Glassfish V3 Admin Infrastructure To-Do

Contents

Introduction

Summarized here are outstanding engineering challenges as of July 2008.

Glassfish V3 should be understood as a clean break with the past. The appeal to outside developers and third parties is a key strategic goal that must take precedence over backward compatibility; V3 needs to break into the mainstream, creating a community willing to invest time and effort into it. The engineering effort required to maintain substantial backward-compatibility in the admin area is simply too restrictive for the ambitious GlassFish V3.

Key design principles:

- **No special cases** can be admitted in the V3 world. This is not to say that pluggable behavior cannot be offered to solve certain design problems, but that pluggable behavior must be a general feature, accessible to any module.
- **No modules are special**. A module supplied with GlassFish V3 gains no special privileges or support over modules written by third parties.
- **Backward compatibility** cannot always be accomodated. Reasons for this include the crippling of new features and the huge overhead in terms of engineering time, code complexity and testing.

Open areas

Security

Goal: Role Based Access Control (RBAC)

In the author's view, our ability to implement this without considerable effort is dubious at best for the following reasons:

- Multiple entry points: JMX, REST, internal access.
- REST access bypasses JMX, so both JMX and REST would need to implement the same security, which model to use?

Support for JDBC realm as well as file based realms might be desirable. However, the admin users are likely to be a small group, so perhaps JDBC is overkill.

Domain creation and offline configuration

Goal: create a domain.xml file

This problem can be as complex as desired. At the most basic level, a domain could be created assuming certain known services, with anything extra being a subsequent administrative action. At its most complex, it could involve creating a domain based on an extensive list of modules, perhaps those installed elsewhere, or even perhaps modules that need to be downloaded and installed.

The term "offline configuration" refers to the ability to populate configuration without starting the server. This concept made more sense in V2, but in V3 it is a bit non-sensical; "offline" would mean starting a bare-bones **distribution** that would somehow make available all @Configured interfaces for modification. How this would work for modules that hadnot yet loaded is unclear, perhaps a brute traversal of all available module. A significant issue is the "chicken and egg" problem: there must be some API to generate new configuration even if it has never existed, a sort of bootstrapping API. Is this facilty JMX-based or is it something else? Unclear.

Module installation

Goal: painless installation of modules via GUI or CLI

User should be able to:

- view all installed modules by name, version number, runtime status and dependencie
- view available module updates for each module
- uninstall a module, cascading the uinstall to any dependent modules
- install a module, cascading the install to install any required modules
- forcibly start a module to verify that it can load correctly

---

Miscellaneous infrastructure

Goal: connectivity, domain, ports,custom MBeans, etc

CLI Design Issues	
Issue	**Comments**
Admin ports	Clarify which ports can be used and white protocols for asadmin, JMX, etc.
JMXConnectors	Which JMX connectors can be used.
Restarting domain via GUI	Support restarting the domain through the GUI.
Embedded startup	Resolve remaining issues to embedding GlassFish; formalize and document the API.
	Modules should be able to load custom MBeans (monitoring, runtime, misc),

Support for custom MBeans	though not necessarily for configuration, which is handled generically. MBeans should load/unload when a module loads/unloads.
Versioning	Module writers should be encouraged to provide a "version" attribute. It is unclear whether a version on domain.xml itself makes sense, given that the content is largely driven by what (arbitrary) modules are to be installed..
Logging	Formalize and document the best practices for logging in GlassFish V3. Make it possible for logging to be emitted as MBean Notifications as in V2.

There are almost certainly a number of other small-to-medium sized issues in this area not captured above.

CLI

Goal: script-capable generic CLI interface, with support for V2-style commands while generalizing to support arbitrary modules.

CLI Design Issues	
Issue	**Comments**
Command completion and history	The suggestion has been made to use jline for history, command completion, etc. This needs to be evaluated, as well as the BSD license from a legal perspective.
Support for MBeans (optional)	It should be feasible to get and set attributes and invoke methods of MBeans; module developers should not be required to write commands to do so. While dotted names do address the get/set issue, it is less powerful than a jmxcmd-style interface. A means to specify an MBean and map conventional options to that MBean's capabilities could make it possible to generically provide command support for any MBean, in a familiar syntax.
Modified syntax	`asadmin [meta options] subcmd [options] [operands]` This syntax supports cvs/svn-style meta options that precede the subcommand. Should 'asadmin' be the command name? Why not gfadmin to sync up with the project?
User preferences	Should persist targeted servers (DAS) and remember the last one used, for user convenience.
Supported commands	Document and maintain a list of supported commands. In V3, these might be best grouped by module (or at least tagged by module), as opposed to a huge grab-bag.

Namespace support	In a pluggable world, it should be possible to disambiguate commands since different module writers could easily choose the same command names.. For example, perhaps two modules would like to implement a create command. Some sort of prefix support should be possible eg:
	
	`gf:get`
	`user:get`

Dotted names

Goal: dotted names to be available for all MBeans

There are significant changes to dotted names in GlassFish V3:

- V3 dotted names are fully general, offering access to *all* attributes of all MBeans.
- A “zero configuration” approach is used so that *any* MBean in the system becomes accessible via dotted names.
- There is no “dotted name registry”; dotted names are *always evaluated dynamically* upon request. This means zero overhead at runtime (aside from one String per MBean).
- No distinction between dotted names for monitoring vs config vs anything else.

Dotted names are partially implemented as of 27 June 2008. The following issues must be resolved and implemented:

Dotted-Names Design Issues	
Issue	**Comments**
Including/excluding attributes	Users won’t expect to see all attributes of configuration MBeans. For example, users expect that configuration MBeans will offer dotted names for the configurable attributes (only), but not for additional MBean attributes. This can be dealt with by the client (asadmin), which can call the API such that only desired attributes are visible.
Namespace	The new hierarchy for all dotted names starts with “root”, which can be omitted. There are no missing nodes as in V2 and **there are no aliases**; the dotted name is an unambiguous mapping to dotted name parts of the containment hierarchy of MBeans. It is automatic, and makes no exceptions for any MBean; a dotted name is defined purely by the MBean hierarchy and nothing else. Example: the dotted name for HttpListener "listener1” is: root.domain.configs.server-config.http-service.http-listener:listener1. Clients (eg asadmin) can choose to offer the convenience of omittingany portion of the prefix, though this involves some peril of ambiguity, and doesn’t play well with wildcarding.
Elimination of special cases	No special cases can be admitted to the V3 dotted names scheme. This results in some breakage with existing dotted names such as “property” names, which were special cased. Some of these can be dealt with by a simply textual substitution prior to using the dotted name.
Syntax change	An unambiguous grammar is needed to preclude ambiguity. This grammar (subject to change) includes provision for both named and anonymous elements as well as singleton and non-singleton elements thereof. The grammar change includes the following scoping mechanism: (1) named elements are scope with the ':' character as in http-listener:listener1 (2) anonymous elements are scoped with the subscript operator as in jvm-options[0]. (3) Attribute names are delimited by the '@' character eg root.domain.servers.server@name.
Wildcards	As with V2, the wildcard * may be used. Its behavior might or might not remain the same, because the syntax is not the same. Care should be taken to not offer client shortcuts that make wildcards ambiguous.

Configuration — reference vs scoped

Goal: eliminate global references, making the configuration model both more expressive and simpler.

The GlassFish V2 reference model is confusing, transactionally-awkward, and requires extra engineering effort at multiple levels. Worse, in practice, the reference based model is used in a **degenerate manner**; clusters and standalone servers **do not** share configuration and resources. A scoping (inheritance) model. The conceptual viewpoint of CSS-style cascading should be understood here.

In V3, the elimination of references simplifies the model and eliminates transactional headaches. The same (or greater) sharing benefits can be achieved through scoping/inheritance. Furthermore, the distinction between configuration and resources is artificial, and can safely be abandoned.

Example of configuration scoped to a cluster

Here we see that a cluster “c1” has been defined. Contained within it are:

- its configuration (`<config>`), applicable to all servers in that cluster, unless overridden by that serve;
- its resources (`<resources>`), applicable to all servers in that cluster;
- its applications, applicable to all servers in the cluster;
- its servers, each of which draws on configuration, resources and applications in the cluster.

Notice some simplification:

- the `<config>` no longer needs a name, since it is implicit by scoping.
- Servers need no references to applications; they are all implicit
- Resources can be shared, or they can conveniently exist directly within the application that needs them.
- Name-scoping is now per-cluster; other clusters can re-use the same names for configuration, resources and/or applications.

```
<clusters>
 <cluster name="c1">
  <config>
      ...all existing config stuff to be shared by all servers...
  </config>
  <resources>
      ...resources shared by all applications...
  </resources>
  <applications>
      <application name="app1"> ...
          <resource name="connection-pool" ... />
      </application>
  </applications>
  <servers>
      <server name="c1-i1" ... />
  </servers>
 </cluster>
</clusters>
```

Example of configuration scoped to a server-group

The same concept can be applied to groups of standalone servers (with one server being a basic case), which are similar to clusters, except that they are *not homogeneous*, and thus can contain server-specific applications and configuration.

Like clusters, server-groups gain the benefits of shared configuration, scoping and implicit inclusion. Note the ability to embed resources at the <server-group> or <server> or <application> level; this flexible scoping can be altered quickly via copy/paste in a text editor, with each application implicitly getting the appropriate resources via scope inheritance.

One wonders whether the cluster and server-group concepts can be merged via the use of a “homogenous” flag eg `<server-group name="cluster1" homogeneous="true">`.

```
<server-groups>
 <server-group name="loner">  ! a single standalone server
  <config>
      ...configuration shared by all servers...
  </config>
  <resources>
      ...resources shared by all applications in the group...
```

```
  </resources>
  <applications>
      <application name="app1"> ...
          <resource name="connection-pool" ... />
      </application>
  </applications>
  <servers>
      <server name="c1-i1" ... />
  </servers>
 </server-group>

 <server-group name="birds-of-a-feather">
  <config>
      ...configuration shared by all servers...
  </config>
  <resources>
      ...resources available to all applications in the group...
  </resources>
  <applications>
      <application name="app1"> ...
          <resource name="connection-pool" ... />
      </application>
  </applications>
  <servers>
      <server name="birds-of-a-feather1" ... />
          <applications>
              <application name="app1"> ...
              <resource name="connection-pool" ... />
          </applications>
          <resources>
              <resource name="connection-pool2" ... />
          </resources>
      </server>
      <server name="birds-of-a-feather2" ... />
      </server>
  </servers>
 </server-group>
</server-groups>
```

Example of domain.xml structure

To summarize, V3 configuration will eliminate the global applications and resources. This is much more flexible, and allows copy/paste to rapidly construct functional domains and/or to move applications and resources into another scope. The proposed (coarse) structure is as follows:

```
<domain name="domain1" ... >
  <!-- no applications or resources in the domain itself -->

   <server-groups>
       <server-group name="birds-of-a-feather">
           <config>...</config>
           <resources>...</resources>
           <applications>...</applications>
           <server name="bf1" ... >
               <applications>...</applications>
               <resources>...</resources>
           </server />
```

```
            <server name="bf2" ... >
                ...
            </server />
        </server-group>

        <server-group name="loner">
            <server name="the-only" ... />
                <config>...</config>
                <applications>...</applications>
                <resources>...</resources>
            </server />
        </server-group>
    </server-groups>

    <!-- clusters are homogeneous -->
    <clusters>
        <cluster name="cluster1">
            <config>...</config>
            <resources>...</resources>
            <applications>...</applications>
            <server name="bf1" ... />
            <server name="bf2" ... />
            <server name="bf3" ... />
        </cluster>
    </clusters>
</domain>
```

Configuration validation

Goal: Reject invalid configuration values

A large chunk of validation involves rejecting valuesof an inappropriate type. Examples include rejecting "xyz" or "12x" when an `Integer` is required, or "hello" when a `Boolean` is required.

The `@Configured` interface currently offers no data-type validation support, and all data types are `String`. The appropriate place to specify data type requirements is in the definition of the attribute itself; the approach of external validation makes no sense now that there is no DTD or schema (the XML being generated directly from the `@Configured` interfaces).

> The annotation `@Attribute` contains `reference()` and `variableExpansion()` booleans, which can help with validation, but basic data type validation goes unaddressed.

Tasks

- Define a scheme for annotating attribute validation requirements (see below);
- Annotate all existing `@Configured` attributes appropriately;
- Implement the code to apply the data type validation when an attribute is changed (and possibly when an attribute is loaded as a warning, in case the file was hand-edited);
- (optional) Design the `Validator` interface, and implement the code to apply the `Validator` class prior to a transaction commit

A possible approach:

The `@Attribute` annotation could be extended for basic validation (eg specifying the required class). For convenience as well as not "polluting" `@Attribute` with too many validation parameters, additional annotations could be used, with care being take to minimize conflicting requirements (eg a number whose default value is "hello"). Any data type might take on useful mnemonic values such as "auto" or "none", in addition to variable expansion `${...}` forms. So validation needs to allow mnemonic and variable forms, but for literal forms will restrict

A `Validator` could be applied on `@Configured` as a whole; it would be invoked prior to committing a transaction. Its purpose would be to enforce any relationship invariants between multiple attributes (eg min, cur, max).

```
@Configured
@Validator(className="com.mycompany.MyConfigValidator")
public interface MyConfig extends ConfigBeanProxy, Injectable {
    ...
}

public class <T extends ConfigBeanProxy> ValidationException<T> extends Exception {
    public ValidationException( String msg, T config ) { ... }
}

public interface <T extends ConfigBeanProxy> Validator<T> {
    void validate(T config) throws ValidationException<T>();
}
```

A `NumericValidator` could enforce restrictions on the range of a number. It could allow special mnemonic values, in additional to variables like `${...}`.

```
@Attribute(defaultValue="32")
@NumericValidator(minValue=1, maxValue=512, altValues ={"auto", "none"})
public String getMaxPoolSize();
```

A `BooleanValidator` specifies that the value is a `Boolean`, while allowing for special values, like “`auto`”, which might be useful for intelligent defaults when an application is running on disparate types of systems.

```
@Attribute(defaultValue="true")
@BooleanValidator(altValues={"auto"})
public String getThreaded();
```

A `CollectionValidator` could restrict allowed values to only those of a certain set. Lists of protocols are a good example of such a use-case.

```
@Attribute(defaultValue="http")
    @CollectionValidator(legalValues={"http", "https", "ftp", "default"})
    public String getProtocol();
}
```

Clusters and remote servers

Multiple areas need to be considered.

Propagating configuration to remote servers

When configuration changes, it must be propagated to the remote cluster(s) and server(s). In GlassFish V2, fairly complex

hard-coded knowledge was used to optimize delivery of only the changed elements to only those servers that needed them. For GlassFish V3, basic functionality can include propagating everything to all affected servers. In addition, an evaluate should be made of the wisdom of out-of-sync domain.xml purely to save a few K of transmitted configuration.

DETECTING/DISPLAYING WHETHER A REMOTE SERVER IS ACTUALLY USING THE CURRENT CONFIGURATION

The “restart required” functionality has been hard-coded in V2. In GlassFish V3, no assumptions can be made here. Instead, a new more flexibile and informative approach is needed. Ideally, a client such as the GUI should be able to show exactly which attribute(s) require a reboot, their current stored value, and the value currently in use.

REMOTE MBEANS

Remote MBeans of various kinds will need to be cascaded into the DAS. These include logging and monitoring Mbeans. We should re-evaluate whether the cascading service is appropriate, or whether we should implement our own more intelligent scheme.

AMX configuration

AMX configuration has traditionally involved too-intimiate knowledge of all the configuration interfaces, requiring synchronization between changes in underlying components and changes in the AMX interfaces. AMX has had to maintain the `com.sun.appserv.management.config.*` classes consistent with all underlying components. That approach is unworkable in V3. Instead, the components (modules) themselves define the configuration and the corresponding AMX interface.

WHY @CONFIGURED CANNOT BE THE AMX INTERFACE

The `@Configured` interface defined by a component cannot work as the AMX interface. For one thing, it entails all sorts of dependencies (HK2, etc) which are unsuitable for a remote, over-the-wire API. For another, navigation involves(underneath it all) using ObjectNames to find children and parents in the MBean hierarchy. While some of this is hidden by dynamic proxies, “plain” JMX clients must use ObjectNames, and data types should be restricted to standard types.

MOVING TO A GENERIC API FOR CONFIGURATION

The ideal system is one that requires no developer action to automatically expose the configuration as an MBean. The existing V3 AMX implementation comes close to this, but does not yet achieve this goal. Yet moving forward, that *must* be possible. The issue remaining is whether a “custom” AMX interface adds enough value to make the additional complexity worthwhile.

A generic API would mean the elimination of specify getter/setter methods eg `getPort()` and setPort(). Instead, clients would be required to use `getAttribute("port")` and `setAttribute("port", "123")`. The difference is really syntax only, since compile time type checking is precluded by support for variable resolution (everything is a String).

This generic approach should **not** be confused with the raw JMX `get/setAttribute(s)` methods which require an `ObjectName`; we contemplate the V2-style dynamic proxies *which hide the raw JMX behavior*. The generic interface for configuration would look something like this:

```
public interface GenericAMXConfig extends AMXConfig, AttributeResolver, ... {
    public Map<String,String> getAttributes();
}
```

One complication is that “custom” AMX interfaces implement additional interfaces such as `Container`, `NamedConfigElement`, etc. A generic interface would either need to be defined as implementing those interfaces (which would be false advertising), or the dynamic proxy would need to be created with only the actual interfaces available at runtime. In short, the client might have to do:

```
GenericAMXConfig gac = ...;
```

```
if (gac instanceof Container) {
   Container cont = (Container)gac;
}
```

While this is somewhat undesirable, it actually has some benefits. A generic interface no longer supplies any specific return type, so there is no “custom” sub-interface available without casting anyway. Second, it forces client code to not make assumptions about the capabilities of the element in advance.

Transactional creation of sub-elements

In GlassFish V2, one could create a new element with properties and system properties; this was special-cased. This continues (currently) in V3, but it makes minimal sense to support it unless it is generalized to include any sub-element, not just properties and system properties. Furthermore, since V3 offers transactions, the expectation is likely to be that such an ability would transactionally create the entire hierarchy of elements.

Variable expansion

As currently implemented, AMX deals with variable resolution using the `AttributeResolver` interface; a normal `getAttribute()` call returns the “raw” value eg `${...}` and `resolveAttribute()` (various forms) must be used to fetch the resolved value. Runtime modules have their values injected (they do not use AMX) so the issue is really about how management clients access attributes for display and/or editing. Management clients such as GUI will presumably want to display both the raw value and the resolved value for clarity.

Moving to a generic model would mean elimination of all of today's AMX interfaces. This has several advantages and disadvantages:
- no need to maintain an AMX interface for the corresponding `@Configured` interface
- no place to Javadoc the API except the `@Configured` interface (this is probably OK)
- eliminates the need to map Java method names to attribute names (eg FooBar to "foo-bar").
- somewhat less convenient
- could be an either/or solution: explicit or generic could both be supported.
- create methods would become generic (and therefore somewhat opaque)

Usage would differ from today only in syntax:

```
amx.getPort() => amx.getAttribute("port")
amx.setPort("1234") => amx.setAttribute( "port", "1234" );
amx.createHTTPListenerConfig(...) => amx.createContainee( final String
j2eeType, final Map<String,String> attrs );
```

AMX monitoring

How modules can expose monitoring data and/or MBeans. Integration with getStats() of JSR 77 MBeans.

One complication is the apparent requirement for monitoring MBeans to depend on the amx-impl module so as to achieve standard AMX MBean behavior. It might work better to provide one standard MBean implementation, along with a Delegate interface.

AMX runtime MBean (JSR 77 et al)

Support the Java EE (JSR 77) Mbeans.

Reference

TBD

Schedule

TBD