**How To Write a Scheduled Task**

In G-SRS, a *Scheduled Task* is a task that can be performed by an administrator at will via the web interface, or on a specific schedule. They can be thought of as server-side macros.

***When you might want to use one:***

* You want to produce an automated report every day
* You want to update a lucene index for custom searches every week
* You want to refresh a data pull periodically
* You want to allow an admin to temporarily disable browsing the web interface

***When you might NOT want to use one:***

* You want to allow an admin to be able to perform some arbitrary task that takes a single parameter via the web interface (you can not pass parameters to the tasks via the web interface -- so making a task that allows an admin user to say “produce a report of all records edited by user XXX” would not currently be possible at runtime)
* You want the web server to restart every night. (currently you can not restart the server from inside the server -- you would want to configure a cron job on the server instead)
* You want to perform a certain task whenever a new record is added. (you should use an EntityProcessor for this, as the scheduled tasks have no arbitrary trigger mechanism)

**Generic Initializer Interface**

The **Initializer** interface is used to initialize the application in some way. Currently, all scheduled tasks are added via an **Initializer**, though that is not the only use for an initializer. For scheduled tasks, an **Initializer** will tell the application to add a task to the scheduled task registry on startup, along with whatever setup parameters might be needed. An **Initializer** must be a class with a public 0 argument constructor that has 1 required method:

```

public void onStart(Application app);  
```

There is also 1 method with a default implementation, which can be overridden:

```

public default Initializer initializeWith(Map<String,?> m)

```

* The onStart method passes in the Application object and is called once, when the application is launched, provided the initializer is specified in the config file.
* The *initializeWith* method passes the config object specified for the initializer in the config file, instantiated as a Map.

For example, in the config file, you can register the intializer as follows:

```  
 ix.core.initializers=[

# schedule a full dump of the data

{

"class":"ix.ginas.initializers.ScheduledExportTaskInitializer",

"username":"admin",

"cron":"0 9 2 \* \* ?", #2:09 AM every day

"autorun":false,

"name":"Full GSRS export"

}

]

```

This tells GSRS to instantiate a **ScheduledExportTaskInitializer**, passing the config object to the *initializeWith* method.

**ScheduledTaskInitializer Abstract Class**

**ScheduledTaskInitializer** is a helper abstract class for Scheduled Task Initializers, taking care of some of the most common things that would be done when registering a scheduled task to the registry. This implements the **Initializer** interface, and will automatically register a defined task to the scheduled task registry with common parameters. A subclass requires implementation of 2 methods:

```

public void run(TaskListener l);

public String getDescription();

```

* The *run* method performs the actual task, taking a **TaskListener** object to give feedback to the application about the current running state of the executing task. Typically, a scheduled task will make calls to two methods in TaskListener to update the status of the running task:
  + public TaskListener progress(double p)
    - Updates the progress percentage (from 0 to 100)
  + public TaskListener message(String msg)
    - Updates a status message
* The *getDescription* method should return a simple description of the task, for displaying on the admin panel.

In addition, the **ScheduledTaskInitializer** supports and uses the following config variables for initialization:

|  |  |
| --- | --- |
| autorun | Boolean value of whether to have task execute on schedule automatically, or to wait to be explicitly enabled by an admin |
| cron | Cron-tab schedule string format |

**Example Implementation: Output simple names report from SQL command every day at 6pm**

For example, we may want to output a simple text report from a specified SQL command periodically. In this example we will create a **ScheduledTaskInitlializer** which takes a specified output path and a specified SQL statement, and outputs that report to a file.

```

package ix.ginas.initializers;

import java.io.BufferedOutputStream;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.PrintStream;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.time.format.DateTimeFormatter;

import java.util.Map;

import java.util.Optional;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

import ix.core.initializers.Initializer;

import ix.core.plugins.SchedulerPlugin.TaskListener;

import ix.core.util.TimeUtil;

import play.Logger;

import play.db.DB;

/\*\*

\* Used to schedule output of certain reports, using defined SQL queries in the

\* config file

\*

\* @author tyler

\*

\*/

public class SQLReportScheduledTaskInitializer

extends ScheduledTaskInitializer {

private String name = "sqlReport";

private String sql;

private String path;

private DateTimeFormatter formatter = DateTimeFormatter.ISO\_LOCAL\_DATE;

private Lock lock = new ReentrantLock();

/\*\*

\* Returns the File used to output the report

\*

\* @return

\*/

public File getWriteFile() {

String date = formatter.format(TimeUtil.getCurrentLocalDateTime());

String fpath = path.replace("%DATE%", date);

return new File(fpath);

}

@Override

public Initializer initializeWith(Map<String, ?> m) {

super.initializeWith(m);

sql = Optional.ofNullable((String) m.get("sql")).get();

name = Optional.ofNullable((String) m.get("name")).orElse(name);

path = (String) m.get("output.path");

if (path == null) {

path = "reports/" + name + "-%DATE%.txt";

}

return this;

}

private PrintStream makePrintStream(File writeFile) throws IOException {

return new PrintStream(

new BufferedOutputStream(new FileOutputStream(writeFile)),

false, "UTF-8");

}

@Override

public void run(TaskListener l) {

try {

lock.lock();

l.message("Initializing SQL");

File writeFile = getWriteFile();

writeFile.getParentFile().mkdirs();

try (PrintStream out = makePrintStream(writeFile)) {

l.message("Establishing connection");

try (Connection c = DB.getConnection()) {

Statement s = c.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_READ\_ONLY);

l.message("Executing Statement");

ResultSet rs1 = s.executeQuery(sql);

// Count rows

l.message("Counting Rows");

rs1.last();

int total = rs1.getRow();

rs1.beforeFirst();

l.message("Preparing export rows");

int ccount = rs1.getMetaData().getColumnCount();

l.message("Getting column names");

for (int i = 1; i <= ccount; i++) {

out.print(rs1.getMetaData().getColumnName(i));

out.print("\t");

}

out.println();

double denom = 1 / (total \* 100.0);

// Output each row

while (rs1.next()) {

for (int i = 1; i <= ccount; i++) {

out.print(rs1.getString(i));

out.print("\t");

}

out.println();

int r = rs1.getRow();

l.progress(r \* denom);

if (r % 10 == 0) {

l.message("Exporting " + r +

" of " + total);

}

}

rs1.close();

} finally {

l.message("Closed Connection");

}

} catch (Exception e) {

Logger.error("Error writing SQL export", e);

}

} finally {

lock.unlock();

}

}

@Override

public String getDescription() {

return "SQL Report:" + name + ". Output to:" + getWriteFile().getPath();

}

}

```

Here, we initialize certain properties like the output path, name of report, and the SQL to use inside of the *initializeWith* method. Private member fields are initialized there, which will be read from the config file.

* Note that *initializeWith* returns **this**, as it is required to return an **Initializer**.

The real meat of the code occurs in the *run* method. Here, we perform the following steps:

1. Establish a lock so that 2 instances of the same task can not be done at the same time.
2. Establish the file to be written
3. Establish a connection
4. Perform the query
5. Count the rows
6. Export the column names to the file
7. Iterate through the rows and output each row, tab-delimited, to the output file.
8. Close the connection

As we perform the steps involved in the task, the **TaskListener** is updated with information to give back to the user. This is done using the *message* and *progress* methods.

**Enabling the Scheduled Task in the Config File**

After writing an **ScheduledTaskInitializer**, it must be enabled in the application. To do this, first make sure that it is visible to the classpath. Next, in the config file for the application (typically ginas.conf), it must be explicitly registered in the *ix.core.initializers* property. Adding these lines to the end of the config file will do that:

```

ix.core.initializers+={

"class":"ix.ginas.initializers.SQLReportScheduledTaskInitializer",

"autorun":true,

“cron":"0 1 18 \* \* ?", #6:01 PM every day

"sql":"select b.uuid, b.approval\_id, a.name, a.type from ix\_ginas\_name a join ix\_ginas\_substance b on b.uuid=a.owner\_uuid",

"name":"Simple Name Report",

"output.path":"reports/nameReport-%DATE%.txt"

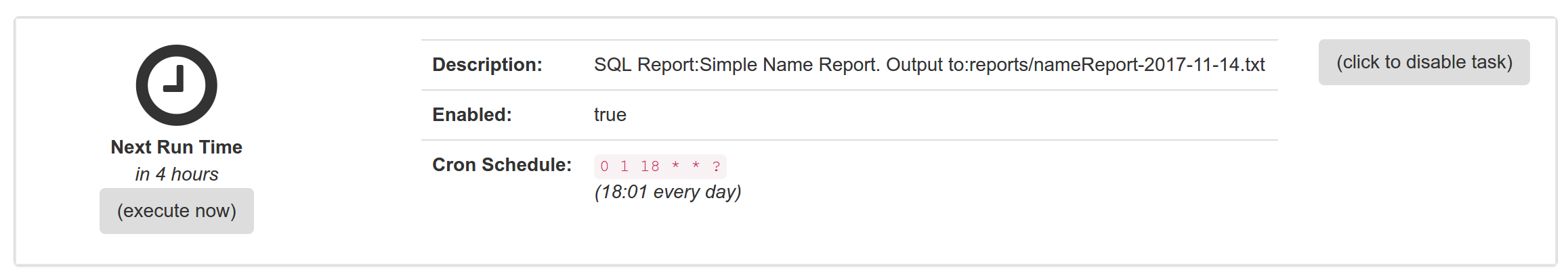
}

```

This adds the initializer, which will make the scheduled task appear in the admin panel of the UI. Note the settings given here:

* *class* gives the fully qualified path to the class we will be using as an initializer
* *autorun* is set to true, meaning that it will be enabled without admin intervention
* *cron* is set to a cron-tab value for 6PM every day
* *sql* here is set to the raw SQL command used to generate our simple name report
* *name* here is set to “Simple Name Report”, for use in the admin panel of the UI
* *output.path* is set to “reports/nameReport-%DATE%.txt”. In our **ScheduledTaskInitalizer**, we set the path to create the necessary folders if they don’t exist, and also to replace “%DATE%” with the current Date in the format “YYYY-MM-DD”.

After starting the application, log in as an admin. In the admin panel, under scheduled tasks, you should now see the new task we’ve created:



From here, you can either chose to execute the task immediately, or disable the task temporarily.

**Some Notes:**

* You can register multiple Initializers using the same class, but different config parameters. So it would be possible to have several tasks all using the SQL export mechanism, but with different SQL statements.
* Currently, setting a task to be enabled in the UI will not persist across restarts

**Writing Tests**

**Coming soon**