Developing and Deploying a Geronimo GBean

{scrollbar}

Geronimo is a system framework that can be used to build a variety of tailored infrastructure services, with **GBean** providing it with a loosely-coupled and configurable runtime environments. GBean is an implementation of Inversion of Control (IoC), or the dependency injection, which allows the automatic injection of references as they become available. This tutorial is organized as follows:

Geronimo Kernel and GBean

Geronimo kernel

The Geronimo kernel is a framework for kernel services, and controls the basic server components with the following services:

- Component configuration
- Component repository
- Dependency management
- Lifecycle management

The basic server componenets, namely **security**, **logging**, **transactions**, and **naming**, work with the kernel services to build Java EE server components, such as Tomcat, Jetty, and OpenEJB. All these components, cooperate with the kernel to support Java EE applications.

GBeans

Almost everything in Geronimo is a GBean: containers, connectors, adapters, applications and so on. A GBean is an atomic unit in Geronimo, consisting of the implementation classes (.jar) and a *plan*.

The *plan* is a configuration file through which a GBean relies on the kernel for IoC and dependency injection at runtime. Dependencies on other GBeans can be configured through the attributes and references in plans. Geronimo's configuration builders construct GBeans out of these plans, group them as *configurations*, and store them in the *configuration store*.

A *configuration* is a logically coupled collection of one or more GBeans and a classloader. The configurations are loaded with the server. The *configuration store* is the standard Geronimo storage mechanism.

Applications that run in a Java EE container, described by custom *deployment descriptors* and Java classes, will be parsed and converted into GBeans by the configuration builders of the Geronimo **Deployer**. These GBeans are also stored in the configuration store.

GBean Lifecycle

A GBean can be in any of the three states: stored, loaded, or running.

- Stored: The GBean exists in a plan, or configuration store.
- Loaded: The GBean is mapped to a non-persistent name when loaded by the kernel, allowing multiple instances of the same GBean to exist.
- Running: The GBean is in the running state, coordinating the application classes that it contains.

To participate in the life cycle operations, GBeans implement a *GBeanLifecycle* interface. This interface defines three methods, `doStart()`, `doStop()` and `doFail()`.

- The `doStart()` method is used to start the GBean, and implement the GBean operations. It should not be used when the GBean is in a failed state.
- The `doFail()` method is executed when the GBean fails to start, run or stop. It marks the GBean as failed, and always calls doStop() to stop it.
- The `doStop()` menthods stops the GBean. java package org.apache.geronimo.gbean; public interface GBeanLifecycle{ void doStart() throws Exception; void doStop() throws Exception; void doFail(); } If the optional interface GBeanLifecycle is implemented, the implementation will get lifecycle callbacks from the kernel.

GBean methods

The class for a GBean should implement specific methods to support each feature of the GBean:

- **GBeanInfo**: A GBean has a method called getGBeanInfo() that returns an instance of the GBeanInfo class. GBeanInfo defines the methods that should be exposed to the other subsystems, exposes attributes and references so that they can be injected at runtime, and define information about the GBean so that it can be located later. The GBeanInfo object is constructed in a static initializer.
- **Constructor**: The GBean constructor arguments should correspond to either attributes or references for the GBean. It is defined by the **setConstructor** method in GBeanInfo.
- **Attributes**: An attribute should have a setter method, a getter method, or both. The **addAttribute** method in GBeanInfo should define the name of the attribute, its type and whether or not it's persistent, so that the attribute can be injected via the Geronimo kernel.
- **Operations**: Any public method on the GBean that is not a getter or setter can be an operation. The operation methods should match the name and argument types listed by **addOperation** for the operations in the GBeanInfo.
- **References**: A reference, like an attribute, is injected at runtime. However, a reference is another GBean rather than a String or an integer.
- **Interfaces**: Adding a reference in GBeanInfo is a faster way of adding attributes or operations separately. All of the variables in the interface will be added as attributes and all of the methods will be added as operations.

Developing a GBean

For developing a GBean, you at least have to go through the following steps:

1. Write a POJO class
2. Adding the GBeanInfo part to expose attributes,methods, interfaces, and constructors
3. Implement GbeanLifecycle inteface to use the Gbean Kernel lifecycle callback (**Optional**)

A sample GBean

The sample GBean SimpleServerGBean simply starts a socket on the Geronimo server, and retrieves the name of the GBean and the socket port. The following code snippet implements the Plain Old Java Object (POJO) section:

javaSimpleServerGBean POJO public class SimpleServerGBean implements GBeanLifecycle, InterfaceNamed{ private final String gbeanName; private int port; private boolean started = false; private ServerSocket serversocket; public SimpleServerGBean(String gbeanName, int port) { this.gbeanName= gbeanName; this.port = port; } public String getName() { return this.gbeanName; } public boolean isStarted() { return started; } private void printConsoleLog (String log) { System.out.println(" LOG : " + log); }

In the following code snippet, the GBean exposes its attributes, operations, interfaces and constructor for the GBEAN_INFO static initializer.

javaSimpleServerGBean GBeanInfo private static final GBeanInfo GBEAN_INFO; static { GBeanInfoBuilder infoFactory = new GBeanInfoBuilder(SimpleServerGBean.class.getName(), SimpleServerGBean.class); infoFactory.addAttribute("gbeanName", String.class, false); infoFactory.addAttribute ("port", int.class, true); infoFactory.addOperation("isStarted", "String"); infoFactory.addInterface(InterfaceNamed.class); infoFactory.setConstructor(new String[] { "gbeanName", "port" }); GBEAN_INFO = infoFactory.getBeanInfo(); } public static GBeanInfo getGBeanInfo() { return GBEAN_INFO; }

The SimpleServerGBean Gbean is simple to start up and shutdown. During startup, it simply accepts an incoming socket connection requests, and sends out the echo message. When being stopped, the GBean closes the resouces that it consumes.

javaSimpleServerGBean Lifecycle public void doFail() { started = false; printConsoleLog("GBean " + gbeanName+ " failed"); } public void doStart() throws Exception { serversocket = new ServerSocket(port); started = true; Thread simpleServerThread = new Thread(new Runnable() { Socket socket; InputStream is; OutputStream os; public void run() { while (started) { try { socket = serversocket.accept(); is = socket.getInputStream(); os = socket. getOutputStream(); BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(is)); String responseMessage = "simpleServer response :" + bufferedReader.readLine(); os.write(responseMessage.getBytes()); bufferedReader.close(); if (os != null) { os.close(); } if (socket != null && ! socket.isClosed()) { socket.close(); } } catch (Exception e) { //ingore } } } }); simpleServerThread.start(); printConsoleLog("GBean " + gbeanName + " started and it's listening on port:" + port); } public void doStop() throws Exception { started = false; serversocket.close(); printConsoleLog("GBean " + gbeanName+ " stopped"); } }

For this sample, we still need a class to test the SimpleServerGBean. This section creates a socket and sends a connection request to the SimpleServerGBean. After the connection has been established, it sends out messages and retrieves the echo message from the simpleServerGBean server instance.

SimpleServerTest package org.apache.geronimo.sample; import java.io.BufferedReader; import java.io.InputStream; import java.io.InputStreamReader; import java.io.OutputStream; import java.net.InetSocketAddress; import java.net.Socket; import org.junit.Ignore; import org.junit.Test; public class SimpleServerTest { @Test @Ignore public void connectToSimpleServer() throws Exception { Socket socket = new Socket(); socket.connect(new InetSocketAddress("127.0.0.1", 7777)); OutputStream os = socket.getOutputStream(); String requestMessage = "Hello simple server"; os.write (requestMessage.getBytes()); os.write("\n".getBytes()); os.flush(); InputStream is = socket.getInputStream(); BufferedReader bufferReader = new BufferedReader(new InputStreamReader(is)); String responseMessage = bufferReader.readLine(); System.out.println(responseMessage); socket.close(); } }

Before deploying your GBean, you still have to compile the GBean class and build a proper .jar file. Maven is used to build Geronimo, so it can be used to build your GBean project. You have to write a `pom.xml` file. See Maven website for more information about the file.

xmlpom.xml <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"> <modelVersion>4.0.0</modelVersion> <groupId>org.apache.geronimo. sample</groupId> <artifactId>simpleServer</artifactId> <packaging>jar</packaging> <version>1.0</version> <name>sample GBean as a simple server< /name> <dependencies> <dependency> <groupId>org.apache.geronimo.framework</groupId> <artifactId>geronimo-kernel</artifactId> <version>2.1.3< /version> </dependency> <dependency> <groupId>junit</groupId> <artifactId>junit</artifactId> <version>4.1</version> <scope>test</scope> < /dependency> </dependencies> <build> <plugins> <plugin> <artifactId>maven-compiler-plugin</artifactId> <configuration> <source>1.5</source> <target>1.5</target> </configuration> </plugin> </plugins> </build> </project>

In the POM file above,

- The only dependency you need to develop a GBean is xml <dependency> <groupId>org.apache.geronimo.framework</groupId> <artifactId>geronimo-kernel</artifactId> <version>2.1.3</version> </dependency>
- We add following section to the POM only for the sake of the testing code. xml <dependency> <groupId>junit</groupId> <artifactId>junit< /artifactId> <version>4.1</version> <scope>test</scope> </dependency> </dependencies> <build> <plugins> <plugin> <artifactId>maven-compiler-plugin</artifactId> <configuration> <source>1.5</source> <target>1.5</target> </configuration> </plugin> </plugins> </build>

Deploying the GBean

Follow these steps to deploy your GBean to Geronimo.

1. Deploy the GBean package. There are two methods to deploy a GBean package to Geronimo.
    - Deploy the .jar to the server via the **JAR repository** portlet in Geronimo administrative console. For example, you can deploy the SimpleServerGBean with the following information:

        - **GroupId**: org.apache.geronimo.sample
        - **ArtifactId**: simpleServer
        - **Version**: 1.0
        - **Type**: jar
    - Mannually copy the jar file to the GERONIMO_HOME/repository with a path and package name rule.
        - Path : `GroupId/ArtifactId/Version/ArtifactId-Version.Type`
        - Sample: `<GERONIMO_HOME>/repository/org/apache/geronimo/sample/simpleServer/1.0/simpleServer-1.0.jar`
          Note: If the GBean depends on an external library, you have to deploy both .jars into the repository, and list both of them in the <dependencies> section of the deployment plan described below.
2. The next step is to develop a GBean deployment plan. A deployment plan is to define a Geronimo *module* as an .xml file that includes the descriptions of one or more instances of exsiting GBeans. xmlsimpleServer_deployment_plan.xml <?xml version="1.0" encoding="UTF-8"?> <module xmlns="http://geronimo.apache.org/xml/ns/deployment-1.2"> <environment> <moduleId> <groupId>org.apache.geronimo.sample< /groupId> <artifactId>simpleServerModule</artifactId> <version>1.0</version> <type>car</type> </moduleId> <dependencies> <dependency> <groupId>org.apache.geronimo.sample</groupId> <artifactId>simpleServer</artifactId> <version>1.0</version> </dependency> < /dependencies> <hidden-classes /> <non-overridable-classes /> </environment> <gbean name="echoserver" class="org.apache.geronimo. sample.SimpleServerGBean"> <attribute name="port">7777</attribute> <attribute name="gbeanName">simpleServer</attribute> </gbean> < /module> Let's briefly go through this plan.
    - The **<moduleId>** element is used to provide the configuration name for the module as deployed in the Geronimo server. It contains elements for the **groupId**, **artifactId**, **version** and **type**. Module IDs are normally printed with slashes between the four components, such as `GroupID/ArtifactID/Version/Type`. In this example, The module ID will be `org.apache.geronimo.sample /simpleServer/1.0/car`.
    - The **<dependencies>** element is used to provide the configurations on which the module is dependent upon. In this example, the module is dependent on `org.apache.geronimo.sample/simpleServer/1.0`.
3. Deploy the deployment plan to the server. There are two methods to deploy a GBean Module with a plan to Geronimo.
    - Using the **Deploy New** portlet in the administrative console.
    - Using the deployer tools or GShell command.
4. If success, you can visit http://localhost:7777 and will get an echo from the server.

Note: If a GBean implementation (possibly a .jar) already exisits in the Geronimo server, you only need to develop a deployment plan and deploy it to the server for the GBean instance to work.

Referencing another GBean

Other GBeans can be used in a GBean by injecting **Reference** methods. For instance, the GBean *SimpleServerGBean* can make use of another GBean named *EchoMessageGBean* with its POJO class and GBeanInfo implemented like the following:

javaEchoMessageGBean package org.apache.geronimo.sample; import org.apache.geronimo.gbean.GBeanInfo; import org.apache.geronimo.gbean. GBeanInfoBuilder; public class EchoMessageGBean implements EchoMessageInterface { private final String msg; public EchoMessageGBean(String name) { this.msg = name; } private static final GBeanInfo GBEAN_INFO; static { GBeanInfoBuilder infoFactory = new GBeanInfoBuilder(EchoMessageGBean.class); infoFactory.addAttribute("msg", String.class, false); infoFactory.addInterface(EchoMessageInterface.class); infoFactory. setConstructor(new String[] { "msg" }); GBEAN_INFO = infoFactory.getBeanInfo(); } public static GBeanInfo getGBeanInfo() { return GBEAN_INFO; } public String getEchoMessage() { return msg; } }

This EchoMessageGBean does not implement GbeanLifecycle for simplicity. It only exposes an interface *EchoMessageInterface* which retrieves predefined echo messages.

To reference the *EchoMessageGBean* in SimpleServerGBean, the following steps have to be taken:

1. SimpleServerGBean now has more private members and a different constructor: javaExcerpt from SimpleServerGBean POJO private final String gbeanName; private int port; /* A new private member */ private EchoMessageInterface echo; private boolean started = false; private ServerSocket serversocket; /* Constructor changed */ public SimpleServerGBean(String gbeanName, int port, EchoMessageInterface echomsg) { this.gbeanName= gbeanName; this.port = port; this.echo= echomsg; }
2. In doStart() of the SimpleServerGBean implementation, we now retrieves messages from EchoMessageGBean and reorganizes the reponse message: javaExcerpt from SimpleServerGBean doStart() String appendMessage = echo.getEchoMessage(); String responseMessage = appendMessage + bufferedReader.readLine();
3. Add a **Reference** method to GBeanInfo: javaSimpleServerGBean GBeanInfo private static final GBeanInfo GBEAN_INFO; static { GBeanInfoBuilder infoFactory = new GBeanInfoBuilder(SimpleServerGBean.class.getName(), SimpleServerGBean.class); infoFactory. addAttribute("gbeanName", String.class, false); infoFactory.addAttribute("port", int.class, true); infoFactory.addOperation("isStarted", "String"); infoFactory.addInterface(InterfaceNamed.class); /* Add a reference */ infoFactory.addReference("echoMessenger",EchoMessageInterface.class); /* A different constructor */ infoFactory.setConstructor(new String[] { "gbeanName", "port", "echoMessenger"}); GBEAN_INFO = infoFactory. getBeanInfo(); }
4. Correspondingly, the dployment plan for SimpleServerGBean has to be modified. xmlExcerpt from simpleServer_deployment_plan.xml <gbean name="echoserver" class="org.apache.geronimo.sample.SimpleServerGBean"> <attribute name="port">7777</attribute> <attribute name=" gbeanName">simpleServer</attribute> <reference name="echoMessenger"> <name>msgAppender</name> </reference> </gbean> <gbean name="msgAppender" class="org.apache.geronimo.sample.EchoMessageGBean"> <attribute gbeanName="msg">[Echo from server]:< /attribute> </gbean> After we add the .jar file to the server, and deploy the plan, the test class which initiates a socket on port 7777 will get a reponse message from the server.
   true[Echo from server]: Hello simple server

The GBean SimpleServerGBean can reference different instances of EchoMessageGBean. For instance, the SimpleServerGBean listens on port 7778 instead, and gets messages from the instance **msgAppender2**.

xmlExcerpt from simpleServer_deployment_plan2.xml <gbean name="echoserver" class="org.apache.geronimo.sample.SimpleServerGBean"> <attribute name="port">7778</attribute> <attribute name="gbeanName">simpleServer</attribute> <reference name="echoMessenger"> <!-- Reference another

instance of EchoMessageGBean --> <name>msgAppender2</name> </reference> </gbean> <gbean name="msgAppender" class="org.apache.geronimo.sample.EchoMessageGBean"> <attribute name="msg">[Echo from server]:</attribute> <!--Another instance of EchoMessageGBean --> <gbean name="msgAppender2" class="org.apache.geronimo.sample.EchoMessageGBean"> <attribute name="msg">[Echo from server 2]:</attribute> </gbean>

When we deploy this plan (The .jar file does not have to be changed), the test class which starts a socket on port 7778 will gets the reponse from server 2. true[Echo from server 2]: Hello simple server

Geronimo Bootstrap

When Geronimo starts, the `<GERONIMO_HOME>/bin/server.jar` is executed with `org.apache.geronimo.cli.daemon.DaemonCLI` as the main class. This will boot the Geronimo kernel, and load initial GBeans from module `j2ee-system-2.1.3.car` into the kenel.