012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
1. Introduction
    1.1. Project/Component Working Name:
        Modularization of GlassFish using OSGi

    1.2. Name(s) and e-mail address of Document Author(s)/Supplier:
        Sahoo: Sahoo@Sun.COM

    1.3. Date of This Document:
        12 July 2008

2. Project Summary
    2.1. Project Description:
        OSGi [1] module system will be used to implement the primary features
        of GlassFish V3 release, such as: modularity, extensibility, and
        embeddability.

    2.2. Risks and Assumptions:
        Making the existing code base modular is the biggest risk, as it
        involves significant redesign of core functionality. Existing GlassFish
        implementation relies heavily on a single class loader hierarchy.
        OSGi hardly uses such parent delegation model for class loading.
        Adopting the new class loading scheme can be a challenge.
        More over, it can lead to some backward incompatibility. We will try
        to minimize incompatibility, if not completely avoid it. At this point,
        the only known incompatibility issue is with regards to support of
        "Classpath Prefix." This can't be supported.

3. Problem Summary
    3.1. Problem Area:
        Modularity, extensibility, and embeddability are the need of the hour.
        As Java EE stack matured over time, implementations that include
        the entire stack were perceived as heavy weight solutions. Not every
        application uses all the functionality offered by the stack. Users
        do not want to pay the runtime cost for features that they don't use.
        A modular application server based on OSGi addresses this as it can
        ensure that only required parts of the system are started.

        In order to cater to the needs of different classes of applications
        and their varying need for the stack, we have to come up
        with a solution that allows us to quickly build customized stack.

        We would like the implementation to be extensible so that
        capabilities of the system can be augmented after the system is
        shipped/installed. This is important for the targeted use case where
        user starts with a reduced set of requirement, and their requirements
        grows over time necessacitating installation of newer capabilities in
        the system. Such extensions does not have to be necessarily provided by
        GlassFish project team, so there is a need to expose well defined
        extension points as well.

        There is a need for embedding GlassFish in existing runtimes to augment
        their capabilities for greater adoption of GlassFish. There is
        often a need for an application server like GlassFish to be used in
        other self-contained application, but it has not been possible because
        of their size and difficulty involved in managing their lifecycle.

        A modular system should allow upgrading/patching of a subset of modules.
        It is expected that a modular system can be patched at runtime without
        requiring a complete restart of the system. Upgrading/patching and
        versioning go together. A complex product like GlassFish, which has so
        many modules developed by so many different groups, requires a well

defined versioning and patching policy.

If GlassFish can benefit from OSGi, why not applications deployed in GlassFish? Application developers would like to use sophisticated versioning, class loading, dependency management and component model of OSGi. There is a growing demand for servers that expose such facilities to application developers. So, we shall investigate the use of OSGi by applications.

There is a need for applications deployed in GlassFish to have controlled access to implementation classes. This is not only true for external GlassFish specific API classes, but also for various open source libraries, like Apache Commons, ASM, etc., that's used by GlassFish internally. Not only are these libraries often used by end user applications, they also have multiple versions out there. Our implementation can *not* be compatible with all versions of such libraries out there and we can not force applications to use any particular version of those libraries. So, as a work around, we have been forced to repackage such libraries into our private namespace like com.sun.org.apache. Not only repackaging has development and maintenance cost, it also had a runtime cost, for it has the undesirable side effect of not allowing applications to share one runtime instance of a library with implementation even when they intend to use the same version of the library.

3.2. Justification:

A modular application server is difficult to implement, but it has longer term benefits. It is definitely more maintainable. A modular application server based on OSGi will not only allow us to build different flavours/profiles of GlassFish, it will also make it easier to build new products based on GlassFish that target specific market segment. Substitutability of OSGi bundles opens the door for interesting mix-and-match type of distributions giving greater choice to users.

OSGi has a pretty established versioning and update policy, which GlassFish can benefit from.

So far, we have been exposing implementation classes to applications. There has been a long standing need to provide desired level of isolation between implementation and application class spaces on a per application basis. Use of OSGi addresses this need. It will also avoid the need to repackage libraries and thereby removing the short-comings associated with repackaging as mentioned in the previous section. A point worth noting here is that there were earlier proposals to address this issue by use of sophisticated class loader implementation, but they were not completely implemented as the change was considered too pervassive and risky at that point of time. OSGi makes it easier to implement this feature.

4. Technical Description:

4.1. Details:

We propose to implement GlassFish as a set of OSGi bundles. A bundle is a unit of deployment in OSGi. It is packaged as a Java ARchive (jar) with proper manifest entries. The manifest entries define the essential details about the module, e.g., name, version, its dependencies and capabilities. At the heart (core) of GlassFish, there will be an OSGi framework. Although we plan to use Apache Felix [6] as the default OSGi framework, we do not want to make any such assumption in our code. So, the core implementation only uses OSGi R4 API, which makes it possible to run with alternative OSGi runtimes by changing configuration. Whether alternative runtimes will be supported or not is not an engineering decision.

GlassFish developers can make use of HK2 component model, which substantially eases the development of Service oriented application. We shall provide bi-directional mapping between HK2 service and OSGi service [7].

Since GlassFish will be implemented as a set of OSGi bundles, distributions of with different capabilities can be made by packaging suitable set of bundles. Once a particular distribution is installed, bundles can be added, removed or updated to add, remove or update system's capabilities.

It will be possible to embed GlassFish in existing OSGi runtime. A typical example of taht would be embedding GlassFish in Eclipse IDE, which starts its own OSGi framework called Equinox.

Now, a note about how traditional Java EE applications would run in the new runtime. There are two alternatives that come to my mind:
A) Java EE applications runs outside of an OSGi bundle context,
B) Java EE applications runs within an OSGi bundle context.
Approach #B requires converting user supplied, non-OSGi applications to be converted to equivalent OSGi bundles at runtime. This is inherently a risky proposal, as the conversion is not always that straight forward and can require user intervention. More over, the Java EE class loading scheme may not fit well in this model. This will affect use of many existing libraries and frameworks that rely on Java EE class loading scheme. This severly impacts backward compatibility of the server. However, approach #A does not have most of these problems. So, we prefer approach #A.

Having said that, we do want GlassFish to be able to accept OSGi-ed applications, irrespective of them being Java EE applications or not.

4.2. Bug/RFE Number(s):

4.3. In Scope:
We will try to make the server compatible with other popular OSGi platforms. We will provide necessary maven plugins and build script support to help developers build OSGi bundles. We will make various Java EE APIs available with proper OSGi metadata.

4.4. Out of Scope:
OSGi facilities can't be made available to standard (non-OSGied) Java EE applications in this release. We will experiment with running OSGi-ed Java EE applications. We will not wrap commonly used libraries as OSGi bundles. Instead, we will encourage people to get such wrapped bundles from projects like "Apache Felix Commons [4]." This project team shall modularise the kernel, define extension points, provide infrastructure for modularising GlassFish and come up with best practices, but won't be responsible for actually modularising every functional area. That has to be done by owners of respective functional area. Patching/Upgrading at runtime is not going to be implemented in Prelude release.

4.5. Interfaces:

4.5.1 Exported Interfaces
Interface: Felix remote shell
Stability: External
Comments: This is a simple command line shell to administer the OSGi framework remotely.

Interface: asadmin GUI and CLI enhancements to administer OSGi framework
Stability: Evolving

Comments: This will not be done as part of V3 Prelude release.

Interface: HK2 Module API and component model
Stability: Evolving

4.5.2 Imported interfaces

Interface: OSGi R4 API
Stability: Standard
Exporting Project: Felix (http://felix.apache.org)

4.6. Doc Impact:
We expect chnages to administration guide and classloader chapter of developers' guide and application development guide. We also expect new chapters to be written to explain how to develop plugins for GlassFish.

4.7. Admin/Config Impact:
In this release, we will provide administration commands for OSGi bundle lifecycle management. I expect commands to be very similar to what is available in Felix shell which look like the following:

```
headers [<id> ...]        - display bundle header properties.
install <URL> [<URL> ...] - install bundle(s).
packages [<id> ...]       - list exported packages.
ps [-l | -s | -u]         - list installed bundles.
refresh [<id> ...]        - refresh packages.
resolve [<id> ...]        - attempt to resolve the specified bundles.
services [-u] [-a] [<id> ...]  - list registered or used services.
start <id> [<id> <URL> ...]    - start bundle(s).
stop <id> [<id> ...]           - stop bundle(s).
uninstall <id> [<id> ...]      - uninstall bundle(s).
update <id> [<URL>]            - update bundle.
```

I am hoping Admin spec to provide further details. This need not be done for Prelude as we have an alternative solution available in the form of Felix shell. More over, this can always be done as an additional module and made available in UC. In a future release, we will allow user to configure the OSGi framework via GlassFish configuration management facilities. More over, the Admin/Config team should explore the use of OSGi facilities like "Configuration Admin Service" specification in V3.

4.8. HA Impact:
Not Applicable as of now, as V3 Prelude release does not have HA features.

4.9. I18N/L10N Impact:
Class loading restrictions in an OSGi environment applies to resource bundles as well. It requires changes to our code which has so far been assuming that all the necessary resource bundles are always available to all the implementation modules.
In the past, we have been using Class-Path manifest headers in various jar files to specify jar files containing localized resources. e.g., appserv-rt.jar in GlassFish V2 has the following entry:

```
Class-Path: appserv-rt_ja.jar appserv-rt_zh.jar appserv-rt_fr.jar
            appserv-rt_de.jar appserv-rt_es.jar appserv-rt_it.jar ...
```

In GlassFish V3, Class-Path manifest header has little use. It is recommended to use *fragement bundles* to provide localized content.

4.10. Packaging & Delivery:
To be discussed in packaging proposal[5].

4.11. Security Impact:
The OSGi Security Layer is an optional layer that underlies the OSGi Service Platform. The layer is based on the Java 2 security architecture.

4.12. Compatibility Impact
Since the server would be using new class loading scheme, there will be some impact on backward compatibility. Details to be discussed soon.

4.13. Dependencies:
1. Felix OSGi framework:
   http://felix.apache.org
2. Maven bundle plugin developed as part of Felix project:
   http://felix.apache.org/site/maven-bundle-plugin-bnd.html
3. Felix shell service and remote shell

5. Reference Documents:
1. OSGi Module System:
   http://osgi.org
2. GlassFish V3 Overview Functional Specification:
   http://wiki.glassfish.java.net/Wiki.jsp?page=V3OverviewFunctionalSpec
3. HK2 component model:
   https://hk2.dev.java.net/components.html
4. Apache Felix Commons project:
   http://felix.apache.org/site/apache-felix-commons.html
5. GlassFish V3 Packaging proposal:

http://wiki.glassfish.java.net/attach/V3FunctionalSpecs/V3PreludePackagingFileLayout.txt
6. Apache Felix project:
   http://felix.apache.org
7. Providing HK2 services on OSGi:
   http://wikihome.sfbay.sun.com/asarch/attach/Attachments%2FHK2OSGi.pdf

6. Schedule:
6.1. Projected Availability:
Same as GlassFish V3 release dates. Not everything proposed in this document can be implemented in one release. We plan to make the initial implementation ready by V3 Prelude release. The initial implementation would have the modular architecture in place allowing for extensions to be made on top it.