Deployment One-pager for GF v3

1. Introduction
    1.1. Project/Component Working Name:
        Deployment

    1.2. Name(s) and e-mail address of Document Author(s)/Supplier:
        Tim Quinn (timothy.quinn@sun.com)
        Hong Zhang (hong.zhang@sun.com)

    1.3. Date of This Document:
        Dec. 15, 2008

2. Project Summary
    2.1. Project Description:
        Extensibility interfaces relevant to deployment
        Application management (app config customization)

    2.2. Risks and Assumptions:

3. Problem Summary
    3.1. Problem Area:

Extensibility I/Fs – Container developers must be able to extend GlassFish by building new container types. These deployment-related interfaces define a service provider interface (SPI) which container developers implement so as to plug into the GlassFish deployment infrastructure that is shared by all container types.

Application management – Application developers and administrators want to be able to deploy an application, then customize configuration information that is specific to that application type. As an initial example implementation for the web app type, v3 allows users to customize the env entry values and context param values for a web app after deployment. Specifically, users can modify the values of existing env entries or context param settings, add new env entries or context params that are not present in the application's descriptor, and can suppress the effect of an env entry or context param that is present in the descriptor. (This last feature is not truly “deleting” the item from the descriptor, although the net effect is the same.)

    3.2. Justification:

Extensibility I/Fs – Greatly simplify ability to plug in new container types.

App mgt – Ease-of-use for fine-tuning app configuration after deployment without requiring the user to edit the deployment descriptors, repackage, and redeploy the application.

4. Technical Description:
     4.1. Details:

Extensibility I/Fs – The GF v3 prelude Add-on Component Development Guide documentation describes the interfaces in detail. http://docs.sun.com/app/docs/doc/820-6583/ghmon?a=view
They are summarized briefly below for convenience. Once the container developer implements these interfaces, he or she packages the Sniffer into one OSGi module and the other classes into one or more other OSGi modules. (The deployment infrastructure must be able to ask each sniffer implementation if it handles an application being deployed. The other implementation classes for the container type are not needed unless an application of that type has been deployed to the server. The sniffer should be packaged separately from the other classes so GlassFish can load it without loading the other classes related to the same container type.)

```
public interface org.glassfish.api.container.Container {
    public Class<? extends Deployer> getDeployer();
    public String getName();
}

public interface ApplicationContainer<T> {
    public T getDescriptor();
    public boolean start(ApplicationContext startupContext) throws
Exception;
    public boolean stop(ApplicationContext stopContext);
    public boolean suspend();
    public boolean resume() throws Exception;
    public ClassLoader getClassLoader();
}

public interface org.glassfish.api.container.Sniffer {
    public String getArchiveType();
    public String getDefaultApplicationName(ReadableArchive archive);
    public boolean handles(ReadableArchive archive) throws IOException;
    public ClassLoader getClassLoader(ClassLoader parent, ReadableArchive
archive);
    public void expand(ReadableArchive source, WritableArchive target)
throws IOException;
    public Manifest getManifest(ReadableArchive archive) throws
IOException;
}

public interface org.glassfish.api.container.Container.Deployer<T extends
Container, U extends ApplicationContainer> {
    public MetaData getMetaData();
    public <V> V loadMetaData(Class<V> type, DeploymentContext context);
    public boolean prepare(DeploymentContext context);
    public U load(T container, DeploymentContext context);
    public void unload(U appContainer, DeploymentContext context);
```

```
    public void clean(DeploymentContext context);
}

public interface org.glassfish.api.deployment.archive.ReadableArchive
extends Archive {

    public InputStream getEntry(String name) throws IOException;
    public boolean exists(String name) throws IOException;
    public long getEntrySize(String name);
    public void open(URI uri) throws IOException;
    public ReadableArchive getSubArchive(String name) throws IOException;
    public boolean exists();
    public boolean delete();
    public boolean renameTo(String name);
}

public interface org.glassfish.api.deployment.archive.WritableArchive
extends Archive {
    public void create(URI uri) throws IOException;
    public void closeEntry(WritableArchive subArchive) throws IOException;
    public OutputStream putNextEntry(String name) throws
java.io.IOException;
    public void closeEntry() throws IOException;
    public WritableArchive createSubArchive(String name) throws
IOException;
}

public interface org.glassfish.api.deployment.archive.ArchiveHandler {
    public String getArchiveType();
    public String getDefaultApplicationName(ReadableArchive archive);
    public boolean handles(ReadableArchive archive) throws IOException;
    public ClassLoader getClassLoader(ClassLoader parent, ReadableArchive
archive);
    public void expand(ReadableArchive source, WritableArchive target)
throws IOException;
    public Manifest getManifest(ReadableArchive archive) throws
IOException;
}
```

Application Management

Container developers can choose what, if any, configuration to open for post-deployment customization. They make this customization possible by completing these tasks:

1. Declare a Configured interface, representing the module type's customizable configuation, that extends ApplicationConfig and is particular to the container type. For example, here is definition in GlassFish v3 prelude for web app customization:

```
@Configured
public interface WebModuleConfig extends ApplicationConfig ... {
...
   @Element
   public List<EnvEntry> getEnvEntry();

   @Element
   public List<ContextParam> getContextParam();
...
}
```

The container developer also implements any lower-level interfaces referenced from this interface (EnvEntry and ContextParam in this example).

By defining these interfaces, the container developer implicitly extends the domain.xml format as well. Specifically, a container-provided interface which extends ApplicationConfig corresponds to a new child element of <application>. For example, the web container implementation in v3 prelude introduces <web-module-config> as a new optional child of <application> for those <application> elements that correspond to web apps. Because the definitions of and inheritance among the @Configured interfaces determines the domain.xml format the container developer does not need to do any additional work to expand the domain.xml format to allow for customizable configuration.

2. Implement the ApplicationContainer interface's start method to retrieve the application-type-specific customization information and use it to prepare the execution environment for the application as part of starting the application. The com.sun.enterprise.web.WebApplication class (which implements ApplicationContainer) illustrates this for the web module type.

3. If desired, create AMX interfaces specific to the new module type with convenience methods for accessing the customizable configuration for the module type. (The com.sun.appserv.management.config.WebModuleConfigConfig interface illustrates this for the web module type.) Defining these interfaces simplifies writing the GUI plug-in.

4. Implement a GUI plug-in which developers and administrators can use to customize the configuration that is exposed for the given container type. The plug-in uses AMX (either the generic access methods common to all AMX entities or the module-specific AMX interfaces defined above) to retrieve and assign values for the customizable configuration for a given application type. The plug-in should reside in its own OSGi module or one with few other classes, since the GUI will load and use the plug-in module and we want to keep the admin console footprint as small as possible.

The GUI plug-in for customizing web apps was deferred to the JavaOne release of v3.

The plug-in for customizing web apps will expose the env entries and context params from the application's descriptor, augmented by the contents of the default-web.xml and the context.xml files. The display to the user will show the value for each env-entry and context-param as the web container would compute it as if it were preparing to start the app. The user will be able to modify the value of an existing item, add a new env-entry or context-param, or mark an existing env-entry or context-param to be ignored (suppressed).

The GUI will support different customizations of a given application for different targets. Although the exact visual appearance has not been defined, users will be able to specify that a particular customization should apply to all targets or, alternatively, identify which target or targets a specify customization should apply to. Thus it will be possible for the user to customize the same env-entry (or context-param) differently for different targets.

This target-specific customization will be implemented using tokens. That is, the value of a customized configuration item will be recorded as ${token-name} and the token will be defined differently and multiple times in the configuration according to how the user defines the values through the GUI.

5. Implement a command which developers and administrators can use to customize the configuration. The Add-on Component Development Guide (http://docs.sun.com/app/docs/doc/820-6583/ghpwe?a=view) explains this process in detail.

The command(s) for customizing web apps were deferred to the JavaOne release of v3.

These commands as proposed for the v3/JavaOne release are listed below and follow the customization guidelines listed in the add-on documentation. Note that the create, delete, and set commands require the user to identify the item uniquely using the application name, the config-type, the name of the configured item, and the target of the command. Note that the --config-type option specifies which variety of configuration information is of interest, while the --type option specifies, for creating or setting env-entry items, what env-entry-type setting should be assigned. The --target option is not required; if omitted the command applies globally to the application on all targets to which it is assigned.

```
create-web-config --config-type=[env-entry | context-param] --
application=app-name --name=name --type=env-entry-type --
description=description --value=value [--target=target]

delete-web-config --application=app-name --config-type=[env-entry |
context-param] --name=name  [--target=target]

set-web-config --application=app-name --config-type=[env-entry | context-
```

param] --name=name [--value=value] [--type=env-entry-type] [--description=description] [--target=target]

list-web-config --application=app-name [--config-type=[env-entry | context-param]] [--name=name] [--target=target]

Each container can define the configuration commands that make sense for it. The planned commands listed here for the initial web configuration implementation strike a balance between brevity and clarity. For example, the --type option does not make sense if the --config-type is context-param. But the commands as proposed are simpler (shorter, certainly) than if the commands used orthogonal option names for each type of config, such as env-entry-name, param-name, env-entry-value, param-value, etc.

This also seems like a more user-friendly choice than entirely separate commands for each config item type. That is, having different create/delete/set/list commands for each of web-context-param, web-env-entry, etc. would quickly become unwieldy for users.

4.2. Bug/RFE Number(s):

Extensibility: Issue 4103 https://glassfish.dev.java.net/issues/show_bug.cgi?id=4103

App mgt: Issue 4105 https://glassfish.dev.java.net/issues/show_bug.cgi?id=4105

4.3. In Scope:
as described

4.4. Out of Scope:

4.5. Interfaces:

4.5.1 Exported Interfaces

Extensibility

The Java interfaces and the methods defined on them (see the Technical Description) are, at this point, evolving (toward the “stable” end of “evolving”). During the course of v3 prelude engineering these interfaces did change as we understood subtle aspects of extensibility and they might continue to do so through the v3 JavaOne engineering process.

Application management

For the framework affecting all post-deployment customization, regardless of app type:

The ApplicationConfig @Configured Java interface is evolving.

The Application @Configured Java interface is evolving. It is defined to permit optional ApplicationConfig children and also provides a “duck-typed” method for retrieiving ApplicationConfig children of a given type as a convenience to the ApplicationContainer implementation that needs access to customization data relevant to its container type.

For the initial implementation of web app customization:

The Java interfaces (WebModuleConfig, EnvEntry, ContextParam) are evolving.

The specific configuration items subject to customization are evolving in that further development might allow customization of more config items.

The web-specific CLI commands for creating, deleting, setting, and listing config customizations are evolving. In particular, if and when new config items become customizable the commands will evolve to accept the new config-type as well as to allow the user to specify any other information that is specific to the new config items not already handled by existing options.

The GUI itself, once created, will be an evolving interface.

The domain.xml contents, while we discourage users from directly accessing it, is in fact an exported interface. The app management feature evolves domain.xml by potentially allowing containers to add container-specific customization data as child elements below <application>. The specific changes which support web app customization indeed represent evolution in the domain.xml interface.

4.5.2 Imported interfaces

hk2 @Configured framework

4.5.3 Other interfaces (Optional)

4.6. Doc Impact:

Extensibility – The referenced doc already describes many aspects of extensibility. We need to review it (again) for completely and accuracy especially if the Java interfaces change further.

App management – The add-on development guide should be enhanced to describe the process for supporting config customization.

4.7. Admin/Config Impact:

See discussion of the web customization GUI and web customization commands.

4.8. HA Impact:
None

4.9. I18N/L10N Impact:
Web customization GUI will be subject to localization.

4.10. Packaging & Delivery:
No packaging impact. No installation impact. Upgrade from prelude to v3/JavaOne is not affected; upgrade beyond should be minimal because, even if the customizable items for web apps expands, the relevant elements in domain.xml are optional, and the corresponding changes to the @Configure interfaces will be backward compatible.

4.11. Security Impact:
No impact.

4.12. Compatibility Impact

See packaging/delivery comments.

There are no incompatible changes from v3 prelude to v3/JavaOne.

4.13. Dependencies:
Dependency on hk2 @Configured framework.

5. Reference Documents:
GF v3 prelude doc on add-on components: http://docs.sun.com/app/docs/doc/820-6583/ghmon?a=view

6. Schedule:
6.1. Projected Availability:
GlassFish v3/JavaOne release (June '09)