2 - GBeans Architecture

{scrollbar}

Introduction

Geronimo Beans or GBeans are entities in the Geronimo Application Server that are available for use within the Geronimo system (through the Geronimo Kernel). GBeans allow loosely coupled interactions with other entities in the system and dependency injection between modules and subsystems. GBeans are interoperable with JMX tools but are implemented separately from JMX MBeans. GBeans are able to be accessed via JMX technology through adapter classes in the geronimo-system module.

GBeans vs. MBeans

JMX has very similar concepts to GBeans. In JMX, the equivalent to GBeans is MBeans, and the equivalent to the MBeanServer is the Geronimo Kernel. The Geronimo developers created GBeans so that each part of the application server wouldn't need to rely on a full JMX stack. By separating the two, the internals of Geronimo can rely on the lighter-weight GBeans. When external applications need access to the Geronimo modules, the JMX calls get delegated to the appropriate GBean or GBean Kernel methods.

What is a GBean?

A GBean is a Java object that has a method called getGBeanInfo() that returns an instance of the GBeanInfo class. GBeanInfo has three main purposes. One is to define the methods that should be exposed to the other subsystems, the second is to define attributes and references so that they can injected at runtime and the third is to define information about the GBean so that it can be located later. Each of these will be discussed in detail below. When defining the instance of the GBeanInfo object for the GBean, GBeanInfo shouldn't be constructed directly. Instead there is a GBeanInfoBuilder class that will simplify the process (see //1 below). Below is an example of creating a GBeanInfo from the Log4jService class and then returning it in the getGBeanInfo() method:

```
Log4jService.javasolid //... static { GBeanInfoBuilder infoFactory = GBeanInfoBuilder.createStatic(Log4jService.class, "SystemLog"); //1 infoFactory.
addAttribute("configFileName", String.class, true);//2 infoFactory.addAttribute("refreshPeriodSeconds", int.class, true); infoFactory.addAttribute
("configuration", String.class, false); infoFactory.addAttribute("rootLoggerLevel", String.class, false); infoFactory.addReference("ServerInfo", ServerInfo.
class, "GBean");//3 infoFactory.addOperation("reconfigure");//4 infoFactory.addOperation("setLoggerLevel", new Class[]{String.class, String.class});
infoFactory.addOperation("getLoggerLevel", new Class[]{String.class}); infoFactory.addOperation("getLoggerEffectiveLevel", new Class[]{String.class});
infoFactory.addInterface(SystemLog.class);//5 infoFactory.setConstructor(new String[]{"configFileName", "refreshPeriodSeconds", "ServerInfo"});//6
GBEAN_INFO = infoFactory.getBeanInfo(); } public static GBeanInfo getGBeanInfo() { return GBEAN_INFO; } //...
```

There are a few important things to note from above. By convention, the GBeanInfo object is constructed in a static initializer. Also by convention, the GBeanInfo object is named GBEAN_INFO. Below is the portion of a deployment plan that is relevant to the Log4jService (found in $GERONIMO_HOME /configs/j2ee-system/src/plan/plan.xml).

```
xml <gbean name="Logger" class="org.apache.geronimo.system.logging.log4j.Log4jService"> <attribute name="configFileName">var/log/server-log4j.
properties</attribute> <!--1--> <attribute name="refreshPeriodSeconds">60</attribute> <reference name="ServerInfo"> <name>ServerInfo</name> <
/reference> </gbean>
```

The deployment plan allows the injection of dependencies, attributes etc and maintains a loose coupling between the systems. The Geronimo Kernel will (using the deployment plan) inject these dependencies automatically when the GBean is loaded. Below the code samples are explained in more detail.

Attributes

In the above snippet of code //2 points to an adding of an attribute to a GBean. It defines the name of the attribute (in this case "configFileName"), it's type (String) and whether or not it's persistent (true). Now that the attribute is defined in the GBeanInfo, that attribute can be injected via the Geronimo Kernel. To inject a value into this GBean, it needs to be specified in the deployment plan. In comment one in the XML above, it says to assign the string "var/log /server-log4j.properties" to the attribute configFileName. When the kernel reads this in, it will call the setConfigFileName() method and pass in the string as the parameter.

References

In comment 3 in the Java code above, a reference is being added to the GBeanInfo object. A reference is like an attribute in that it is injected at runtime, however a reference is another GBean rather than a String or an integer. At runtime the Geronimo kernel will locate (in this case) the ServerInfo GBean and inject it into the Log4jService GBean.

Executing Methods

GBean Methods can be executed two ways. One way is the one listed above by comment 4. This defines an "operation" in the GBean. Through the kernel, that method can be reflectively invoked. Another way to invoke a method is to retrieve the GBean out of the kernel and then just execute it directly. One important point about invoking methods reflectively, is that it can be slower. The internals of Geronimo use this reflective method invocation, so performance is critical. To help improve performance the Geronimo developers created the *Raw Invoker*. The *Raw Invoker* essentially uses a cached copy of the CGLib (a higher performance reflection library) reflective invocations. When the method is invoked, the *Raw Invoker* containing the cached CGLib method call is used, saving some execution time.

Adding Interfaces

Comment number 5 above adds the interface SystemLog.class to the GBeanInfo object. Adding an interface like above is basically a shorter way of adding everything in the interface individually. All of the variables in the interface will be added as attributes and all of the methods will be added as operations.

Constructors

Comment 6 above adds a constructor to the GBeanInfo object. Since the Log4jService GBean does not have a default (no argument) constructor, the constructor must be defined. When the kernel creates an instance of the GBean, comment 6 lets it know what to pass to the constructor. Note that each of the elements in the array are attributes/references that are defined in the GBeanInfo.

GBeanLifecycle

Although implementing any interfaces for a GBean is not required it is often useful to be able to execute code when a GBean starts up, or shuts down. Implementing the GBeanLifecycle interface will allow the GBean to be notified on startup, shutdown and failure. The required methods to implement in the GBeanLifecycle interface are: doStart(), doStop() and doFail(). Below is some example code from the same Log4jService GBean:

```
Log4jService.javasolid public void doStart() { LogFactory logFactory = LogFactory.getFactory(); if (logFactory instanceof GeronimoLogFactory) {
synchronized (this) { timer = new Timer(true); // Periodically check the configuration file schedule(); // Make sure the root Logger has loaded Logger logger
= LogManager.getRootLogger(); reconfigure(); File file = resolveConfigurationFile(); if (file != null) { lastChanged = file.lastModified(); } logEnvInfo(logger); }
// Change all of the loggers over to use log4j GeronimoLogFactory geronimoLogFactory = (GeronimoLogFactory) logFactory; synchronized
(geronimoLogFactory) { if (!(geronimoLogFactory.getLogFactory() instanceof CachingLog4jLogFactory)) { geronimoLogFactory.setLogFactory(new
CachingLog4jLogFactory()); } } } synchronized (this) { running = true; } } public synchronized void doStop() { running = false; if (monitor != null) { monitor.
cancel(); monitor = null; } if (timer != null) { timer.cancel(); timer = null; } } public void doFail() { doStop(); }
```

Understanding all of the details around the above code is not necessary, it's sufficient to just understand that a Timer, or what could be thought of as a sleeping thread, is created in doStart(). The person writing the code wanted to ensure that the thread would be stopped when the GBean was stopped. When the GBean is stopped, the doStop() method will be called and then the Timer can be stopped safely. Note also that in the event of a failure, the doStop() method is also called, safely stopping the Timer.