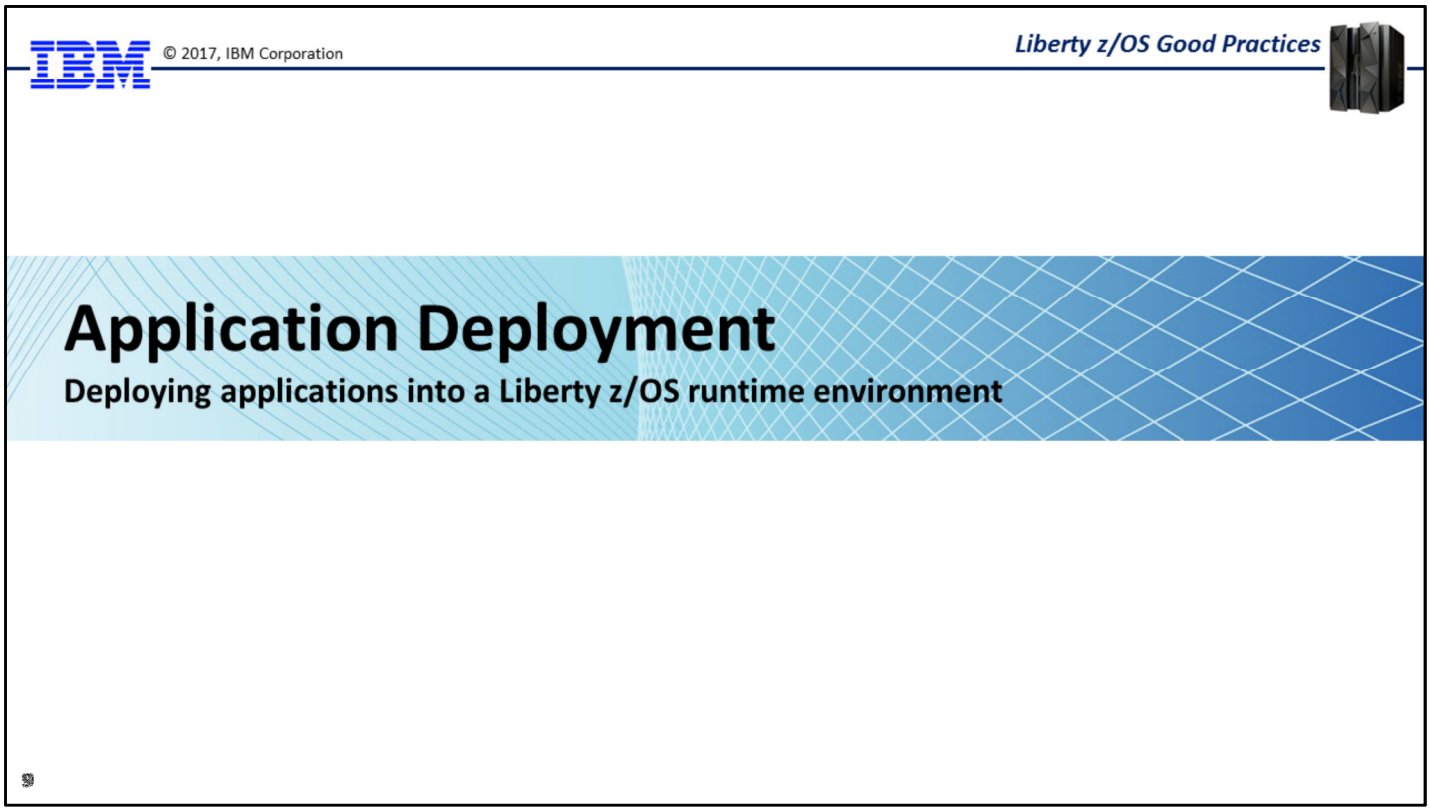
General good practices Java development ...

- Use good design practices
- Maintain good source control
- For best performance, make heavily used code as efficient as possible
- Profile the application prior to deployment into production
- Use proven change control processes

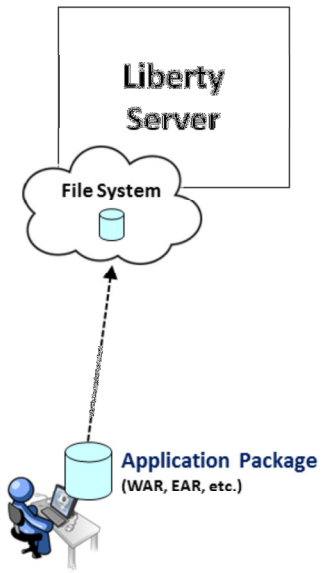
8

This deck is not intended to focus on Java programming good practices. That topic is fairly well documented elsewhere, and with a platform agnostic language like Java part of the discussion, the good practices are even more broadly-based.

It seems almost cliché to say it, but it's true: if you want a good Java program, then design the Java program to be good. There is a well know set of good coding practices in general, and Java specifically, and it starts with the program objectives and program design. Then: practice good source control, test for performance and where heavily used code is involved, make that efficient as can be accomplished. Profile the application and find out where the hot spots are, and correct areas that are hampering performance and throughput. Finally, exercise good and proven change control processes. A million dollar infrastructure can be made useless by a bad change introduced at the last minute without proper testing and control.



The Liberty Deployment Model



Two choices:

1. Drop file in /dropins directory and let Liberty detect and dynamically load
2. Place file in another location and configure application into server.xml

Use whatever mechanism you wish to get application package file from your development / source control environment to a file system accessible by the server

Then the question is: Dynamic update? Or rely on server restart?

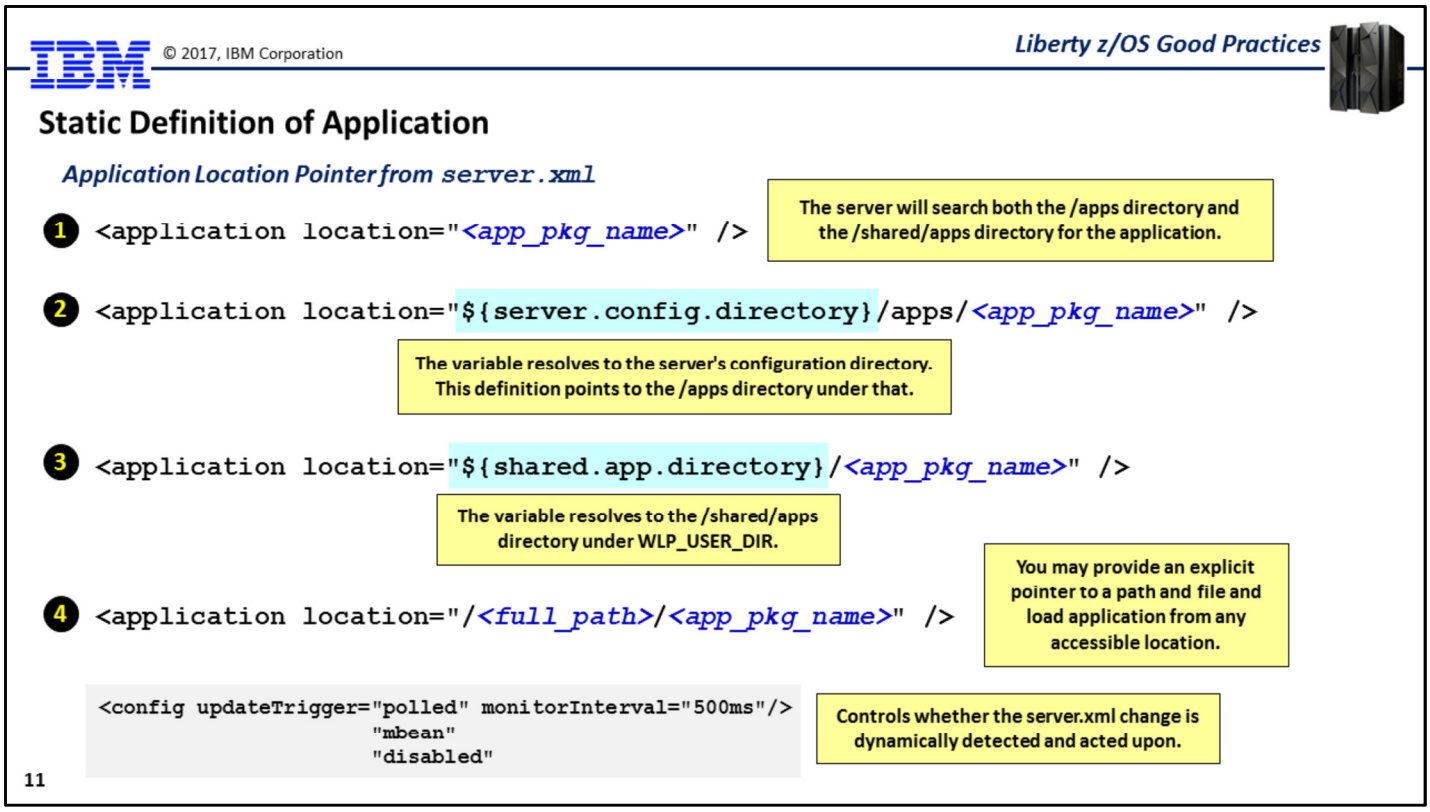
- In general, we see production environments *avoiding* dynamic updates.
- If dynamic updates are disabled, then application are loaded at server restart
- If dynamic updates, the trigger mechanism (polled, mBean) can be configured

10

The deployment model for Liberty is quite a bit different than the deployment mode for WAS traditional. WAS traditional required the application be deployed using the administrative interface: either the GUI Admin Console, or the WSADMIN scripting interface. That's because application deployment in WAS traditional involved unzipping the application archive file and updating a number of XML files to let the runtime know about the application.

Liberty is a simpler model -- either (a) have dynamic updates enabled and place the application file in the /dropins directory, or (b) place the application file in a some directory and point to it with an XML element in server.xml. You can use pretty much any deployment tool to accomplish this.

The next question is whether you wish to rely on the dynamic update capability of Liberty, or whether you wish to control things by mandating a server stop and restart to pick up changes. Let's explore those options over the next few charts.



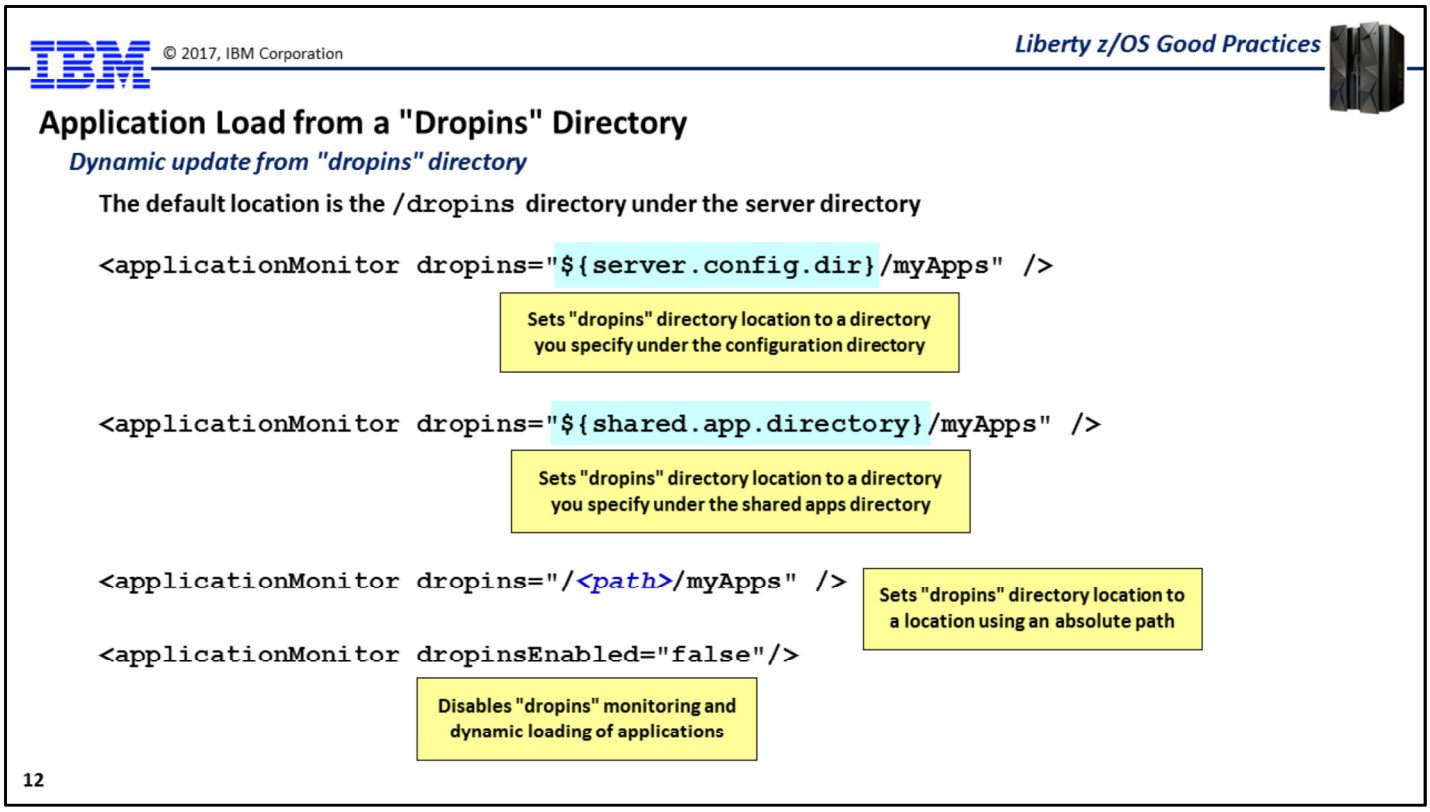
Let's first focus on the static definition of a file. This is done in the server.xml file, or a file that is merged into the server.xml file with an <include> statement.

Note: "static" in this case does *not* rule out the definition being dynamically loaded. It simply means the location and name of the application file is specified in the server configuration. The *configuration* change can still be dynamically loaded, and thus the *application* dynamically loaded based on the new definition in the configuration. The XML shown at the bottom of the chart indicates what controls whether configuration dynamic updates take place, and if so, how often.

You have several options with respect to how you define where the application resides:

1. The server will look in either the /apps directory under the server directory, or it will look in the /shared/apps directory under the WLP_USER_DIR location. If the named application is found in either place, it will be loaded.
2. This uses the ${server.config.directory} variable to resolve the location to the server's directory. The /apps string after that tells Liberty to look in that directory and load the application. This is useful when you are bringing in the application definition using <include> processing. Two servers (in a cluster, for example) could merge in this element, and each would resolve the path to *its* server configuration directory.
3. This uses the ${shared.app.directory} variable to resolve the location to /shared/apps under the WLP_USER_DIR. Here again, this would be useful when pulling in common XML where the application to be loaded by different servers was found at that location. A cluster where the two servers are under the same WLP_USER_DIR could use this pull in the application from a single location.
4. This is a full path pointer to the application location. This can be anywhere; the only requirement is the server ID have read to the location.

The gray box at the bottom provides an illustration of the XML that will control whether the configuration changes will be picked up dynamically. For example, <config updateTrigger="disabled" /> means there is no dynamic update, and the application definition will be picked up at next server start. That may be what you want for production. Or you can specify <config updateTrigger="polled" monitorInterval="60s" /> to indicate every 60 seconds the server will poll for changes and load the changes.



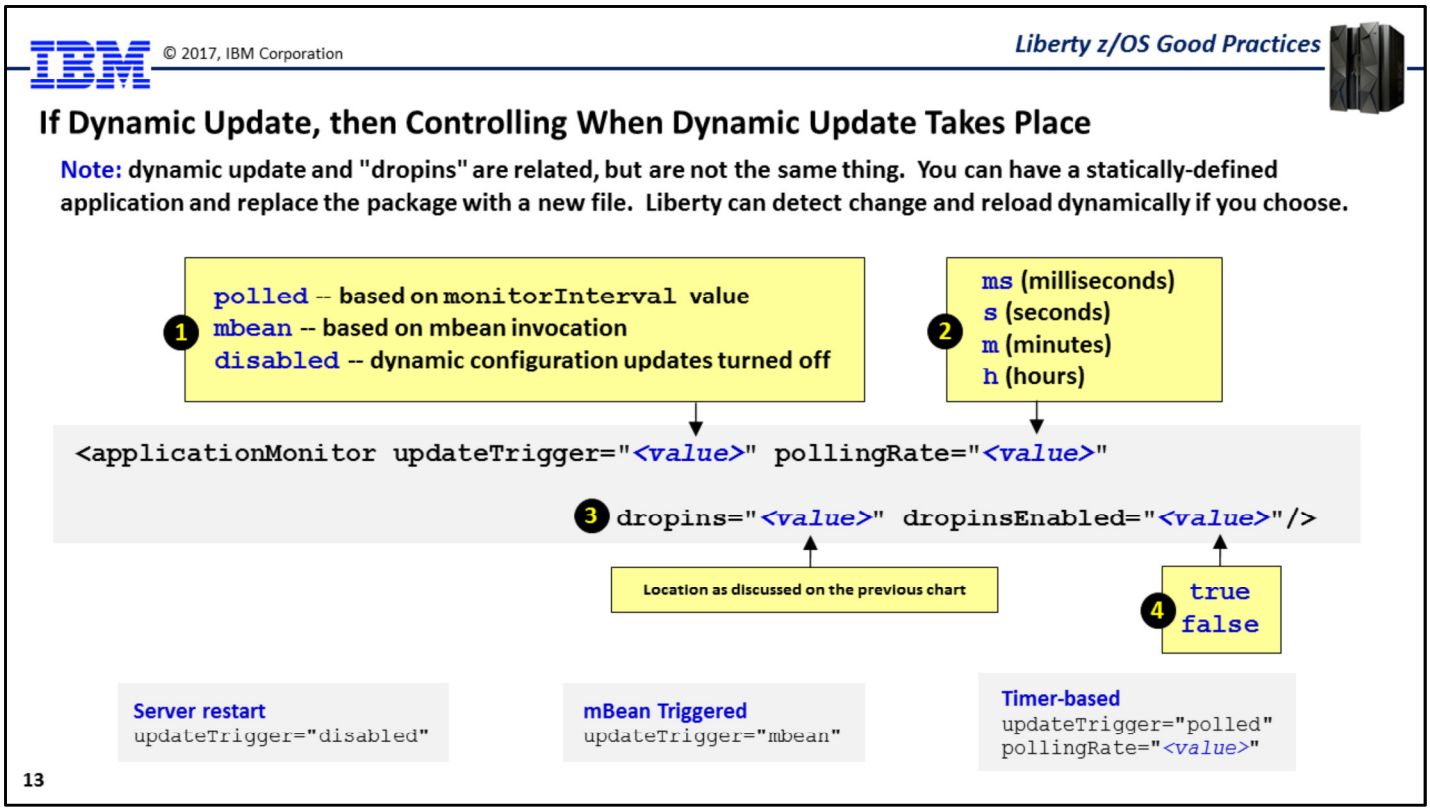
You can avoid statically defining applications in your server and take advantage of the ability of Liberty to monitor a directory for application files and load them when seen. By default that will be the /dropins directory under the server's directory, and you can use that if you'd like. But if you'd like to have the "dropins" directory somewhere else you can do that. The <applicationMonitor dropins=" " /> value is what defines the location.

The first two examples are showing the use of some built-in variables. The ${server.config.dir} will resolve to that server's configuration directory (which is where server.xml resides). The ${shared.app.directory} will resolve to the /shared/apps location under the WLP_USER_DIR location.

Note: those are not the only two built-in variables. There are many more.

The third example shows an explicit path to a location. This can be anywhere. The only requirement is the server ID must have read access.

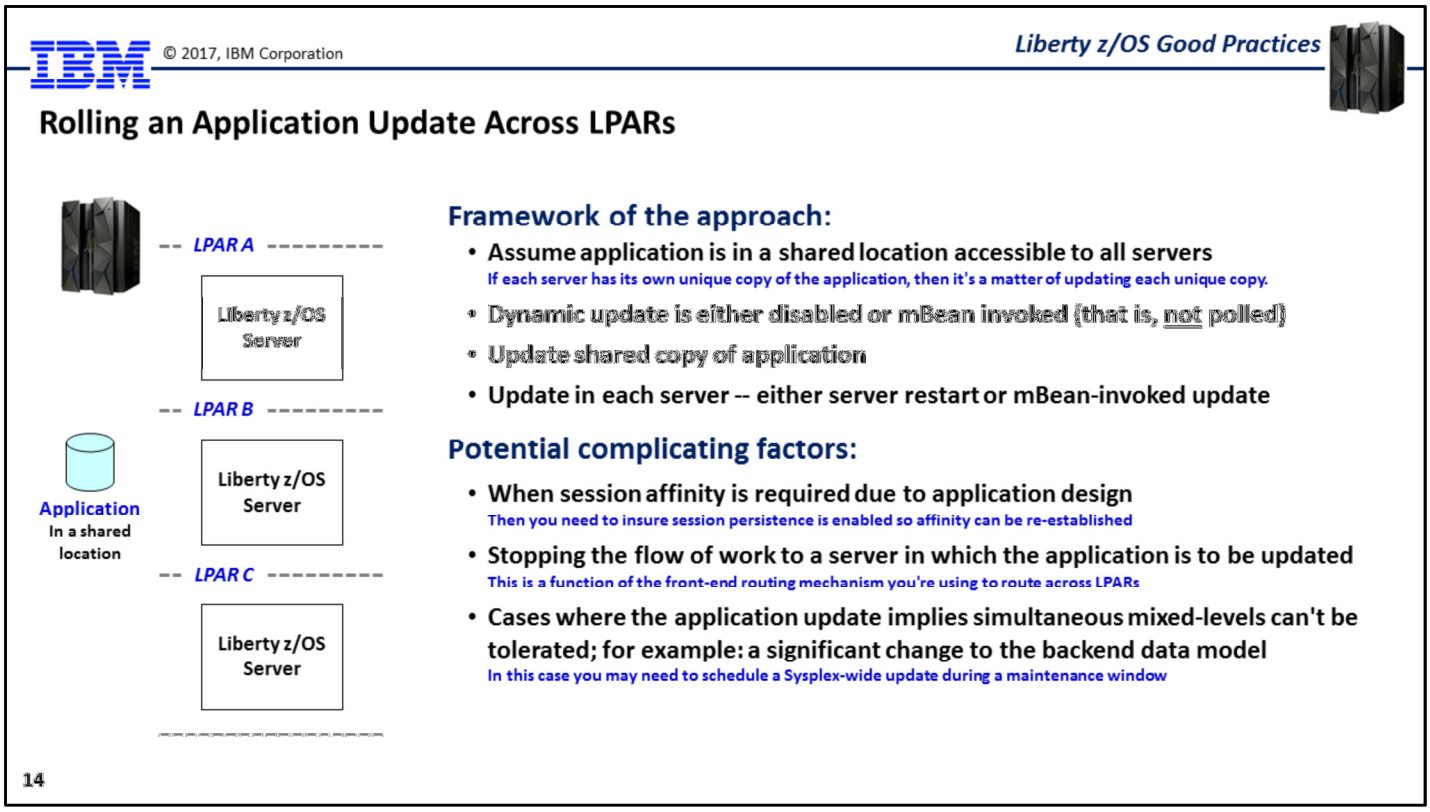
Finally, the last example on the chart illustrates how to turn off dynamic loading from whatever "dropins" directory you specify. With dropinsEnabled="false" set, the server will not look for or load applications from either the default dropins directory a defined one. It won't load from there even on a server restart. So dropinsEnabled="false" is how you can force the use of statically-defined applications.



Here is the <applicationMonitor> element one more time. Let's walk through this and review the different values and what they do:

1. **updateTrigger =** this determines *how* updates will be accomplished. The default is "polled."
   Note: this applies to applications deployed with a static definition or deployed via a "dropins" directory.
   - If "polled," and "dropinsEnabled" is either unspecified or set to true, then the "pollingRate" value is used to determine how often the deployed are checked for updates.
   - If "mbean," then it will check for changes to the deployed applications only when the management bean is invoked. This provides a way to perform application updates without polling and without a server restart.
   - If "disabled," then the applications are not checked; a server restart is required to pick up changes.
2. **pollingRate=** if "updateTrigger=polled," then this determines how often the polling takes place. Your options are expressed in milliseconds, seconds, minutes, or hours. The default is 500ms, or one-half second.
3. **dropins=** this defines the directory path location where the server will look for applications to deploy. This defaults to the /dropins directory. This is ignored if "dropinsEnabled" is set to "false." This is how you would control the ability of others to deploy an application simply by copying in a file.
4. **dropinsEnabled=** this determines if processing of a "dropins" directory is done at all. If "true" (which is the default), then either the default location is checked (/dropins) or the value seen on "dropins=". If set to "false," the server does no processing of applications out of any dropin directory; that means application deployment must be accomplished with a static definition of the application path and file name.

We anticipate for tightly controlled production environments applications will be statically defined and loaded with a server restart. In that case code "updateTrigger=disabled" and "dropinsEnabled=false".



The question often comes up how one would achieve the "rolling" of an application through a set of like-configured servers (which could be a Liberty collective "cluster" but it does not have to be). The purpose of "rolling" an application is to maintain at least one instance of the application up and servicing clients while other instances of the application are refreshed (which implies a brief time when it's not available).

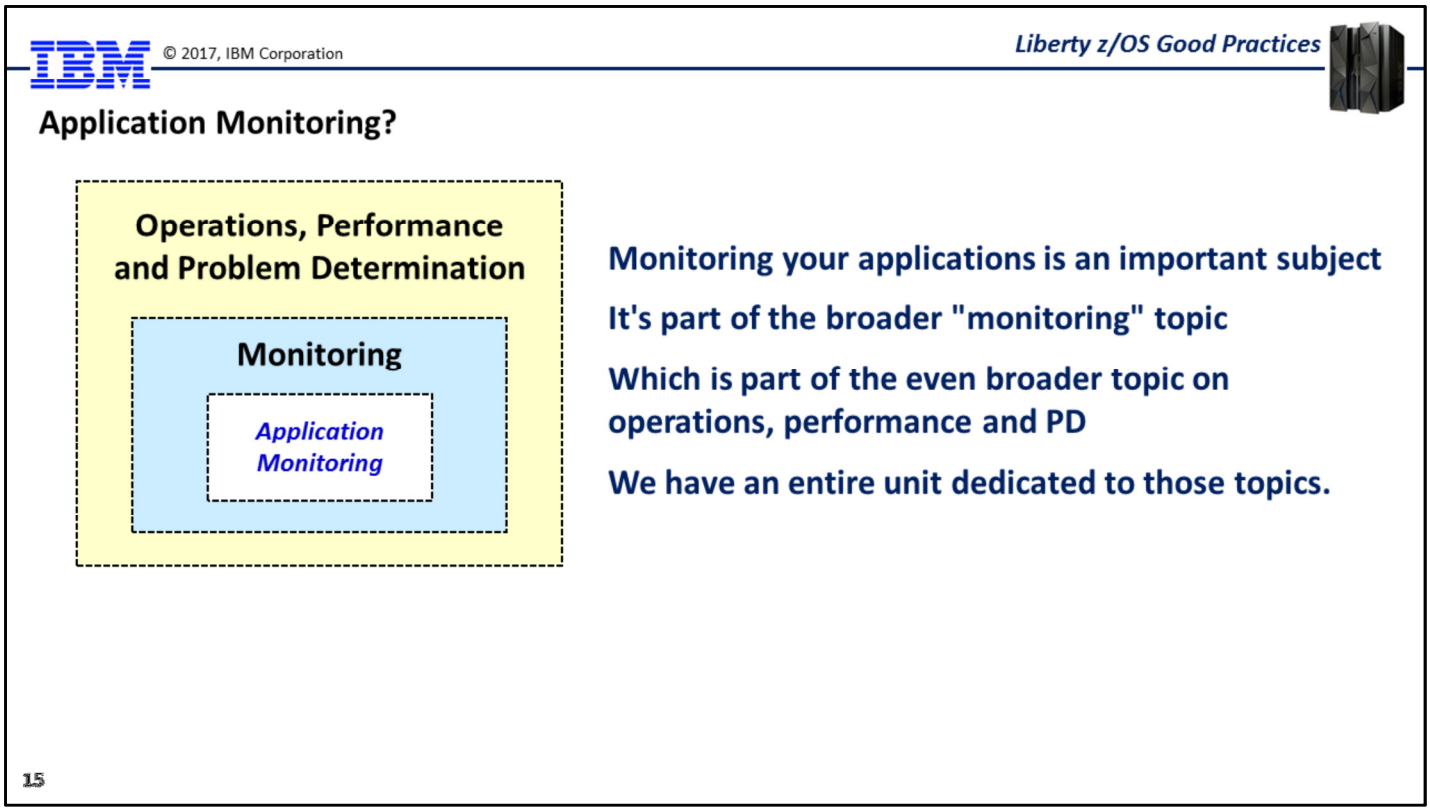
Doing this implies the servers are *not* operating with a polled dynamic update.

Note: polled dynamic update would work if the polling interval was sufficiently long to prevent all the applications being out of service simultaneously, and also that the polling intervals don't "pop" at the same time; that is, they are staggered. That's a fair amount of conditions. It's easier to assume either an mBean update or a server restart update.

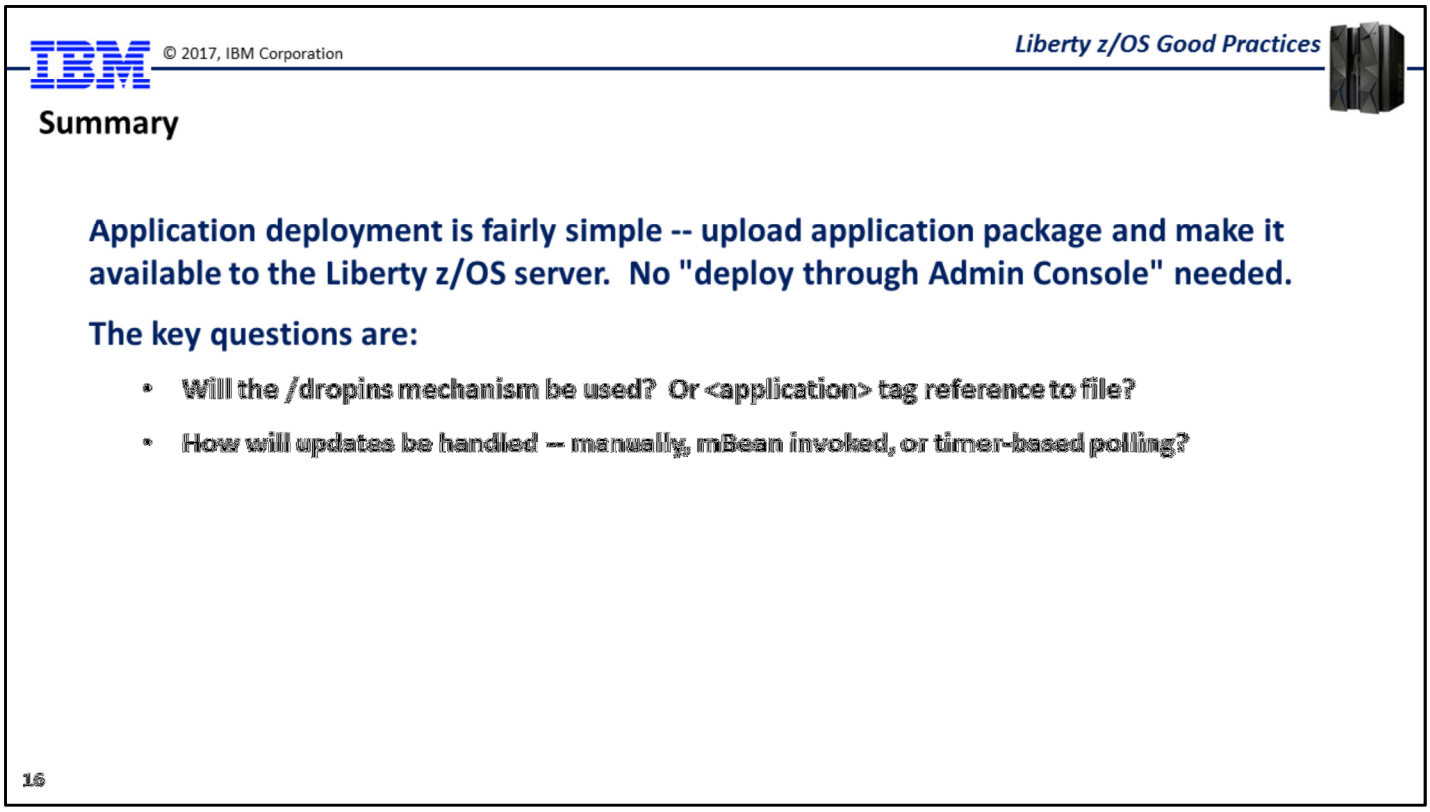
It's assumed a shared application is used between the servers -- after all, if each server had its own copy the application you'd simply deploy the application into each server in turn and roll through the servers manually. So with those assumptions in place (no dynamic update, and a shared application), then it's just a matter of updating the shared application and then triggering the update, either with an mBean or a server restart.

Some things that complicate this scenario:

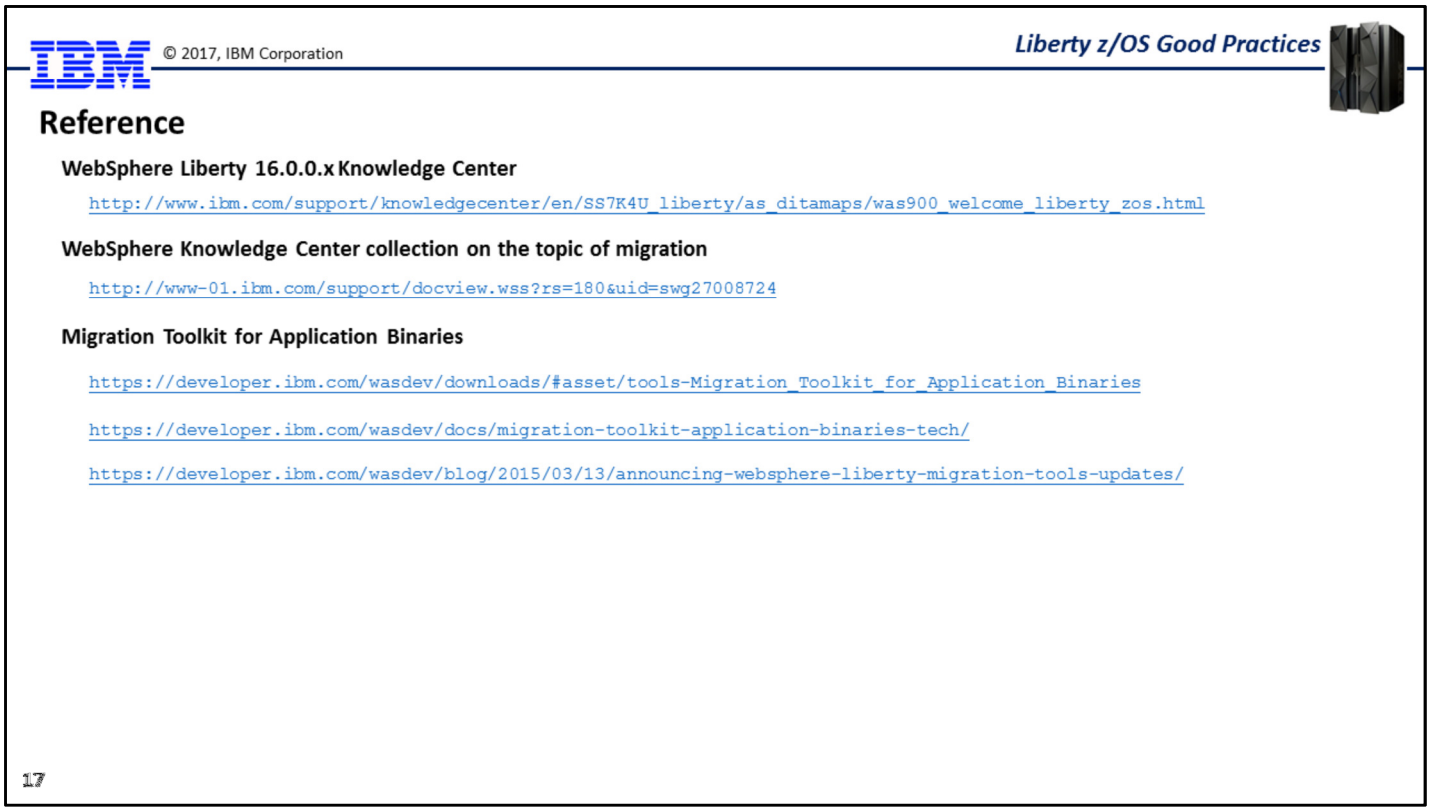
- If the applications are maintain session *state*, then that implies affinity, which implies the need to make sure any state information is available in the other servers if an established client lands elsewhere during the rolling of the applications. This is possible using session persistence (copying state information out to a database, then retrieving it back to another server). So while this makes rolling updates a bit more complicated, it's not impossible to engineer for this.
- When an application is being refreshed, there will be a brief period of time when it is not available. Clients routed to that server will get an error in that brief period of time. This can be avoided if the front-end work distribution function is capable of "turning off" work to a given endpoint. Work can then "drain" from that endpoint, the application restarted, and the router out front "turned back on" to allow work to flow to the updated server.
- "Rolling" an application implies a period of time when you have mixed levels of the application available to clients. That may be okay, or it may not be okay if the nature of the change being introduced is so significant you can't have mixed levels. If that's the case, then you will have to schedule an outage and refresh all instances of the application at the same time.



What about application monitoring? It's an important topic ... not just application monitoring, but the broader topic of monitor in general. We have a unit dedicated to monitoring, as well as operations and problem determination.



And we're to the summary page. Liberty application deployment is relatively simple because it is really just a matter of uploading the file and letting the server know about it. There are two things we discussed: (1) using "static" application definitions (an XML pointer to the application path and file name) vs. using the "dropins directory" location; and (2) whether application changes are dynamic, or are more manually controlled by you. We anticipate the dropins/dynamic model will be used for development and test, while static/manual will likely be the model for production settings. But you may choose differently based on your own set of criteria. In either case, Liberty z/OS can be configured to support what you want it to do.



Here is a reference page that lists a few resources on the topic of Liberty z/OS and migration.

End of Unit