Geronimo Quartz Plugins

Table of Contents

1. Introduction

The Geronimo Quartz plugins integrate the Quartz scheduler into the Apache Geronimo application server. This provides more advanced scheduling features than the J2EE timer service provides, supporting things like cron expressions, scheduling based on dates and holidays, etc.

There are three plugins available, which build on each other to provide different levels of integration with Geronimo:

- Embedded Scheduler
- Deployable Jobs
- Quartz Console

This article describes the features provided by these plugins

1.1. Target Use Cases

These are the problems solved by the Geronimo Quartz Plugins:

Embedded Scheduler only

- Simple access to a Quartz Scheduler from applications
- Quartz scheduler uses Geronimo resources (threads, etc.)

Embedded Scheduler & Deployable Jobs

- Projects including a moderate number of scheduled jobs
    - Including things like a daily import/export, weekly/monthly reports, nightly maintenance, etc.

- Scheduled jobs are closely tied to a project
  - Jobs use same database, session beans, etc. as the application
- Jobs developed during the course of the project
  - Jobs are deployed or redeploy during development, as implementation classes are revised

Embedded Scheduler & Deployable Jobs & Quartz Console

- Job management via Geronimo console
  - Edit schedule and resources, start/stop, run immediately, etc.

1.2. Not Target Use Cases

This integration package is not intended to support large-scale scheduling systems including hundreds or thousands of jobs across many projects or applications. It does not replace a dedicated scheduling tool with extensive management features (such as AutoSys, or a standalone Quartz scheduler with a dedicated Quartz management tool).

Instead, it is primarily intended to support a smaller number of jobs more closely associated with individual Geronimo applications.

2. About the Geronimo Quartz Plugins

The Quartz integration is available as three components: the embedded scheduler, the deployable jobs, and the Quartz console. These components build on each other, so you can use the embedded scheduler alone, the scheduler with deployable jobs, or all three components together.

If you use the **embedded scheduler** alone, this initializes a Quartz `Scheduler` integrated into the Geronimo environment. Other applications or components running in Geronimo can access this `Scheduler` by using a GBean reference, or by asking the Quartz `SchedulerFactory` for available schedulers. To deploy and manage jobs in this scenario, you'll normally just deal with the Quartz `Scheduler`.

The **deployable jobs** plugin lets you deploy single jobs or groups of jobs as Geronimo deployment components. There is no difference to the code for a job, this just provides the packaging allowing you to create a JAR with the code for the jobs and the schedule for the jobs and then deploy, start, stop, and undeploy the job using the standard Geronimo deployment tools. In addition, jobs deployed in this way can be configured with access to Geronimo resources (database pools, JMS connections, JavaMail sessions, etc.), EJBs deployed in Geronimo, or other Geronimo components (GBeans). These resources are handled via dependency injection, so the job class must include a setter method (e.g. `setMyDatabasePool(DataSource ds)`) and that setter will be called before the job is executed.

The **Quartz console** plugin provides access to any jobs deployed using the deployable jobs plugin. It lets you view the job schedule (including previous and next execution times), start and stop these jobs, change the schedule for a job, run a job immediately, change any resources assigned to the job, etc. The console plugin does not show jobs deployed using native `Scheduler` access, only jobs deployed using the deployable job format.

3. Installing the Geronimo Quartz Plugins

The Geronimo Quartz plugins are easiest to install through the Geronimo admin console, though they may also be installed from the command-line (particularly for Little G, which does not presently include the admin console).

Tip

For either installation style, it is only necessary to select one of the plugins for installation. If you select the deployable jobs plugin it will automatically install the embedded scheduler plugin, while if you select the Quartz console plugin, it will automatically install both of the other plugins.

Console Installation Procedure

1. Log in to the console
2. Select the Plugins - Create/Install option in the left navigation bar
3. Select the plugin repository `http://www.geronimoplugins.com/repository/geronimo-1.1` from the repository drop-down list (use the Update Repository List command if necessary to populate the repository list).
4. Click Search for Plugins
5. Select the following plugins from the list to install the different features described here:
   - Embedded Scheduler: **Quartz Scheduler Integration**
   - Deployable Jobs: **Quartz Job Deployer**
   - Quartz Console: *(not yet released)*
6. For each selected plugin, select Install Plugin followed by Start (plugin ID) once the installation completes.

Command-Line Installation Procedure

1. Download the plugin files from the following pages:
   - Embedded Scheduler: http://www.geronimoplugins.com/entry_QuartzSchedulerIntegration.php
   - Deployable Jobs: http://www.geronimoplugins.com/entry_QuartzJobDeployer.php
   - Quartz Console: *(not yet released)*
2. Use the command-line deploy tool to install each plugin:

```
java -jar bin/deployer.jar install-plugin [plugin_file]
```

4. Usage Examples

This section includes sample code and configuration files for using the Geronimo Quartz plugins.

The example job discussed in this section:

- Reads a file
- Inserts data from the file into a database
- Sends an e-mail notifying someone that the data was received.

4.1. Embedded Scheduler Example

Using the embedded scheduler, you can write a standard Quartz job like this. Note that the job must create its own connection to the database and its own JavaMail session.

Job Code

```
public class FileDBJob implements Job {
    public void execute(JobExecutionContext context)
                            throws JobExecutionException {
        // Locate input
        File source = new File(...);
        if(!source.exists()) {
            return;
        }

        // Write to database
        Connection con = DriverManager.getConnection("url",
                                        "user", "password");
        PreparedStatement ps = con.prepareStatement(...);
        ...
        con.close();

        // Send e-mail
        Properties props = new Properties();
        props.put("mail.smtp.host", smtpServer);
        ...
        Session session = Session.getDefaultInstance(props, null);
        Message msg = new MimeMessage(session);
        ...
        Transport.send(msg);
    }
}
```

To deploy the job, you need to:

1. Add the job classes to the component that will schedule the job
2. Access the embedded `Scheduler`
3. Schedule the job by calling the `Scheduler.scheduleJob` method

The code to do the last two steps looks like this. It could be executed in a startup servlet, or on demand by a web or EJB component, etc.

Component that Schedules a Job

```
Context ctx = new InitialContext();
```

```
QuartzScheduler gbean = ctx.lookup("java:comp/env/Scheduler");
Scheduler scheduler = gbean.getNativeScheduler();
JobDetail detail = new JobDetail("name", "group", MyJob.class);
Trigger trigger = ...; // set the schedule for the job
scheduler.scheduleJob(detail, trigger);
```

In order for the JNDI lookup to access the scheduler GBean, the component should include a gbean-ref element in its deployment plan (e.g. geronimo-web.xml or openejb-jar.xml):

GBean Reference in Geronimo Plan

```
<gbean-ref>
  <ref-name>Scheduler</ref-name>
  <ref-type>org.gplugins.quartz.QuartzScheduler</ref-type>
  <pattern>
    <name>QuartzScheduler</name>
  </pattern>
</gbean-ref>
```

Note that the `ref-name` defines the JNDI location where the `QuartzScheduler` GBean will be bound (`java:comp/env/` plus the ref-name value -- or `java:comp/env/Scheduler` in this case, which is where the sample code above looks it up).

4.1.1. Embedded Scheduler FAQ

Q: Is running Quartz in Geronimo this way any different from running Quartz standalone?

A: Not really. The plugin uses a Geronimo thread pool instead of the native Quartz thread pool by default. But the jobs are the same and the same configuration options are available.

Q: How do I customize the scheduler configuration?

A: Once you've installed the plugin, an entry for the `quartz-scheduler` module will be written to `var/config/config.xml`. You can stop Geronimo and edit the Quartz settings in that file. Look at the value for the `quartzProperties` attribute, which is in the standard Java Properties file syntax. You can add any Quartz properties you like there, and they will override the Quartz defaults.

Q: Can I schedule or delete jobs as part of a transaction?

A: Probably, but we haven't tried this before. We'd be happy to help you get this running if you have occasion to try.

4.2. Deployable Jobs Example

With the deployable jobs approach, the sample job can be packaged in a JAR on its own, with its classes and schedule, and can be configured to access a database pool and mail session already deployed in Geronimo (rather than manually configuring connectivity in the job). The same job would look like this:

Job Code

```
public class FileDBJob implements Job {
    private DataSource dataSource;
    private Session mailSession;

    public void setDatabase(DataSource ds) {
        dataSource = ds;
    }
```

```
    public void setMailSession(Session s) {
        mailSession = s;
    }

    public void execute(JobExecutionContext context)
                            throws JobExecutionException {
        // Locate input
        File source = new File(...);
        if(!source.exists()) {
            return;
        }

        // Write to database
        Connection con = dataSource.getConnection();
        ...
        con.close();

        // Send e-mail
        Message msg = new MimeMessage(mailSession);
        ...
        Transport.send(msg);
    }
}
```

Notice that something must call setDatabase and setMailSession on the job before it can be executed. We'll configure the job to use a particular database pool and mail session already configured in Geronimo, and the plugin will ensure that those properties are injected into the job before running the job. The job code itself doesn't need any particular settings (database connection settings or mail server settings), which means not only are those settings more secure, but also the job won't need to change as frequently and will be more portable across different server instances.

To configure the job schedule and resources, an XML deployment plan must be included with the JAR containing the job classes. It can be packaged into the JAR at the location `META-INF/geronimo-quartz.xml`, or provided as a separate argument in addition to the JAR when deploying the job.

geronimo-quartz.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<jobs xmlns="http://geronimo.apache.org/xml/ns/plugins/quartz-0.2">
    <environment xmlns="http://geronimo.apache.org/xml/ns/deployment-1.1">
        <moduleId>
            <artifactId>FileDBJob</artifactId>
        </moduleId>
        <dependencies>
            <dependency>
                 <groupId>example</groupId>
                 <artifactId>javamail-server</artifactId>
            </dependency>
            <dependency>
                 <groupId>geronimo</groupId>
                 <artifactId>system-database</artifactId>
             </dependency>
        </dependencies>
    </environment>
    <job>
        <job-name>Load File into Database</job-name>
        <job-class>com.example.FileDBJob</job-class>
        <cron-expression>0 0 23 * * ?</cron-expression>
        <resource-ref>
            <property>Database</property>
```

```
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
            <res-sharing-scope>Shareable</res-sharing-scope>
            <pattern>
                <name>SystemDatasource</name>
            </pattern>
        </resource-ref>
        <resource-ref>
            <property>MailSession</property>
            <res-type>javax.mail.Session</res-type>
            <res-auth>Container</res-auth>
            <res-sharing-scope>Shareable</res-sharing-scope>
            <pattern>
                <name>MailServer</name>
            </pattern>
        </resource-ref>
    </job>
</jobs>
```

This looks similar to other Geronimo deployment plans, except the meaty part defines Quartz jobs instead of servlets or EJBs. Some important parts to note include:

- The `moduleId` element defines the unique identifier for the deployed job. This name will be used as an argument to the deploy tool to start, stop, or undeploy the job, for example. In this case it is set to **FileDBJob**.
- The `dependencies` element lists other modules that the job depends on. In this case, it depends on the modules containing the JavaMail configuration and the database pool configuration -- the two resources in the server that the job requires.
- Within a job definition, the `job-name` sets the name that will be used to identify this job to Quartz. In this case, it is **Load File into Database**.
- The `job-class` defines the fully-qualified Java class name of the job. This job class should be included in the job JAR. In this case it is **com.example.FileDBJob**.
- The cron-expression defines a cron-based schedule for the job. In this case, the expression "**0 0 23 * * ?**" means that the job should run every day at 11 PM.
- A job may include `ejb-ref`, `resource-ref`, and `gbean-ref` elements to inject various components into the job. These look similar to the same elements in J2EE and Geronimo deployment plans, except instead of a JNDI location used to identify them, they use a `property` -- the name of the JavaBean property that holds the setting in question. Here the properties **Database** and **MailSession** mean the resources will be assigned by calling `setDatabase` and `setMailSession` respectively.
    - The pattern/name elements point to the resources by name, in the modules listed as dependencies in the dependency section. So in this case, we're looking for a JDBC **DataSource** called **SystemDatasource** and a JavaMail **Session** called **MailServer** in the parent modules named **javamail-server** and **system-database**.

If both the job class and deployment plan are packaged into a JAR, the JAR contents might look like this:

```
> jar -tf file-db-job.jar
META-INF/
META-INF/MANIFEST.MF
META-INF/geronimo-quartz.xml
com/
```

```
com/example/
com/example/FileDBJob.class
```

4.2.1. Deployable Jobs FAQ

Q: What's wrong with just accessing the scheduler directly to deploy jobs?

A: Nothing's "wrong" with that, but there are some advantages to the deployable jobs approach. If you manually deploy the jobs, for example, with a startup servlet, then the classes need to be bundled with your application and the application needs to be redeployed in order to update the jobs. Also, the jobs would need to manually look up or create any resources they need such as database connections or EJBs, whereas with deployable jobs those resources can be provided to the job.

Q: Does the job need to have any special code?

A: The job implements the same `org.quartz.Job` interface as any Quartz job. It also needs to have setter methods for any resources configured for the job. But there are no special base classes or interfaces in order for the job to be deployed in this way.

Q: Isn't this an awful lot of XML?

A: It is more than we'd prefer. We'll look at how to reduce it in the future.

Q: Do I need a separate JAR and XML for every job?

A: No, you can list one or many jobs in each module you deploy. However, it's only possible to redeploy the module as a whole. So the benefit of deploying the jobs individually is you can update them individually, but it requires more JARs (or at least more XML plans). Each job can be started and stopped individually in any case; the only difference is redeployment.

Q: Can I configure Quartz to save jobs to a database?

A: There's normally no need to; each time you restart the server, the deployed jobs are scheduled again. If you have a use case where saving to the database is important, please let us know.

Q: I got the error "`Error: Unable to distribute my-jobs.jar: Cannot deploy the requested application module because no deployer is able to handle it...`"

A: That means Geronimo didn't link up the JAR you deployed with the Quartz deployer module. First, check that the namespace is correct in your XML plan. The jobs element should have `xmlns="http://geronimo.apache.org/xml/ns/plugins/quartz-0.2"` (make sure that is exact, though it may change for future releases of the plugin). If that's correct, make sure the `quartz-deployer` module is running. Try `java -jar bin/deployer.jar list-modules` and make sure that you see `gplugins/quartz-dpeloyer/.../car` in the list and that it has a `+` next to it. If not, you can run `java -jar bin/deployer.jar start quartz-deployer` to start it.

4.3. Quartz Console Example

The Quartz console walkthrough will be added when the Quartz console plugin is released.