Migrating Scheduled Tasks from GSRS 2.x to GSRS 3.x

As of 2 March 2022

|  |
| --- |
| **Note:** This document is a rough overview of some of the specific tasks that may be necessary to migrate existing custom scheduled tasks from GSRS 2.x to meet GSRS 3.x expectations. This guide is incomplete and is likely to be updated with more details and clarifications. |

# Creating standalone jar with only the scheduled task(s)

Unlike with GSRS 2.x, GSRS 3.x allows standalone jars that contain one or more GSRS extensions (like scheduled tasks) to be easily created and imported into a custom build of a GSRS entity service without directly requiring changes to the core GSRS library codebase. In addition, a standalone jar can be created with one or more extensions and simply made available in an already built and deployed version of a GSRS entity service, provided it’s accessible from the classpath and has the necessary changes to the configuration file.

This document describes some of the steps needed to create a scheduled task standalone jar. Much of what is described here would also apply to other GSRS extensions.

The steps described here are:

1. Creating a maven project and adding necessary dependencies
2. Writing/migrating a custom Scheduled Task implementation in java
3. Building and Deploying

Step 1: Creating a maven project and adding necessary dependencies

First, create a new Maven project, it does not need to extend Spring boot.

Modify the pom.xml file to add a dependency to “gsrs-scheduled-tasks” and any other gsrs libraries which you may be using. If you are making a plain java maven project without Spring, you will need to add a dependency to “quartz”. The examples below make use of a few other libraries as well, such as Lombok, which should also be added to the pom.xml file as needed.

Step 2: Writing/migrating a custom Scheduled Task implementation in java

**Changes to packages and imports from 2.x to 3.x:**

GSRS 3 has moved some packages around classes

**GSRS 2.x**

|  |
| --- |
| **import** ix.core.plugins.CronExpressionBuilder; **import** ix.core.plugins.SchedulerPlugin; |

are now in the package gsrs.scheduledTasks

**GSRS 3.x**

|  |
| --- |
| **import** gsrs.scheduledTasks.CronExpressionBuilder; **import** gsrs.scheduledTasks.SchedulerPlugin; |

TimeUtil was in

**GSRS 2.x**

|  |
| --- |
| **import** ix.core.util.TimeUtil; |

and is now in

**GSRS 3.x**

|  |
| --- |
| **import** gov.nih.ncats.common.util.TimeUtil; |

**Changes to interfaces and contracts:**

The interfaces have changed in 3.X.

In GSRS 2.x scheduled tasks were actually not a top-level concept but were a special form of the interface “Initializer” that extended the **ScheduledTaskInitializer** class. An initializer is simply a piece of code that gets run when the application starts. Consider the interface below:

**GSRS 2.x**

|  |
| --- |
| **public interface** Initializer {    */\*\*  \* Settings to initialize with, passed from the config file.  \** ***@param m*** *\*/* **default** Initializer initializeWith(Map<String,?> m){  **return this**;  }    **void** onStart(Application app); } |

Each initializer in GSRS 2.x required 2 methods:

**GSRS 2.x**

|  |
| --- |
| **public** Initializer initializeWith(Map<String, ?> m); |

**GSRS 2.x**

|  |
| --- |
| **void** onStart(Application app); |

The first method “initializeWith” was used the passed in Map as a map of key-value pairs from the configuration file which you could use to initialize your object with settings. While flexible, this mechanism was error-prone in that if you changed something in the configuration and forgot to update the java code it might silently fail or behave in an unexpected way.

The second method “onStart” accepts the Play Application object and will perform some executable task based on that application object.

The ScheduledTaskInitializer abstract class then behaved as follows:

**GSRS 2.x**

|  |
| --- |
| **public abstract class** ScheduledTaskInitializer **implements** Initializer{  ...    **@Override**  **public** Initializer initializeWith(Map<String, ?> m) {  String suppliedCron = (String)m.get("cron");  **if**(suppliedCron !=null){  **this**.cron = suppliedCron;  }  Object autoRun = m.get("autorun");  **if**(autoRun ==null){  enabled = false;  }**else if**(autoRun **instanceof** Boolean){  enabled = (Boolean) autoRun;  }**else** {  enabled = Boolean.parseBoolean(autoRun.toString());  }  **return** **this**;  }  **@Override**  **public** **void** onStart(Application app) {  createTask()  .\_to(st -> enabled? st.enable() : st.disable())  .submit();  }  **public** ScheduledTask createTask(){  **return** SchedulerPlugin.ScheduledTask.of((l)->run(l))  .atCronTab(cron)  .description(getDescription());  }  **public** **abstract** **void** run(TaskListener l);  **public** **abstract** String getDescription();  ... |

Effectively this means that all ScheduledTasks were initializers where the initializeWith operation expected some specific key-value pairs, and the onStart operation would submit a wrapped and scheduled version of the supplied `run` method to the SchedulerPlugin.

A typical implementation of a ScheduledTaskInitializer in 2.X may look like:

**GSRS 2.x**

|  |
| --- |
| **public class** MessagePrintScheduledTask **extends** ScheduledTaskInitializer {  **private** String message;  **@Override**  **public** Initializer initializeWith(Map<String, ?> m) {  **super**.initializeWith(m);  message = m.getOrDefault("message", "Hello World");  **return** **this**;  }  **@Override**  **public** **void** run(TaskListener l){  System.out.println(message);  }  **public** String getDescription(){  **return** "Scheduled Task that prints \"" + message + "\"";  }  } |

This scheduled task, when activated, would print whatever the supplied “message” is, specified in the configuration file, to standard out. Note that to have the “message” setting parsed from the configuration file, the class needs to override initializeWith and still call the method on the super class. An example configuration to activate this in 2.X would be:

**GSRS 2.x**

|  |
| --- |
| ix.core.initializers+=[  # schedule a full dump of the data  {  "class":"ix.ginas.initializers.MessagePrintScheduledTask",  "cron":"\*/10 \* \* \* \* ?", #Every 10 seconds  "autorun":true,  "message":"Hello World"  }  ] |

This configuration would tell the above scheduled task to be automated, run once every 10 seconds, and print the message "Hello World".

For 3.0 the pattern is much the same, but the concept of the underlying Initializer class has been removed, the abstract class has been slightly adjusted, and implementing an “initializeWith” method is no longer necessary. Instead, JSON deserialization using the Jackson library will automatically populate any configuration settings into the fields/methods for initialization. The equivalent scheduled task above is shown in 3.x syntax below:

**GSRS 3.x**

|  |
| --- |
| **@Data public class** MessagePrintScheduledTask **extends** ScheduledTaskInitializer {  **private** String message;  **@Override**  **public** **void** run(SchedulerPlugin.JobStats stats,  SchedulerPlugin.TaskListener l){  System.out.println(message);  }  **public** String getDescription(){  **return** "Scheduled Task that prints \"" + message + "\"";  }  } |

The expected configuration change (application.conf) for 3.x would be:

**GSRS 3.x**

|  |
| --- |
| gsrs.scheduled-tasks.list+=[  # schedule a full dump of the data  {  "scheduledTaskClass":"my.tasks.MessagePrintScheduledTask",  "parameters" : {  "cron":"\*/10 \* \* \* \* ?", #Every 10 seconds  "autorun":true,  "message":"Hello World"  }  }  ] |

The parsing of “message” from the config parameter into the private field, in this case, is possible because of the “@Data” annotation which uses the library “lombok.Data” to automatically add setters and getters to the java object. The generated setter for message (“setMessage”) will be detected and used by the Jackson library on initialization of the scheduled task. The abstract class also includes common scheduledTask fields such as enabled (or autorun) and cron so you don’t have to worry about setting those correctly.

The two methods that need to be overwritten for a given scheduled task remain the same as they were in 2.x, with one minor difference to the “run” method.

The “run” method will contain the code to be executed, and receives the same TaskListener as in 2.x which allows communication back to the admin panel to update status messages, etc. A new parameter “JobStats” is also available which allows the executing code to probe statistics on how often the task has been run, when it’s next scheduled to be run, etc. Long running tasks can be performed within the run command, but should update the listener periodically on the status. Best practices would also be to periodically check if “Thread.currentThread().isInterrupted()” returns true, in which case the task has been marked as cancelled. The task is considered complete when the method returns.

The “getDescription()” method will return text to show in the administration panel of the user interface and scheduled tasks API. This is typically a very short description resembling a name for the specific task.

One especially helpful improvement in 3.x is the support for Spring Dependency injection within extensions like scheduled tasks. While 2.x would require configuration to find resources within a constructor or the initializeWith method, GSRS 3.x allows for any fields or methods annotated with @Autowired to be injected with the correct bean.

For example, the code below demonstrates how a resource like “ReindexService” would typically be implemented for use in 2.x.

**GSRS 2.x**

|  |
| --- |
| **public class** ReindexServiceScheduledTask **extends** ScheduledTaskInitializer {  **private** ReindexService service;  **@Override**  **public** Initializer initializeWith(Map<String, ?> m) {  **super**.initializeWith(m);  service = ***<code to find static version of ReindexService>***  **return** **this**;  } **...**  } |

However, in 3.x this can be simplified to:

**GSRS 3.x**

|  |
| --- |
| **public class** ReindexServiceScheduledTask **extends** ScheduledTaskInitializer {  **@Autowired**  **private** ReindexService service;  **...**  } |

There are many beans that can be used and dependency injected within a scheduled task, and this feature typically simplifies development and readability of the codebase.

Step 3: Building and Deploying

Taking the simple MessagePrintScheduledTask scheduled task task above, after writing the single java file into the src/main/java/my/tasks directory, making sure to add any dependencies to the maven pom.xml file, a simple build target should be added to the pom.xml file like the following:

**GSRS 3.x pom.xml**

|  |
| --- |
| <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-resources-plugin</artifactId>  <version>2.5</version>  <configuration>  <encoding>UTF-8</encoding>  </configuration>  </plugin>  </plugins>  </build> |

After this, run `mvn clean package`. This should produce a jar file within the target directory. This jar file can either be directly included in other builds when packaging together a microservice / war file *or* the jar file can even be added to a `lib` directory within a tomcat instance for use by an existing deployed webapp. Alternatively, it’s sometimes desirable to have extensions like this directly embedded as source code into a fork of a repository like `gsrs3-main-deployment` under the src folder for the target microservice, depending on how likely it will be that changes to the code will be needed.

Regardless of *how* the code is accessible within the classpath, once accessible it still needs to be activated within the configuration file (typically application.conf). As described above, the configuration pattern for 3.x is slightly different than 2.x. The following snippet, if appended to the running webapp’s configuration file, should activate this scheduled task:

**GSRS 3.x**

|  |
| --- |
| gsrs.scheduled-tasks.list+=[  # schedule a full dump of the data  {  "scheduledTaskClass":"my.tasks.MessagePrintScheduledTask",  "parameters" : {  "cron":"\*/10 \* \* \* \* ?", #Every 10 seconds  "autorun":true,  "message":"Hello World"  }  }  ] |

A few notes on changes here:

1. Initialization parameters are shown under the “**parameters**” keyword rather than at the root-level. There is experimental support for root-level parsing of parameters as well, but this is not the preferred syntax at this time.
2. The fully qualified class name is now specified under the scheduledTaskClass key instead of the “class” key in 2.x.
3. Some of the mechanisms used to deserialize values in the configuration file have changed in ways that must be addressed. Some testing may be needed to ensure that deserialization works as expected.

On this final point, we use Jackson to map the properties from the config to the scheduled task objects. This wasn’t done like that in GSRS 2.x so we were more flexible in variable names and property look ups. It was not uncommon for a single initialization property to be called “output.path” in 2.x, for example. However, in 3.x this convention would behave differently, as the period character would be interpreted as a signal that there was a nested object. It’s suggested that “.” characters not be used as keys within initialization parameters.