GT4 Java WS Core Container Dynamic Deployment Design

June 14, 2006

Jarek Gawor {gawor@mcs.anl.gov}

Table of Contents

JAVA WS CORE CONTAINER DYNAMIC DEPLOYMENT DESIGN

ClassLoader Design

Java WS Core uses two classloaders. The 'common' classloader is responsible for the basic libraries to run the container such as the xml parser, axis, logging, security, etc. The 'common' classloader is **not** reloadable. The 'service' classloader is responsible for loading the service libraries and is reloadable. The 'service' classloader always first delegates a call to load a given class to the 'common' classloader (that is, the parent of the 'services' classloader is the 'common' classloader).
Currently, the 'common' classloader loads the libraries from the $G_L/lib/common directory while the 'service' classloader loads the libraries from '$G_L/lib/ directory.

Reloading

- During the deployment the container will accept new requests but return 'service unavailable' errors (to prevent deadlocks).
- Only one reload operation can take place at the same time. While reload is underway, any subsequent calls to the reload operation at the same time will cause an exception to be returned (to prevent deadlocks)
- When reload operation is first called, it will block until all currently executing requests finish or until the specified timeout expires (whichever one occurs first)
- A reload action can be passed to the reload operation. A reload action is an action that can be safely executed right after all the services are shutdown but before a new 'service' classloader is obtained. That means that a reload action can update the service libraries, configuration files, etc.
- If reload operation fails during executing the reload action, or getting a new 'service' classloader, or creating a new Axis Engine it will attempt to recover from the error. If reload operation fails in the last step of the reloading sequence (activating the services) recovery will not be attempted.

Reloading sequence

1. Put container is reload mode. That will cause the container to return 'service unavailable' error to any request the container receives during deployment.
    a. This step will also block until all currently executing requests finish or until the specified timeout expires (whichever one occurs first)
2. All services, resource homes, etc. are deactivated (see the 'deactivation' section)
3. Cleanup functions are performed to flush caches that might contain references to the classes loaded by the current 'service' classloader.
4. If a reload action was specified, it will be now executed.
5. A new 'service' classloader is created
6. A new Axis Engine is created using the new 'service' classloader

7. The container service threads are updated with the new 'service' classloader and the new axis engine
    a. If a service thread is currently processing a request, the update of the classloader and axis engine will happen after it is done processing the request.
8. The services, resource homes, etc. are initialized with the new 'service' classloader (see the 'activation' section in the Java WS Core Design Document)

Limitations

1. Security credentials of the service are currently not associated with the thread during service/home deactivation.
2. The 'home' name in JNDI is reserved for the ResourceHome associated with a service. Using the name 'home' outside of '*java:comp/env/services/<serviceName>'/* context is not recommended (if that objects implements the Destroyable interface it might not be called)
3. If one service misbehaves, for example, it doesn't shutdown all of its threads, close database connections, etc. that might eventually crash the container and/or other services and the 'service' classloader might not be properly garbage collected by the JVM.
4. The following global property cannot be changed between reloads: the base container directory, the schema directory, the container id, the base file persistence directory, and the default container port and host name.
5. Dynamic deployment cannot be used if two or more containers are started at the same time from the same installation.

Notes

- Services must implement the proper deactivation steps. The service class can implement the *javax.xml.rpc.server.ServiceLifecycle* interface while the ResourceHome class (or any other object in JNDI) can implement the *org.globus.wsrf.jndi.Destroyable* interface. If a service does not release all of its resources properly it might eventually crash the container and/or other services.
- If *Class.forName(String*) is not quite working use the *org.globus.util.ClassLoaderUtils.forName(String)* method instead.
- If *ClassLoader.getResourceAsStream(String)* is not quite working use the *org.globus.util.ClassLoaderUtils.getResourceAsStream(String)* method instead.

Other Changes

Container State Files

When the standalone container starts up it creates a unique state file under the *$GLOBUS_LOCATION/var/state* directory. Similarly, when the container shuts down, that state file is deleted. The state files are used by other programs to determine if the container is currently running. In certain cases, the container JVM might crash and the state files might not get deleted properly. In such situations the state files might need to be deleted manually.

Offline GAR Deploy/Undeploy Changes

The *globus-deploy-gar* and *globus-undeploy-gar* tools (and their corresponding Ant tasks) will now first check if the container is currently running (see the *Container State Files* section for details). If the container is running the deploy/undeploy operation will be aborted.

Also, the *globus-deploy-gar* tool will now first check if a given GAR is already deployed. If the GAR is already deployed, the operation will be aborted unless the user specifies the *'-overwrite'* option to overwrite the existing deployment. It is recommended to undeploy the existing deployment first before deploying the newer GAR.

Tomcat Notes

Container Differences

In Tomcat a web application can be stopped, started or reloaded at any point. The reload operation is really a stop, start sequence. It effectively it works just like the reload operation in the standalone container. The same deactivation and activation steps are performed.
Because of the different container design more of the common libraries needed by Java WS Core can be updated during a reload in Tomcat then in the standalone container. For example, *axis.jar* can be updated in Tomcat without a restart but cannot in the standalone container. However, there is a still a subset of libraries that cannot be updated without a restart in Tomcat. Any libraries that are put into the *$TOMCAT_DIR/common/lib* or the *$TOMCAT_DIR/server/lib* directory cannot be updated. For example, the security libraries are put into one of these directories.
Also, because in Tomcat the entire web application gets reloaded any global settings such as the container id or the global persistence directory, etc. can be changed between the reloads. However, in the standalone container only certain global settings can be changed. See the *Limitations* section for details.

The Deploy Service

Service

The DeployService is used to remotely deploy and undeploy GARs. It can also be used to force the container to reload itself. The DeployService is a secure service, configured just like the Shutdown service. That means that only the user that is running the container can only use the service by default.

The DeployService supports five operations: *upload()*, *download()*, *deploy()*, *undeploy(),* and *reload()*.

To deploy a GAR file, the GAR file first has to be transferred to the service. The GAR files can be transferred to the service in two ways:

1. via SOAP with Attachments using the *upload()* function
2. via the *download()* function, where the DeployService uses globus-url-copy to copy the GAR file passed by the user as a URL to the local file system.

One the GAR is transferred it can be deployed via the *deploy()* function. Once the GAR file is deployed, it is deleted. Any GAR can be undeployed via the *undeploy()* function (even if a GAR was deployed while offline). The logs of the deploy/undeploy operations are stored in the *$GOBUS_LOCATION/var/deployment/logs/* directory by default.

The *reload()* function forces the container to reload itself. It can be used to tell the container to simply refresh itself, or as a way force the container to re-read the service configuration (for example if the service configuration was updated).

Resource Properties

The DeployService publishes two resource properties:

1. '*undeployable'* - list of GAR IDs that can be undeployed
2. '*deployable'* - list of GAR IDs that can be deployed (transferred to the service but not yet deployed)

Limitations

1. The DeployService does not work in Tomcat
2. Since the DeployService relies on the underlying Ant tasks to perform the actual deploy/undeploy operations, it suffers from the same limitations as the tasks. For example:
    a. The undeploy operation might not remove all files deployed by the GAR such as for example the generated command line clients.
    b. The undeploy operation does not remove client-side type mappings

c. The undeploy operation does not address at all what happens to the persistent data of the service.

Clients

globus-remote-undeploy-gar

The *globus-remote-undeploy*-gar works just like *globus-undeploy-gar* and takes the same arguments. It just uses the DeployService to perform the undeploy operation.

globus-remote-deploy-gar

The *globus-remote-deploy-gar* is slightly different from the *globus-deploy-gar* but the concept is the same. It performs two operations by default. It transfers the GAR file to the DeployService and then automatically deploys it (the tool can be also be configured to perform transfer or deploy operation only). If a file path is passed to the tool, it will transfer the file to the service via SOAP with Attachments (the *upload()* function) using the MTOM format. If an URL is passed, the tool will call the *download()* function of the service, and let the service download the GAR file.

globus-reload-container

The *globus-reload-container* just calls the *reload()* function on the DeployService which causes the container to reload itself.