GlassFish V3

Engineer's guide

rev 0.2 10/08/07

Jerome Dochez

This document is basically a brain dump of how V3 is organized today with some insights on how it could be re-architecture along the V3 engineering effort.

Table of Contents

1 Modules

GlassFish v3 is build around a module subsystem. Functionalities are declared through a contract/service implementation paradigm. See http://hk2.dev.java.net for more details on modules and components.

1.1 What is a module

A module is an encapsulated definition of reusable code. It offers services, requires other modules's services, and have public APIs as well as a private set of classes implementing the public APIs. There is a lot of literature around modules but to be short :

- is a set of java classes
- offers services or a public API or both
- implements those public interfaces with a set of private classes.
- requires other modules (dependencies)

1.2 How to define a module

Defining a module is not always easy and will in most cases be an iterative process. Some of the criteria to define a module can be summarized :

- Is my code offering very different kind of services
- Is my code offering some services independently of other services it supports
- Is my code used by more than one module.

If part of your code can answer yes to any of these three questions, chances are that this code need to be encapsulated in a module.

The easiest starting point is to check if your code offers more than one service. For instance, the current connector container offers 2 services to GlassFish v2, it's both the jdbc driver container and the Java EE resource adapter container. It could make sense to separate these behaviors in two modules so they can be used independently.

From there, you also need to look into other services that your "code" offers to GlassFish. Don't create a module just for the fun of it, we should create a module only if it contains a piece of code that can be reused independently by other components (or just like above to make things more encapsulated/efficient).

1.3 Module definition

Once you have determined a set of classes that you want to offer as a module, you need to start thinking about its definition. The definition is a name and a version that will be used by other modules to identify their dependency on your module.

The next aspect of modularization is to figure out the exact dependencies that your modules have. For instance, you may depend on the com.sun.enterprise:config-api (name of the module constructed from a group id and an artifact id) to get the configuration out of the domain.xml. What else ?

It is quite important to understand those dependencies not only to successfully compile your module(s) but also to understand what runtime services are necessary for your module to function properly.

So as a generalist introduction on how to change some code into modules :

- look at how many module you want to split your code into with the following criteria
  - re-usability
  - isolation (can I use a feature without the others)
  - efficiency (should I load 3Mb of jar files when I only use 100K of classes 95% of the time)
- figure out the exact dependencies for each identified module

Once you have that information even in an initial raw format, you can start thinking about refactoring the code or reduce the dependency trail and so on.

2 Modules management

2.1 GlassFish Distributions

GlassFish V3 is built as a distribution artifact. There will be various distributions of GlassFish V3 each providing a unique set of features. Elements of a distribution are integrated using maven 2 repositories, so adding a module to a GlassFish v3 distributions necessitate three steps.

1. compile the module and create the jar file with the necessary hk2 manifest entries
2. publish the module artifacts to a maven repository.
3. add the module's identification to the distribution list.

As you can see, distributions are constructed out of integration of binary files procured from a maven repository, there is a complete separation between how to produce a particular module versus how this module is integrated into a distribution.

2.2 Building a module

Each module can be integrated in various distributions hence there cannot be any distribution specific knowledge introduced during the module's build.

Although a module could be potentially built with other tools, it is highly recommended that a module be built with maven2. The Maven 2 building environment gives us the ability to run a number of maven plugins that can do some of the boiler plate activities such as HK2 manifest entries, components definition and so on.

A module source is usually comprised of the following artifacts

- source code usually located in src/main/java
- resources usually located in src/main/resources
- a maven 2 pom.xml defining the module compilation process and the module's dependencies.

A complete description of maven 2 pom.xml file is located at http://maven.apache.org but in most cases, individual contributors will not need to understand all the features of maven to build simple modules.

2.3 pom.xml structure

The basic structure of a pom.xml is as follows :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

    <parent>
        <artifactId>bootstrap</artifactId>
        <groupId>com.sun.enterprise.glassfish</groupId>
        <version>10.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.jvnet.glassfish</groupId>
    <artifactId>glassfish-api</artifactId>
    <packaging>hk2-jar</packaging>
    <version>${project.parent.version}</version>
    <name>Public APIs of Glassfish V3</name>

    <developers>
     <!-- information on the module's developers -->
    </developers>
    <dependencies>
     <!-- Dependencies on other modules/jars -->
    </dependencies>
</project>
```

The parent definition is important as it includes all the system wide properties like version identification and basic compilations and packaging rules for your module.

Other important information :

1. the name of your module is the groupId:artifactId tuple, here org.jvnet.glassfish:glassfish-api
2. the version is coming from the parent pom.xml which in this case is 10.0
3. packaging is hk2-jar which makes this created bundle an HK2 jar with the right manifest entries and components definition.

2.3.1 adding a developer

Adding a developer to the module is defined at http://maven.apache.org/pom.html#Developers, but here is an example of adding myself as the module owner (lead) and developer.

```
   <developers>
        <developer>
            <id>dochez</id>
            <name>Jerome Dochez</name>
            <url>http://blogs.sun.com/dochez</url>
            <organization>Sun Microsystems, Inc.</organization>
            <roles>
                <role>lead</role>
                <role>developer</role>
            </roles>
        </developer>
    </developers>
```

2.3.2 adding a dependency

Once your module uses other modules to compile properly, you need to add a dependency declaration to the pom.xml

```
    <dependencies>
        <dependency>
            <groupId>com.sun.enterprise</groupId>
            <artifactId>hk2</artifactId>
            <version>${hk2.version}</version>
        </dependency>
    </dependencies>
```

In the example above you can see that the module is using some hk2 core APIs. Whether you depende on a module (hk2-jar) or a normal jar file does not change how you declare a dependency, the system will figure out the complete lists of dependencies,

2.4 Licenses

Modules inherit the parent module's licenses information, but if your module is not part of GlassFish's source license agreements, you should add the following declaration to your pom.xml

```
<licenses>
  <license>
    <name>Apache 2</name>
    <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
    <distribution>repo</distribution>
    <comments>A business-friendly OSS license</comments>
  </license>
</licenses>
```

2.5 Source code management

Each module is basically free to host its source code where it wishes. The only condition for a module to be included in a GlassFish distribution is to publish the module binaries to a maven repository. If a module is hosted by a different java.net project than glassfish.dev.java.net, its pom.xml should include a source code management entry and a issue tracker management entry like :

```
     <scm>
        <connection>scm:hg:http://mercurial.sfbay.sun.com/glassfish_v3</con
nection>
    </scm>
    <issueManagement>
        <system>IssueTracker</system>
        <url>https://glassfish.dev.java.net/servlets/ProjectIssues</url>
    </issueManagement>
```

2.6 Build magic

2.6.1 maven plugins

A number of plugins have been developed to help the task of building modules. These plugins will use the pom.xml or the annotated classes to generate the metadata associated with a module like its HK2 manifest entries as well as its components definition. These plugins are declared in the GlassFish bootstrap pom.xml (used as a parent for each module pom.xml) therefore relieving the module owner from having to call the plugins themselves or create the metadata by hand.

2.6.2 ant

It is still sometimes useful to call ant tasks while building your module, here is an example on how to do that :

```
        <plugin>
            <artifactId>maven-antrun-plugin</artifactId>
            <executions>
              <execution>
                <phase>process-sources</phase>
                <configuration>
                  <tasks>
                        <copy file="jsp-api.mf"
                                toFile="${project.build.directory}/manifest
.mf">
                            <filterset>
                                <filter token="VERSION"
value="${jsp.spec.version}"/>
                            </filterset>
                        </copy>

                  </tasks>
                </configuration>
                <goals>
                  <goal>run</goal>
                </goals>
              </execution>
            </executions>
          </plugin>
```

2.6.3 javadoc

Javadocs can be automatically created by calling mvn javadoc. You can also influence how the javadoc will be created by customizing the plugin that calls the javadoc tool.

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
            <executions>
              <execution>
                <phase>javadoc</phase>
                <goals>
                    <goal>javadoc</goal>
                </goals>
                <configuration>
                  <groups>
                    <group>
                      <title>JavaServer Pages API Documentation</title>
                      <packages>javax.servlet.jsp</packages>
                    </group>
                  </groups>
                  <bottom>Portions Copyright &amp;copy; 1999-2002 The
Apache Software Foundation.  All Rights Reserved. Portions Copyright
&amp;copy; 2005-2006 Sun Microsystems Inc.  All Rights Reserve</bottom>
                </configuration>
                </execution>
            </executions>
      </plugin>
```

2.7 Distribution Dashboard

Each distribution will have a dashboard wiki page (see http://wiki.glassfish.java.net/Wiki.jsp?page=pe-10.0-SNAPSHOT for instance) which will extract information from all the modules of the distribution. The information in the pom.xml of each module is used to create the dashboard hence it is important to keep the information as up to date and complete as possible.

2.8 Contributors Dashboard

Each distribution will have a contributors dashboard where all contributors per module will be present. The information will be extracted from the pom.xml information as specified in 2.3.1.

3 Runtime

3.1 Startup sequence

GlassFish is now an HK2 executable therefore

- it has no main() or Main class
- it is implemented as a set of modules.
- the first code to be executed (often referenced as the bootstrap module) is a small jar file called glassfish-{Version}.jar

The bootstrap module which is not a real module since it is loaded by the application class loader, has the following responsibilities :

1. identify the first HK2 module that will be loaded by the module subsystem. That module must be an HK2 module and must have a ModuleStartup interface implementation. This module is identified as the unique import of the bootstrap module, therefore the bootstrap module cannot have more than one dependency declared in its manifest.
2. optionally set up the parent class loader of all the modules loaded by the HK2 module subsystem.

Once the bootstrap jar has identified the first glassfish V3 module called com.sun.enterprise:v3-core, the module will be loaded and the ModuleStartup contract implementation will be instantiated and constructed according to HK2 component model (see http://hk2.dev.java.net).

When embedded, GlassFish V3 startup sequence is essentially unchanged, the outer Java environment decides in which classloader the V3 bootstrap module should be loaded essentially defining the parent classloader for V3.

At the end of this startup sequence, this is how the classloader hierarchy is established in V3 :

1. System Class Loader : JDK classes
2. Application Class Loader : all jar files present in the -cp parameter during the jvm invocation or in the embedded case the classloader used to load the glassfish-{version}.jar file.
3. MaskingClassLoader : parent classloader for all V3 modules, initialized by the glassfish-{version}.jar, its main responsibility is to mask the jdk classes so we don't have to use mechanisms such as endorsed directories when using newer implementations of technologies than the JDK bundled versions.
4. com.sun.enterprise:v3-core class loader.

3.2 Services

3.2.1 Services defined

Programming with modules is difficult, but moreover, does not necessarily mean that GlassFish can run parts of its functionalities based on users actions. If all the modules forming a GlassFish distribution were referencing each other, the entire set of modules would be loaded at startup basically erasing all the modularization advantage.

To isolate the engineers from programming with modules directly, there is a concept of Services in glassfish. These services are usually POJO implementing an interface. At runtime, glassfish relies on services implementation lookup to find the services for a particular interface (called contract).

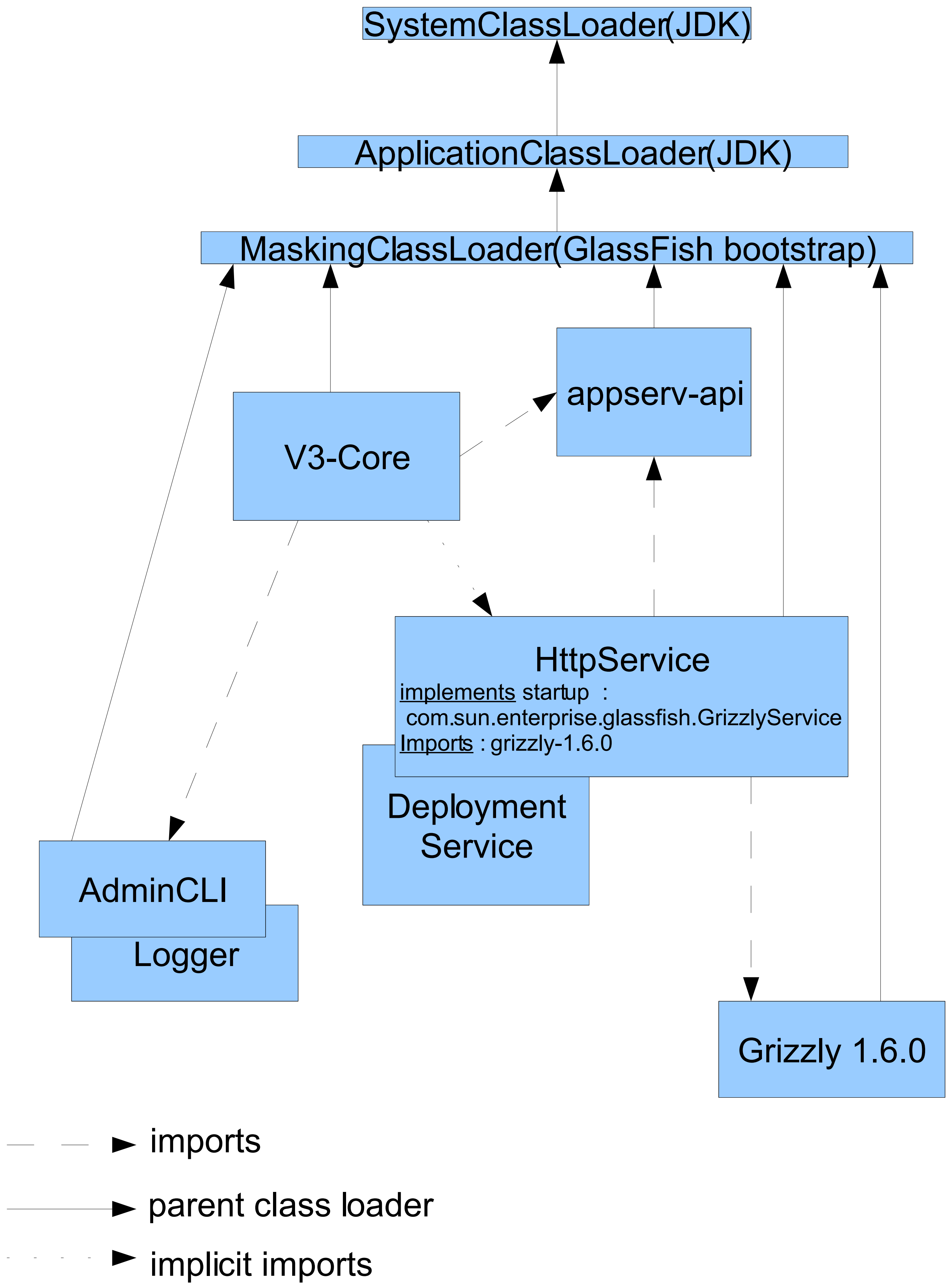
Let's take an example : com.sun.enterprise:appserv-api module which is the module containing all GlassFish V3 public API contains an interface called Startup. That interface defines code that must be run when GlassFish is started (kind of equivalent to the Lifecycle implementations in V2). Any modules in the GlassFish distribution can have one to many implementations of the Startup interfaces called startup services.

When glassfish is started, the ModuleStartup code from the com.sun.enterprise:v3-core module described earlier will do the following HK2 lookup :

```
Collection<Startup> startups = habitat.getAllByContract(Startup.class);
```

Even if v3-core module does not import directly the modules implementing the startup services, the collection returned by the above code will contain those implementations. This is an implicit import of instances from one module to another while the classloaders are not directly referencing each other.

Each module containing one to many Startup implementations can also bring up other modules through the dependency mechanisms. For instance the HttpService module which contains a Startup service can declare its dependency on the grizzly module. Once the Startup services have run successfully this is how the classloader hierarchy will look like.


SystemClassLoader(JDK)
ApplicationClassLoader(JDK)
MaskingClassLoader(GlassFish bootstrap)
V3-Core
appserv-api
HttpService
implements startup :
com.sun.enterprise.glassfish.GrizzlyService
Imports : grizzly-1.6.0
Deployment Service
AdminCLI
Logger
Grizzly 1.6.0
imports
parent class loader
implicit imports

In the example above, please note the following :

- v3-core imports admin-cli and logger modules.
  This is a direct dependency from v3-core to those modules because v3-core uses directly classes that are packaged in these modules
- v3-core does not imports HttpService or DeploymentService modules
  At runtime, v3-core just looks up the Startup implementations. HttpService and DeploymentServices both offers Startup services implementation which are looked up by HK2 component manager, instantiated, injected and returned to the v3-core.
- All modules have the same parent class loader.
- appserv-api module defines the Startup interface which is the contract for the service and is imported by both the v3-core and the modules containing a Startup implementation.

3.2.2 Add a new type of service

A new type of service or contract is just a plain java interface annotated with the @Contract interface.

```
@Contract
public interface Startup {
}
```

Remember that @Contract annotated classes must be accessible from the modules implementing the contract (service providers) as well as the modules looking up such implementations. In the example above the Startup interface is defined in the appserv-api module and is imported by both the v3-core module which requests all Startup implementations and by the modules defining the implementations (HttpService module)

3.2.3 Add a new service implementation

Adding a new service type is as easy as defining a new POJO implementing a contract and annotated with @Service class.

```
@Service
public class RandomService implements Startup, PostConstruct {

     public void postConstruct() {
          logger.info("I am not that useful yet");
     }
}
```

Service implementations can be located in any module packaged in the distribution.

3.2.4 Configuration

Configuration of a service can come from either the configuration file (domain.xml) or other component implementations. When a service needs to get some configuration from the domain.xml, it should declare itself with the @Configured interface.

The combination of @Attribute and @Element to bind to xml attributes and xml elements can be used to denote which information should be fetched from the configuration file.

```
@Configured
public class HttpService
        implements Serializable {

    @Element
    protected AccessLog accessLog;
    @Element(required=true)
    protected List<HttpListener> httpListener = new ArrayList();
...
}
```

Although it could be conceivable that an annotated @Configured class is also a service implementation, this sort of converged @Configured plus @Service component is discouraged. Most @Configured objects need to be available and be configured without an associated runtime (so the admin-gui can configure the http-service in the example used above without necessarily starting the http service). It is therefore recommended to split the configuration from the behavior into two POJOs and implement the following pattern :

```
@Service
@ConfiguredBy(HttpService.class)
public void GrizzlyService implements Startup {

     @Inject
     private HttpService config;

...
}
```

3.2.5 Existing services

A handfull of services have already been identified and added to the org.jvnet.glassfish:appserv-api module which represents the public APIs of the GlassFish implementation.

Brief introduction of these services here. more to come in the following paragraphs.

1. Startup : services which are run upon GlassFish startup.
2. Sniffer : services responsible for identifying deployment artifacts and setting up associated containers.
3. ArchiveHandler : services that recognize application bundle types (WAR, EAR...)
4. Deployer : services responsible for deploying artifacts to a particular container
5. AdminCommand : services which represent a CLI administrative command

3.3 Application Deployment

GlassFish v3 needs to support the convergence model for applications where users can ship application bits targeted to different containers all in the same package. This is a break from how we have traditionally considered application delivery in the Java EE world where each container had a dedicated bundle format. For instance, web containers expect war files, ejb containers expect jar files.

In v3, several types of applications can be co-packaged in the same bundle while still requiring some form of cooperation among the application parts (requiring a single class loader to load the application bits to be shared by all containers). There cannot be an assumption on the format of the bundle where a container can load the application bits from, therefore the containers do not control how the class loader for the applications bits are created, this is handled through another type of deployment service called ArchiveHandler.

There can be only one ArchiveHandler for a particular bundle type, the archive handler implementation will be responsible for creating an appropriate application class loader that can successfully load classes and resources from that bundle type. GlassFish V3 will create a parent classloader to be used by the ArchiveHandler when creating the application class loader. This parent class loader will allow Glassfish runtime to dynamically add imports to the application (see below). Each Sniffer implementations have the ability to setup their associated containers if the bundle appears to contain components they support. Once the sniffers have run successfully, all Deployer services currently found by HK2 will have a chance to deploy bits of the bundle to their associated containers.

Deployers will either claim part of the application bits or not. If a particular deployer instance successfully process a deployment request using the class loader created by the ArchiveHandler. the public APIs associated with that container (eg the Java EE APIs) will be added to the parent class loader (create above for that purpose).

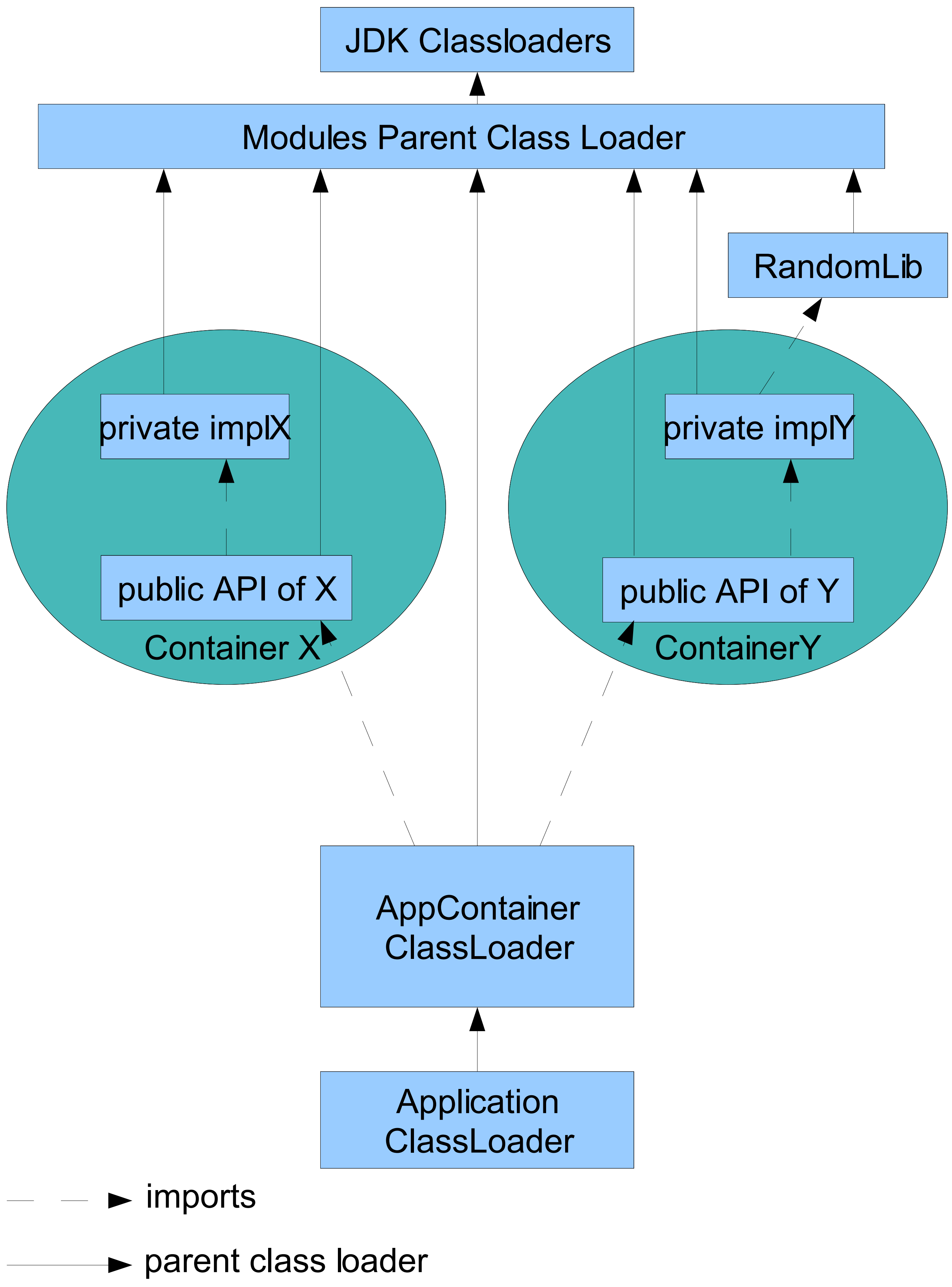
Remember that supporting different types of bundle is a desired feature, therefore, power users can have the ability to create/register new ArchiveHandler. Since ArchiveHandler are responsible for creating the classloaders, there is no possibility to directly add the container's public API to these classloaders as they are simple java.lang.ClassLoader implementations. This the reason behind the creation of this dynamic parent class loader. There is however a slight limitation to this scheme, the ClassLoader implementation returned by the ArchiveHandler should support the registration of java.lang.instrument.ClassFileTransformer instances to support bytecode enhancement dependent technologies like JPA. The EjbClassLoader and WebClassLoader that we ship with V3 both support such registration.

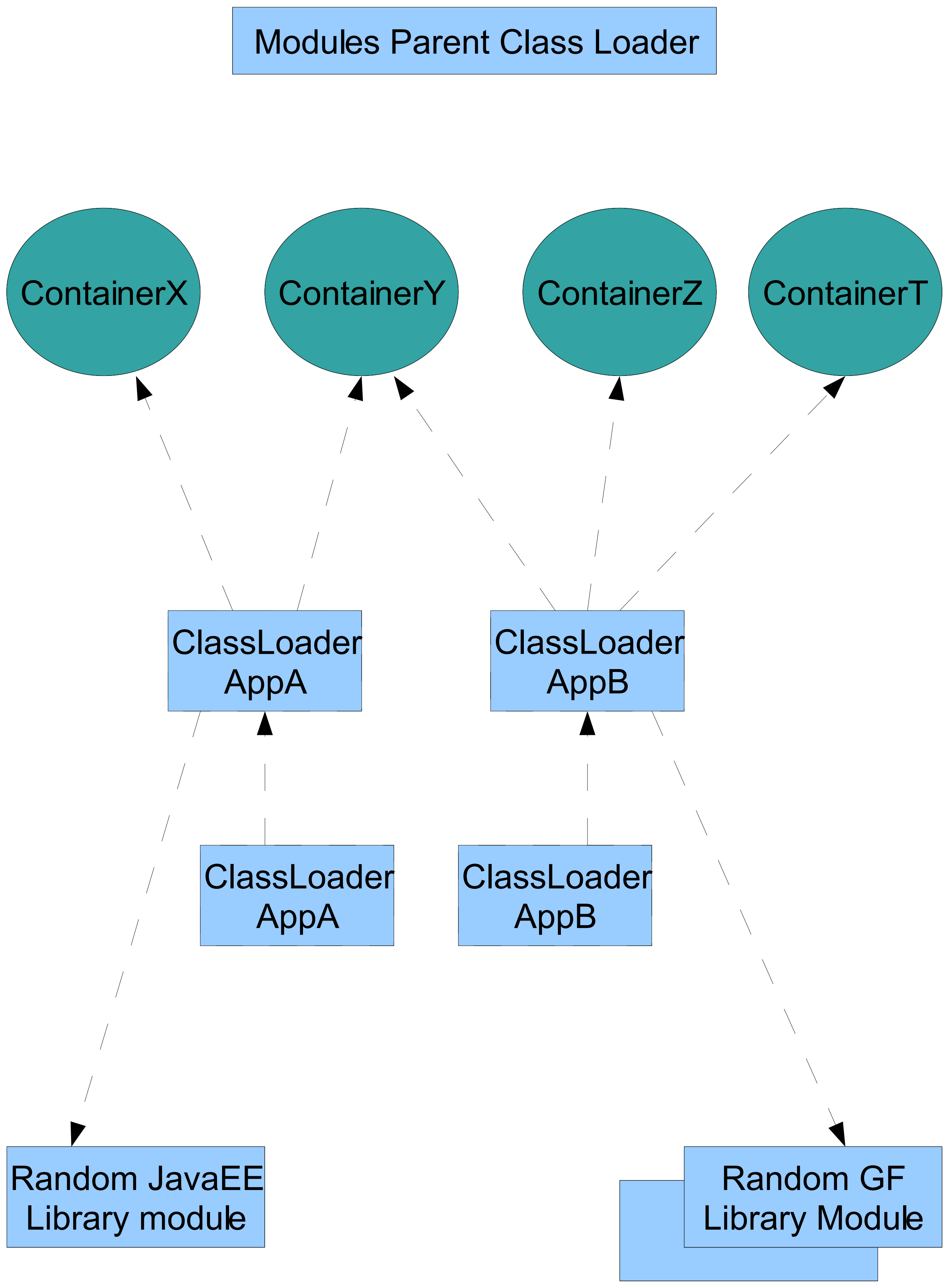
In the diagram below that we have two containers, one application being deployed to both containers and one random library module used by one of the container. Note the following :

1. the application bits are loaded from the bundle using the "Application Class Loader" that is created by the ArchiveHandler that claimed handling the bundle type.
2. the parent class loader of that "Application Class Loader" created by GlassFish v3 is a module class loader where new module imports can be added on the fly as container's deployers declare their support for part (or the whole) application components.
3. the private classes of each containers are not accessible by parent class loader (2).
4. the public API of each container are usually the APIs used by the components (like javax.ejb for instance) plus any other classes that may be required by generated code like stubs.

In the second diagram, you will see a simplified version of the first diagram showing the relationships between two applications and the containers they are successfully deployed into. Note the following :

1. AppA is deployed to containerX and containerY
2. AppB is deployed to containerY, Z and T
3. Each application has a class loader (created by the ArchiveHandler) and a parent class loader which is a module class loader created by GlassFish and providing the application with all the necessary public APIs.
4. Each container is a set of public and private modules.


JDK Classloaders
Modules Parent Class Loader
RandomLib
private implX
public API of X
Container X
private implY
public API of Y
ContainerY
AppContainer ClassLoader
Application ClassLoader
imports
parent class loader


Modules Parent Class Loader
ContainerX
ContainerY
ContainerZ
ContainerT
ClassLoader AppA
ClassLoader AppB
ClassLoader AppA
ClassLoader AppB
Random JavaEE Library module
Random GF Library Module

4 Persistence

Implementing persistence in V3 proposes several challenges due to the age and usage patterns of some JDBC classes, the multiple implementations of the javax.persistence APIs and it's capacity in running in Java SE mode (even embedded in GlassFish) or in Java EE mode.

The following patterns have been identified :

1. Applications can do a Class.forName() call to load a JDBC driver and make direct calls through the JDBC APIs to the underlying database. Applications do not need to identify special interest in such drivers, there is no indication they will use such low level access APIs.
2. Applications can decide to use the default JPA provider, lettingGlassFish choose which one to use.
3. In Java SE mode, appplications can use the PersistenceProvider.createEntityManagerFactory() call to get a specific persistence provider (name extracted from the peristence.xml).
4. In Java EE mode, GlassFish is responsible for loading the persistence.xml file, and looking up the expected persistence provider according the its settings (potentially defaulting the provider name).
5. In Java EE mode, GlassFish is responsible for identifying the connection pool to be used with the entity manager. This resource adapter is retrieved at deployment time from the JNDI name space and wired up with the persistence provider to create the entity manager.

Stranger pattern not necessarily supported by the JPA or Java EE specs.

6. A Java EE application can co-bundle a persistence provider. This is usually done for applications targeted to Tomcat, and therefore it is quite possible to find such packaging.
7. Some Java EE components like entity beans can be installed in shared environment that should be accessible by all deployed applications.

To resolve the problems at end, a shared library module will be introduced. This module will load all jar files located in the glassfish/lib and glassfish/javadb/lib directories (remember that v3 modules and jars will be relocated somewhere else). This shared library will have access to the JDBC drivers as well as javadb runtime.

The solutions below reference the diagram next page, the number in parentheses indicate which class loader relationship will satisfy the requirement.

4.1 Java SE mode

4.1.1 case 1

Applications can load JDBC drivers using the Class.forName() API. The parent class loader of the application class loader (named the application container class loader) which gives access to the public APIs of an application will also include the shared library class loader (1)

4.1.2 case 2 and 3

In Java SE mode, applications use javax.persistence.PersistenceProvider.createEntityManagerFactory() API to create an entity manager. Applications use a persistenceUnitName parameter to identify the provider in the persistence xml file. HK2 will make the META-INF/services resolution for the

PersistenceProvider.java implementation and JPA will be loadedlike any other HK2 service.

The JDBC driver is provided by some non portable persistence.xml properties and is loaded directly by the persistence provider (2)

4.2 Java EE mode

In Java EE, there is a need to add ClassFileTransformers to the application class loader. This is a bit tricky as we don't necessarily provide the class loader for the application bits (see Deployment above). JPA infrastructure will have a Deployer implementation, therefore the prepare phase should be used by the JPADeployer to pre-load classes that may require bytecode enhancements. During the prepare phase, the JPADeployer should also add some java.lang.intrument.ClassFileTransformers) instances to the DeploymentContext. The Deployment backend will then recreate a new class loader, register all installed ClassFileTransformers in this new class loader instances and will invoke the load phase with the new classloader.

4.2.1 case 4

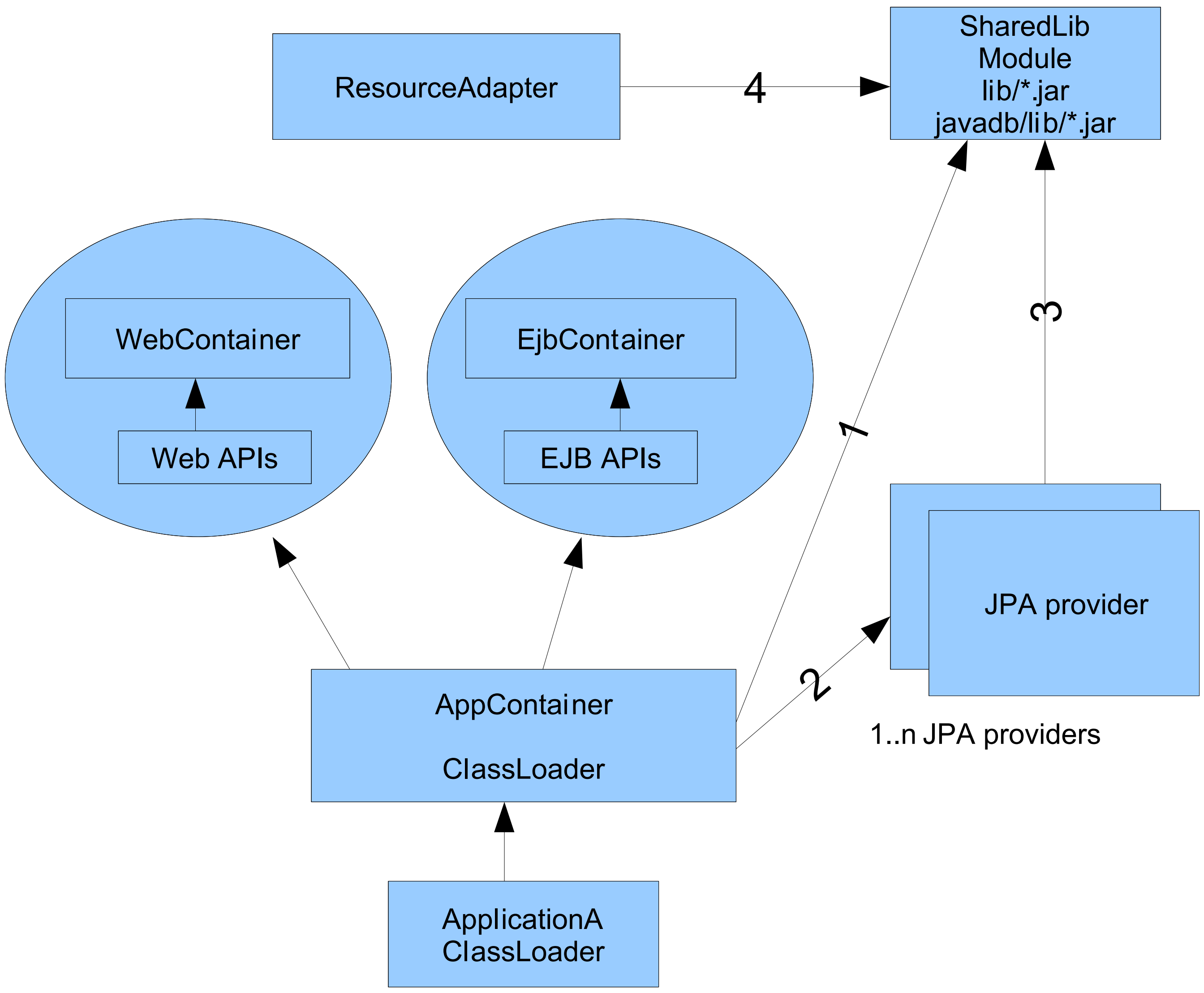
Normal HK2 service resolution should suffice to resolve the JPA provider based on the META-INF/services entries.

4.2.2 case 5

Naming service will be available though the HK2 service resolution mechanism, from which the connection pool can be looked up. After wiring up the connection pool with the JPA provider, the entity manager can be created by the deployment code and injected in the application code.

4.2.3 case 6, 7

Normal class loading delegation should be enough to support such use cases.


ResourceAdapter
4
SharedLib
Module
lib/*.jar
javadb/lib/*.jar
WebContainer
Web APIs
EjbContainer
EJB APIs
1
3
JPA provider
2
1..n JPA providers
AppContainer
ClassLoader
ApplicationA
ClassLoader

5 Container pluggability

Containers are entities that can run applications, just like the web container, the ejb container and so on. Such containers can be installed or removed from a GlassFish V3 installation, therefore there is no special code referencing the different application containers like the Switch.java in V2.

Containers have the ability to install themselves in an existing GlassFish installation with the following Services implementations :

- Sniffer : Invoked during a deployment operation (or server restart), sniffers will have the ability to recognize part (or whole) application bundles. Once a sniffer has positively identified parts of the application, its setup method will be called to have a chance to setup the container for usage during that server instance lifetime.
- ContainerProvider is called each time a container is started or stopped. A container is started once when the first application using it is deployed and will remain started until it is stopped when the last application using it was undeployed or stopped.
- Deployer is called to deploy/undeploy applications to a container
- AdminCommand : Containers can come with special set of CLI commands that should only be available once the container has been successfully installed in a GlassFish V3 installation. These commands should be used to configure the container's features.
- TBD : there will also be a AdminGUI pluggability layer but it has not been designed yet

6 Admin CLI

Like everything in V3, we rely on HK2 components to identify administrative commands. So in this case, the AdminCommand interface is annotated with @Contract, so all you have to do to add a new command to GlassFish V3 is to create a Service implementing the AdminCommand interface and the runtime will find it :

```
package com.foo.bar;

@Service(name="super")
public class MySuperCommand implements AdminCommand {
...
}
```

Now this is nice but we can do better, for instance because admin commands are Services, we can use injection, so say for instance you want to have access to the domain configuration, you just do

```
@Inject
Domain domain;
```

and the domain information will be injected in the instance field before the command execute() is invoked, that's nice but why not extending this to the command parameters. Too many times, I have seen command implementations code just ensure that all mandatory parameters to the command are present, values are correct, and extraction from the String[] representation.

This is where the new annotation @Param become useful, annotate any instance field of the command implementation with the @Param annotation to make it a command parameter. So for instance adding the following

```
@Param
String name
```

adds a name parameter to the command. Of course, parameters can be optional

```
@Param(optional=true)
String myoptionalparam
```

or the default parameter (but there can only be one of course)

```
@Param(default=true)
String path
so you can now do :
asadmin deploy --path foo.war
or
asadmin deploy foo.war
```

Also the param tag is automatically linked to the required LocalStrings.properties resource file of your classloader, so when you add the

```
com.sun.foo.bar.mysupercommand=Some random super command
com.sun.foo.bar.mysupercommand.path=path to the file the command applies
to.
```

strings to your LocalStrings.properties, the system will find them and use it for any type of error checking or help :

```
$prompt>asadmin super
FAILURE : super command requires the path parameter : path to the file the
command applies to.

$prompt>asadmin super --help
Some random super command

Parameters :
     path : path to the file the command applies to.
```

The above behaviours are completely handled by the system, the command will not be executed as long as all mandatory parameters are provided by the users. There is more to do of course, like automatic help page and localized page creation, there should be some maven 2 plugins developed to automate such pages creation and ensure that all strings are properly localized.

7 Building

Although this information will most likely change and is not entirely implemented, this is an introduction to how we will organize the source code in V3.

The bootstrap directory is not going to be moved forward but we will use a directory layout to separate and represent modules segments. Functionalities linked to packaging are moved to the distribution modules.

7.1 Source Code Management

Source code management is using CVS for now, we plan to move to Mercurial at some point. As for now, you checkout the code with the following cvs command :

```
cvs -d :pserver:<userID>@cvs.dev.java.net:/cvs co glassfish_v3
```

7.2 Modules

Modules are organized following the maven best practice pattern located at http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

Although each module can use a dedicated scm repository, it is important to follow that structure for consistency and tool support.

Complete description of a V3 module structure is located at http://wiki.glassfish.java.net/Wiki.jsp?page=V3WorkspaceStructure

but for the most simple cases, it should be the following files/directories

pom.xml	maven build file
src/main/java	module's sources
src/main/resources	module's resources like localized strings, xml...
src/tests/...	module's unit tests

7.3 Functional Areas

Modules will be grouped into functional areas and moved under specific directories. A user that wish to checkout the entire V3 workspace should do so by issuing the command described in IV.1. However for developers that wish to only work in a specific area should checkout the set of modules that are identified in that area.

The following areas have been already identified, this may evolve

glassfish_v3/web	web related technologies (jsp, jsf...)
glassfish_v3/ejb	ejb related technologies
glassfish_v3/core	v3 core infrastructures
glassfish_v3/addons	adminGUI and other tools

for instance, under glassfish_v3/web, you will find subdirectories for each module that are part of our web profile. Users only working on the jsp compiler will be able to issue the following commands :

cvs -d :pserver:<userID>@cvs.dev.java.net:/cvs co glassfish_v3/web to work on all web modules

cvs -d :pserver:<userID>@cvs.dev.java.net:/cvs co glassfish_v3/web/jspc to work on the jsp compiler module.

Once we will complete the switch to Mercurial, such tree based organization does not support checking out part of the tree so we will most likely move each sub-directories under glassfish_v3 to a new Hg repository.

7.4 Building

Basic understanding of maven and maven repositories is necessary to build V3, but at each level, users will be able to do a mvn command to build a module, a functional set of modules, or the entire V3 workspace.

mvn clean	cleans the binary outputs (.class, and .jar files) from the module target directory. Does not remove the module artifact from the local maven repository.
mvn install	builds and install the binary artifact in the local maven repository.
mvn javadoc	creates javadoc

Artifacts cannot be published directly to the global HTTP maven repositories because a certain number of integration tests need to happen. When a developer feels confident that his code is robust and has made enough integration tests on his local machine (by building a distribution and running tests), it will check in the sources. Hudson, the continuous building system, will monitor checkins and will trigger a build/test/publish cycle each time a developer checks in a change set.

7.5 Distributions

Distributions are created by assembling jars and other artifacts. There is a flexible distribution description mechanism based on maven pom file for binary dependencies and ant scripts for the other types of artifacts. More to come from Paul Sterk.

7.6 HK2

HK2 which is the java.net project for our module subsystem and lightweight component model has a separate java.net project located at http://hk2.dev.java.net

7.6.1 sources

HK2 sources are stored in a Subversion repository, so checking out sources is usually done with

```
svn checkout https://hk2.dev.java.net/hk2/trunk
```

7.6.2 Module Subsystem documentation