



JOB SCHEDULER

Tutorial Job Implementation

Documentation
November 2008

Contents

1	Introduction.....	3
2	Summary	3
3	Communicating the Job Script to the Job Scheduler	4
3.1	An Example of a Script Directly Included in the XML Configuration.....	4
3.2	An Example of a Reference to an External Script.....	5
3.3	An Example for a Reference to an External Script with Subsequent Call of a Method.....	6
4	Implementation.....	6
4.1	Job implementation with Java	6
4.1.1	Minimal implementation	6
4.1.2	Use of the Java API.....	7
4.2	Job implementation with script languages.....	7
5	Job Scheduler Objects	8
6	Job Scheduler Methods.....	8
6.1	spooler_init().....	8
6.2	spooler_open()	8
6.3	spooler_process().....	9
6.4	spooler_close().....	9
6.5	spooler_on_success()	9
6.6	spooler_on_error().....	9
6.7	spooler_exit().....	9
6.8	Special rules for order jobs.....	9
6.8.1	spooler_task.order	9
6.8.2	Return value of spooler_process()	9
6.8.3	Execution Sequence	10
7	Example Job: FTP Download.....	11
7.1	Establish a Connection to an FTP Server	12
7.2	Download Files from the FTP Server	12
7.3	Convert Script Parameters into Job-Parameters	13
7.4	Integration of Error Handling	15
7.5	Use of the Job Scheduler Methods	16
7.6	Create an External Script File Referenced in the XML Configuration	18
8	Debugging Jobs in a Java IDE	20

1 Introduction

This tutorial describes the programming of a job with the JOB SCHEDULER.

A job is the content of a <job> element in the JOB SCHEDULER'S XML configuration. The element specifies the name, title, program code, time slot and start time for every job.

Further information can be found in the following documents:

- *Technical Documentation* (Online Documentation)
Can be found at either [Installation directory of the Job Scheduler]/config/html/doc/de/index.html
or
via [http://\[Job Scheduler host\]:\[Job Scheduler port\]/doc/en/index.html](http://[Job Scheduler host]:[Job Scheduler port]/doc/en/index.html)
- *API Documentation.*
Can be found at either [Installation directory of the Job Scheduler]/config/html/doc/en/api.xml or
via [http://\[Job Scheduler host\]:\[Job Scheduler port\]/doc/en/api/api.xml](http://[Job Scheduler host]:[Job Scheduler port]/doc/en/api/api.xml)

The documentation in PDF and HTML format can be found at
[Installation directory of the Job Scheduler]/doc/en/scheduler_api.pdf
and
[Installation directory of the Job Scheduler]/doc/en/scheduler_api/sos_help.htm

2 Summary

The organization of the program code for a job for the JOB SCHEDULER is described in Chapter 3.

In Chapters 4-6, the Job Scheduler object is described, together with its methods. The methods and objects described can be used in any job implementation.

An example job is described from Chapter 7 onwards. This job has the purpose of collecting data from an FTP server. The script language used for this job is JavaScript. Similar jobs could be written in Java, Perl or VBScript.

3 Communicating the Job Script to the JOB SCHEDULER

There are three possible ways of making the JOB SCHEDULER aware of a job's program code:

- incorporation of the script directly in the XML configuration file
- including a reference to the external script file in the XML configuration
- including a reference to the external script file in the XML configuration and calling the function of the referenced script in the XML configuration

Whilst the first method is the most direct solution and effective with short scripts, with larger scripts, this method tends to make the XML configuration too complicated.

In the second method it is possible to process a job script without an additional function call when particular JOB SCHEDULERS method names are used in the referenced script (see Chapter 4).

The XML configuration file is located in the directory *config* relative to your installation. The default configuration file is named *scheduler.xml*.

3.1 An Example of a Script Directly Included in the XML Configuration

```
<?xml version="1.0" encoding="iso-8859-1"?>
<spooler>
  <config>
    <jobs>
      <job name="test_job">
        <script language="JavaScript">
          <![CDATA[
            spooler_log.info( "Hello world!" );
          ]]>
        </script>
      </job>
    </jobs>
  </config>
</spooler>
```

3.2 An Example of a Reference to an External Script

```
<?xml version="1.0" encoding="iso-8859-1"?>
<spooler>
  <config>
    <jobs>
      <job name = "test_job">
        <script language = "JavaScript">
          <include file = "jobs/hello_world.js"/>
        </script>
      </job>
    </jobs>
  </config>
</spooler>
```

alternatively for Java:

```
      <script language = "Java"
              java_class = "scheduler.job.HelloWorld"/>
    </job>
  </jobs>
</config>
</spooler>
```

Functions can be implemented in an external script, in order to save their being written repeatedly (see Chapter 4).

For example, "hello_world.js" contains:

```
function spooler_process() {
  spooler_log.info("Hello world!");
  return false;
}
```

In the same way, the Java class "HelloWorld.java" can be implemented so:

```
package scheduler.job;
import sos.spooler.*;

public class HelloWorld extends Job_impl {

  public boolean spooler_process() {
    spooler_log.info("Hello world!");
    return false;
  }
}
```

3.3 An Example for a Reference to an External Script with Subsequent Call of a Method

```
<?xml version="1.0" encoding="iso-8859-1"?>
<spooler>
  <config>
    <jobs>
      <job name = "test_job">
        <script language = "JavaScript">
          <include file = "jobs/log.js"/>
          <![CDATA[
            log_info("Hello world!");
          ]]>
        </script>
      </job>
    </jobs>
  </config>
</spooler>
```

The file "jobs/log.js" contains, for example:

```
function log_info( msg ) {
    spooler_log.info( msg );
}
```

4 Implementation

4.1 Job implementation with Java

You are free to implement any job with Java. The JOB SCHEDULER does not restrict the use of jobs to implementations that use the Java API interface. You can use the API Interface to make use of the objects and methods that the Job Scheduler exposes, e.g. for logging or mail processing.

4.1.1 Minimal implementation

To make your implementation available to the JOB SCHEDULER follow these steps:

- Create your Java job as a class file and add it to a Java archive.
- Modify the configuration file `factory.ini` that is located in the directory `config` of your installation: add the path to your Java archive to the entry `class_path` in the section `[java]`. Use `;` as separator for archive paths on Windows Servers, use `:` as separator for paths on Unix Servers. Example:

```
[java]
class_path          = /mypath/sample.jar;/scheduler/lib/sos.connection.jar;...
```

- Add a job definition to the configuration file `scheduler.xml` in the directory `config` of your installation. A minimal Java job definition for a class `com.example.job.HelloWorld` that is contained in the archive `/mypath/sample.jar` might look like this:

```
<spooler>
  <config>
    <jobs>
      <job name = "test_job">
        <script language = "Java"
          java_class = "com.example.job.Helloworld"/>
      </job>
    </jobs>
  </config>
</spooler>
```

- Restart the JOB SCHEDULER to load your job implementation.
- If you want your job to be started automaticalle then add a `<run_time>` element to your job definition. For more information of the job definition see the documentation that is located in your directory docs.

4.1.2 Use of the Java API

A Job in Java inherits from the abstract `sos.spooler.Job_impl` super class. This class is contained in the archive `sos.spooler.jar` located in a directory named `lib` of your installation. Please, include this archive in your Java IDE project and import `sos.spooler.Job_impl`.

For deployment of your job, please follow the steps in the preceeding chapter.

You can implement the abstract methods of the super class that are explained in more detail in the following chapters. These methods (`spooler_init()`, `spooler_process()` etc.) give the JOB SCHEDULER more control on the behaviour of your job.

You can use all the objects and methods exposed by the JOB SCHEDULER API. A detailed documentation is included in your installation directory docs in the formats XML, PDF and HTML. A good reference is <http://www.sos-berlin.com/doc/en/scheduler.doc/api/api.xml>.

4.2 Job implementation with script languages

The script languages VBScript, Perl and JavaScript can be used to implement a job.

- JavaScript is available for all platforms, the JOB SCHEDULER includes the JavaScript implementation "spidermonkey", see <http://www.mozilla.org/js/>.
- PerlScript is available for Unix Servers and for Windows Servers that use the perl port of ActiveState (<http://www.activestate.com>).
- VBScript is available for Windows only.

5 JOB SCHEDULER Objects

The objects of the JOB SCHEDULER API are:

- `spooler`: object for the methods of the Job Scheduler
- `spooler_log`: object for protocols
- `spooler_job`: the job object
- `spooler_task`: object for the processes in which the job will be carried out in

Example properties and methods of these objects will be used and explained in this tutorial. A complete overview can be obtained in the documentation for the relevant Java, COM or Perl interface.

6 JOB SCHEDULER Methods

Every job implementation can implement optional methods, independent of the script language. These methods are automatically called by the JOB SCHEDULER, in so far as they occur in a script. These additional methods are:

- `spooler_init()`
- `spooler_open()`
- `spooler_process()`
- `spooler_close()`
- `spooler_on_success()`
- `spooler_on_error()`
- `spooler_exit()`

The following sections describe when these methods should be called. Their implementation is optional.

6.1 `spooler_init()`

`spooler_init()` is called once after the script has been loaded.

A return value of `True`, `1` or `Empty` (no return value) is interpreted as `True` and allows the processing to continue.

A return value of `False`, `0`, `Nothing` or `Null` is interpreted as `False` and the processing is stopped. `spooler_exit()` will be called and the script closed.

Should an error occur in `spooler_init()` then the task will be continued with `spooler_exit()`.

`spooler_init()` is particularly suited for setting up of objects, database connections, etc.

6.2 `spooler_open()`

Is called at the start of a task. The return value is interpreted as per `spooler_init()`. Should the value `False` be returned or should an error occur, then `spooler_close()` is called. Otherwise, `spooler_process()` will be started.

`spooler_open()` is particularly suited for setting up a number of objects and / or opening a connection (database, FTP server, etc.).

6.3 spooler_process()

The return value is interpreted as with `spooler_init()`. Should `False` be returned, then the task will be continued using `spooler_close()`; after `True` the task will be continued with a further call of `spooler_process()`.

Each call of `spooler_process()` is counted as a job step in the Scheduler interface.

`spooler_process()` is particularly suited for the step-by-step processing of a number of objects, which have been, for example set up using `spooler_open()`. The implementation of `spooler_process()` offers the advantage that progress of the process can be followed in Job Scheduler interface and that each task can be monitored on a step by step basis.

6.4 spooler_close()

Is called at the end of a task, either after an error or after the methods `spooler_open()` or `spooler_process()` return `False`.

Either `spooler_on_success()` or, in the event of an error, `spooler_on_error()` will be called after `spooler_close()`.

`spooler_close()` is particularly suited for closing connections (database, FTP server, etc.) which may be open.

6.5 spooler_on_success()

Is called after `spooler_close()`, should no error have occurred.

6.6 spooler_on_error()

Is called after `spooler_close()`, should an error have occurred. Another task can then access `spooler_task.error`.

6.7 spooler_exit()

Is called immediately after a script is closed.

`spooler_exit()` is particularly suited for clearing up remaining objects.

6.8 Special rules for order jobs

Jobs that are used in job chains need to be configured to process orders: `<job order="yes">`. For job implementation this results in the following consequences:

6.8.1 spooler_task.order

An order object is available in `spooler_process()`.

6.8.2 Return value of spooler_process()

The return value of order jobs determines if an order was successful or not.

True: The order will be continued in the `next_state` of the current job chain node.

False: The order will be continued in the `error_state` of the current job chain node.

If however a job error occurs (caused by `spooler_log.error()`), the job will be stopped and the order will remain in the current job chain node. In this case the return value doesn't have any effect.

6.8.3 Execution Sequence

The execution sequence of the job functions basically remains the same. However, `spooler_process()` may be called repeatedly if the job processes multiple orders without ending in between (i.e. if the `idle_timeout` is greater than the time between two orders).

For a job which processes 3 orders without ending in between, the following functions will be called:

```
spooler_init()
  spooler_open()
    spooler_process() (spooler_task.order contains order 1)
    spooler_process() (spooler_task.order contains order 2)
    spooler_process() (spooler_task.order contains order 3)
  spooler_close()
  spooler_on_success() oder spooler_on_error()
spooler_exit()
```

`spooler_on_success()` and `spooler_on_error()` do not refer to errors of the order but to errors of the job.

7 Example Job: FTP Download

This Job downloads files from an FTP server.

The script language used is JavaScript.

As JavaScript can neither access an FTP server nor a file system, objects from suitable Java classes are used to achieve this.

The use of these Java classes is encapsulated in an external JavaScript file ("jobs/ftp.js"). The content of this file is not relevant to the purpose of this tutorial (the source code of this file is delivered with the JOB SCHEDULER). It can be assumed, that an object in the class included in this file possesses methods which enable, for example, a connection to an FTP server to be established, or a file to be written in a file system.

After the jobs/ftp.js file has been referenced in the JOB SCHEDULER job definition, the methods and functions contained in this file can then be called (see Section 3.3).

```
<?xml version="1.0" encoding="iso-8859-1"?>
<spooler>
  <config>
    <jobs>
      <job name          = "ftp_get"
          title         = "Get files from ftp server">
        <script language = "JavaScript">
          <include file = "jobs/ftp.js"/>
          <![CDATA[
<!-------Calls are inserted here----->
          ]]>
        </script>
      </job>
    </jobs>
  </config>
</spooler>
```

The XML framework will be extended later and will not be presented in this document until then. At the moment, only the function and method calls will be described.

This script will grow with each section of the example. Every new line of script will be described. Block text is used for lines of script.

7.1 Establish a Connection to an FTP Server

The following calls establish a connection to an FTP server.

Specify the connection parameters to the FTP server (FTP server Hostname, user name and password):

```
var ftp      = null;           // this forms the JavaScript FTP class object
var ftp_host = "localhost";
var ftp_user = "anonymous";
var ftp_pass = "anonymous@localhost";
```

Instantiate the FTP class object with the FTP server host name as argument:

```
ftp = new Ftp(ftp_host);
```

Establish the connection to the FTP server on ftp_host with ftp_user as user name and ftp_pass as password

```
ftp.login(ftp_user, ftp_pass);
```

Protocol information in Log for output to the SCHEDULER user interface using the SCHEDULER objects (see Chapter 5):

```
var msg = "ftp connection successful for " + ftp_user + "/*:*@" + ftp_host + "."
spooler_log.info( msg );
spooler_job.state_text = msg;
```

Closure of the FTP connection:

```
ftp.logout();
```

7.2 Download Files from the FTP Server

The script presented in Section 7.1 is extended here with calls for navigation on the FTP server, reading an FTP directory and downloading the files from the selected directory.

```
var ftp      = null;
var ftp_host = "localhost";
var ftp_user = "anonymous";
var ftp_pass = "anonymous@localhost";
```

Further parameters are defined (FTP and local directories)

```
var ftp_dir  = "/test";
var lokal_dir = "./"; //the files will be saved in the current working directory
```

```
ftp = new Ftp(ftp_host);
ftp.login(ftp_user, ftp_pass);
var msg = "ftp connection successful for " + ftp_user + "/*:*@" + ftp_host + "."
spooler_log.info( msg );
spooler_job.state_text = msg;
```

Set the transfer mode ("binary" or "ascii"):

```
ftp.binary();
```

Specify the type of transfer ("active" or "passive"):

```
ftp.passive();
```

Change into the FTP directory ftp_dir:

```
ftp.cd(ftp_dir);
```

Read the names of the files in the FTP directory ftp_dir:

```
var list = ftp.dir();
```

Write the current FTP directory and the number of files found into the Log file and output the same information to the JOB SCHEDULER user interface (see Chapter 5):

```
msg = "ftp current working directory: " + ftp.pwd() +  
      " containing " + list.length + " files.");  
spooler_log.info( msg );  
spooler_job.state_text = msg;
```

Make an entry in the protocol with the name of each file found (will only be written if the debug level of the JOB SCHEDULER is ≥ 3) and save each file in lokal_dir:

```
for (var i in list) {  
    spooler_log.debug3("retrieving file: " + list[i]);  
    ftp.getFile(list[i], lokal_dir + list[i]);  
}
```

Output to the JOB SCHEDULER user interface

```
spooler_job.state_text = i + " file(s) successfully processed";
```

```
ftp.logout();
```

7.3 Convert Script Parameters into Job-Parameters

The ftp_host, ftp_user, ftp_pass, ftp_dir and lokal_dir script variables are defined as job parameters in this section. To this end, a <params> element is added into the <job>- element (see the XML configuration documentation). This has the advantage that the job can be more easily configured. The complete script will be later moved to an external file and referenced in the XML configuration.

Should, for example, the FTP connection data change, it is no longer necessary to make changes in the script - instead the job parameters in the XML configuration merely need to be changed. In addition, this opens the possibility of making the parameters for every job editable via a user interface.

```

<?xml version="1.0" encoding="iso-8859-1"?>
<spooler>
  <config>
    <jobs>
      <job name          = "ftp_get"
           title        = "Get files from ftp server">
        <params>
          <param name = "ftp_host"  value = "localhost"/>
          <param name = "ftp_user"  value = "anonymous"/>
          <param name = "ftp_pass"  value = "anonymous@localhost"/>
          <param name = "ftp_dir"   value = "/test"/>
          <param name = "lokal_dir" value = "./"/>
        </params>
        <script language = "JavaScript">
          <include file = "jobs/ftp.js"/>
          <![CDATA[
<!------- Calls are inserted here ----->
          ]]>
        </script>
      </job>
    </jobs>
  </config>
</spooler>

```

The calls are presented below. In comparison with the previous section, the configured parameters are now accessed using the spooler_task SCHEDULER object (see Chapter 5).

```

var ftp      = null;
var ftp_host = spooler_task.params.var("ftp_host");
var ftp_user = spooler_task.params.var("ftp_user");
var ftp_pass = spooler_task.params.var("ftp_pass");
var ftp_dir  = spooler_task.params.var("ftp_dir");
var lokal_dir = spooler_task.params.var("lokal_dir");

ftp = new Ftp(ftp_host);
ftp.login(ftp_user, ftp_pass);
var msg = "ftp connection successful for " + ftp_user + "/*.*@" + ftp_host + "."
spooler_log.info( msg );
spooler_job.state_text = msg;
ftp.binary();
ftp.passive();
ftp.cd(ftp_dir);
var list = ftp.dir();
msg = "ftp current working directory: " + ftp.pwd() +
      " containing " + list.length + " files.");
spooler_log.info( msg );

```

```

spooler_job.state_text = msg;
for (var i in list) {
    spooler_log.debug3("retrieving file: " + list[i]);
    ftp.getFile(list[i], lokal_dir + list[i]);
}
spooler_job.state_text = i + " file(s) successfully processed";
ftp.logout();

```

7.4 Integration of Error Handling

try-catch blocks together with finally will now be added. Should an error occur in a try block, then the catch block is executed. The finally block is executed as a last step in every case.

This variable must be externally (globally) defined, so that it is valid in every block:

```

var ftp      = null;

try {
    var ftp_host = spooler_task.params.var("ftp_host");
    var ftp_user = spooler_task.params.var("ftp_user");
    var ftp_pass = spooler_task.params.var("ftp_pass");
    var ftp_dir  = spooler_task.params.var("ftp_dir");
    var lokal_dir = spooler_task.params.var("lokal_dir");

    ftp = new Ftp(ftp_host);
    ftp.login(ftp_user, ftp_pass);
    var msg = "ftp connection successful for " + ftp_user + "/*:*@" + ftp_host + "."
    spooler_log.info( msg );
    spooler_job.state_text = msg;
    ftp.binary();
    ftp.passive();
    ftp.cd(ftp_dir);
    var list = ftp.dir();
    msg = "ftp current working directory: " + ftp.pwd() +
        " containing " + list.length + " files.");
    spooler_log.info( msg );
    spooler_job.state_text = msg;
}

```

Counter for successful downloads:

```

var cnt_success = 0;

for (var i in list) {
    try {
        spooler_log.debug3("retrieving file: " + list[i]);
        ftp.getFile(list[i], lokal_dir + list[i]);
        cnt_success++;
    } catch(err) {

```

Write a warning in the protocol using the SCHEDULER object:

```

        spooler_log.warn( "error at download " + list[i] + ": " + err.message );
    }
}
    spooler_job.state_text = cnt_success + " file(s) successfully processed";
} catch(err) {

```

Write an error in the protocol using the SCHEDULER object:

```

    spooler_log.error( "ftp command could not be processed: " + err.message );
} finally {

```

The FTP connection should be closed, regardless of whether errors have occurred or not.

The `if(ftp != null)` test is necessary, should the instantiation already have caused an error and `ftp` not be an object which `logout()` can call.

```

    try{
        if( ftp != null ) { ftp.logout(); }
    } catch(err) {}
}

```

7.5 Use of the JOB SCHEDULER Methods

The methods of the JOB SCHEDULER (see chapter 6) are integrated in this section. The FTP object which specifies the connection options and collects the files to be processed is applied in the `spooler_open()` method. The collected files are then processed step by step – i.e. downloaded - in `spooler_process()`. The connection to the FTP server is then closed in `spooler_close()`.

This allows the error handling to dispense with the outer `try/catch-` block described in the previous section. In particular, this allows the job at each step to stop, continue and/or to end. The JOB SCHEDULER Web interface shows through the current number of completed process steps.

```
var ftp          = null;
```

Further global variables must be declared here – the array `list`, which collects the files to be processed; a counter `steps`, which records the process steps and the `cnt_success` counter, which records the successfully completed steps.

```
var list        = new Array();
var steps      = 0;
var cnt_success = 0;
```

Should an error occur in `spooler_open()` or `False` be returned, then `spooler_close()` is called:

```
function spooler_open() {
    var ftp_host = spooler_task.params.var("ftp_host");
    var ftp_user = spooler_task.params.var("ftp_user");
    var ftp_pass = spooler_task.params.var("ftp_pass");
    var ftp_dir  = spooler_task.params.var("ftp_dir");
    var lokal_dir = spooler_task.params.var("lokal_dir");

    ftp = new Ftp(ftp_host);

```

```

ftp.login(ftp_user, ftp_pass);
var msg = "ftp connection successfully for " + ftp_user + "/***@" + ftp_host + "."
spooler_log.info( msg );
spooler_job.state_text = msg;
ftp.binary();
ftp.passive();
ftp.cd(ftp_dir);
list = ftp.dir();
msg = "ftp current working directory: " + ftp.pwd() +
      " containing " + list.length + " files.");
spooler_log.info( msg );
spooler_job.state_text = msg;

```

Should the FTP directory be empty, then `spooler_open()` returns `false`, and `spooler_process()` – which is no longer necessary, as no files to be processed are present - will not be called:

```

return (list.length > 0);
}

```

Should `spooler_open()` return `true`, then `spooler_process()` will be repeatedly called until it returns `false`.

```

function spooler_process() {
  if( steps < list.length ) {
    steps++;
    try {
      spooler_log.debug3("retrieving file: " + list[i]);
      ftp.getFile(list[i], lokal_dir + list[i]);
      cnt_success++;
    } catch(err) {
      spooler_log.warn( "error at download " + list[i] + ": " + err.message );
    }
    return true;
  }
  return false;
}

```

`spooler_close()` will always be called after `spooler_open()` or `spooler_process()`:

```

function spooler_close() {
  spooler_job.state_text = cnt_success + " file(s) successfully processed";
  if( ftp != null ) { ftp.logout(); }
}

```

`spooler_on_error()` is called when an error occurs. The `spooler_task.error` method can be used to hand an error on to the JOB SCHEDULER by way of `spooler_log.error()`:

```

function spooler_on_error() {
  spooler_log.error( "ftp command could not be processed: " + spooler_task.error );
}

```

7.6 Create an External Script File Referenced in the XML Configuration

Should the calls described above be saved in the "jobs/ftp_calls.js" file, then the job can be made aware of this file as follows:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<spooler>
  <config>
    <jobs>
      <job name          = "ftp_get"
           title        = "Get files from ftp server">
        <params>
          <param name = "ftp_host" value = "localhost"/>
          <param name = "ftp_user" value = "anonymous"/>
          <param name = "ftp_pass" value = "anonymous@localhost"/>
          <param name = "ftp_dir"  value = "/test"/>
          <param name = "lokal_dir" value = "./"/>
        </params>
        <script language = "JavaScript">
          <include file = "jobs/ftp.js"/>
          <include file = "jobs/ftp_calls.js"/>
        </script>
      </job>
    </jobs>
  </config>
</spooler>
```

The "jobs/ftp_calls.js" has the following content:

```
var ftp      = null;
var list     = new Array();
var steps    = 0;
var cnt_success = 0;

function spooler_open() {
  var ftp_host = spooler_task.params.var("ftp_host");
  var ftp_user = spooler_task.params.var("ftp_user");
  var ftp_pass = spooler_task.params.var("ftp_pass");
  var ftp_dir  = spooler_task.params.var("ftp_dir");
  var lokal_dir = spooler_task.params.var("lokal_dir");

  ftp = new Ftp(ftp_host);
  ftp.login(ftp_user, ftp_pass);
  var msg = "ftp connection successful for " + ftp_user + "/*:*@* + ftp_host + "."
  spooler_log.info( msg );
  spooler_job.state_text = msg;
  ftp.binary();
```

```
ftp.passive();
ftp.cd(ftp_dir);
list = ftp.dir();
msg = "ftp current working directory: " + ftp.pwd() +
      " containing " + list.length + " files.");
spooler_log.info( msg );
spooler_job.state_text = msg;

return (list.length > 0);
}

function spooler_process() {
  if( steps < list.length ) {
    steps++;
    try {
      spooler_log.debug3("retrieving file: " + list[i]);
      ftp.getFile(list[i], lokal_dir + list[i]);
      cnt_success++;
    } catch(err) {
      spooler_log.warn( "error on download " + list[i] + ": " + err.message );
    }
    return true;
  }
  return false;
}

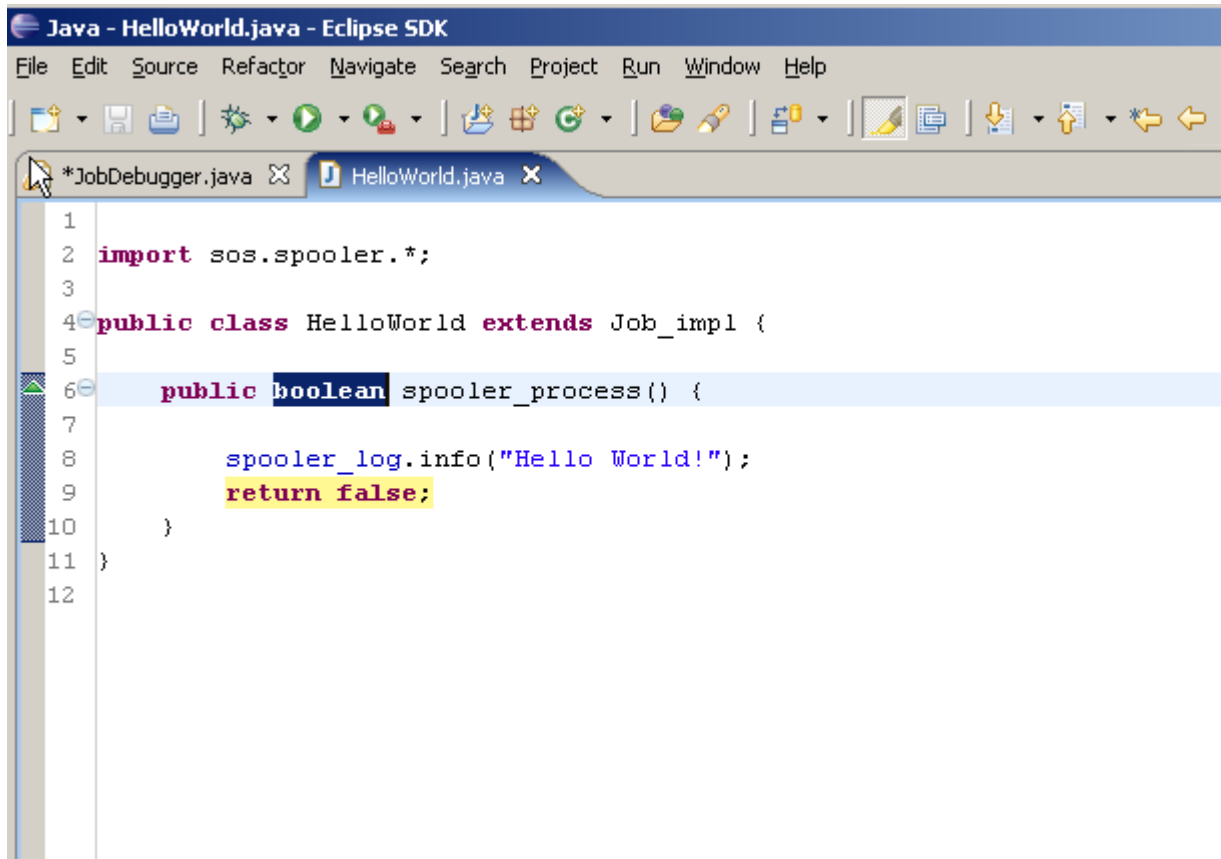
function spooler_close() {
  spooler_job.state_text = cnt_success + " file(s) successfully processed";
  if( ftp != null ) { ftp.logout(); }
}

function spooler_on_error() {
  spooler_log.error( "ftp command could not be processed: " + spooler_task.error );
}
```

8 Debugging Jobs in a Java IDE

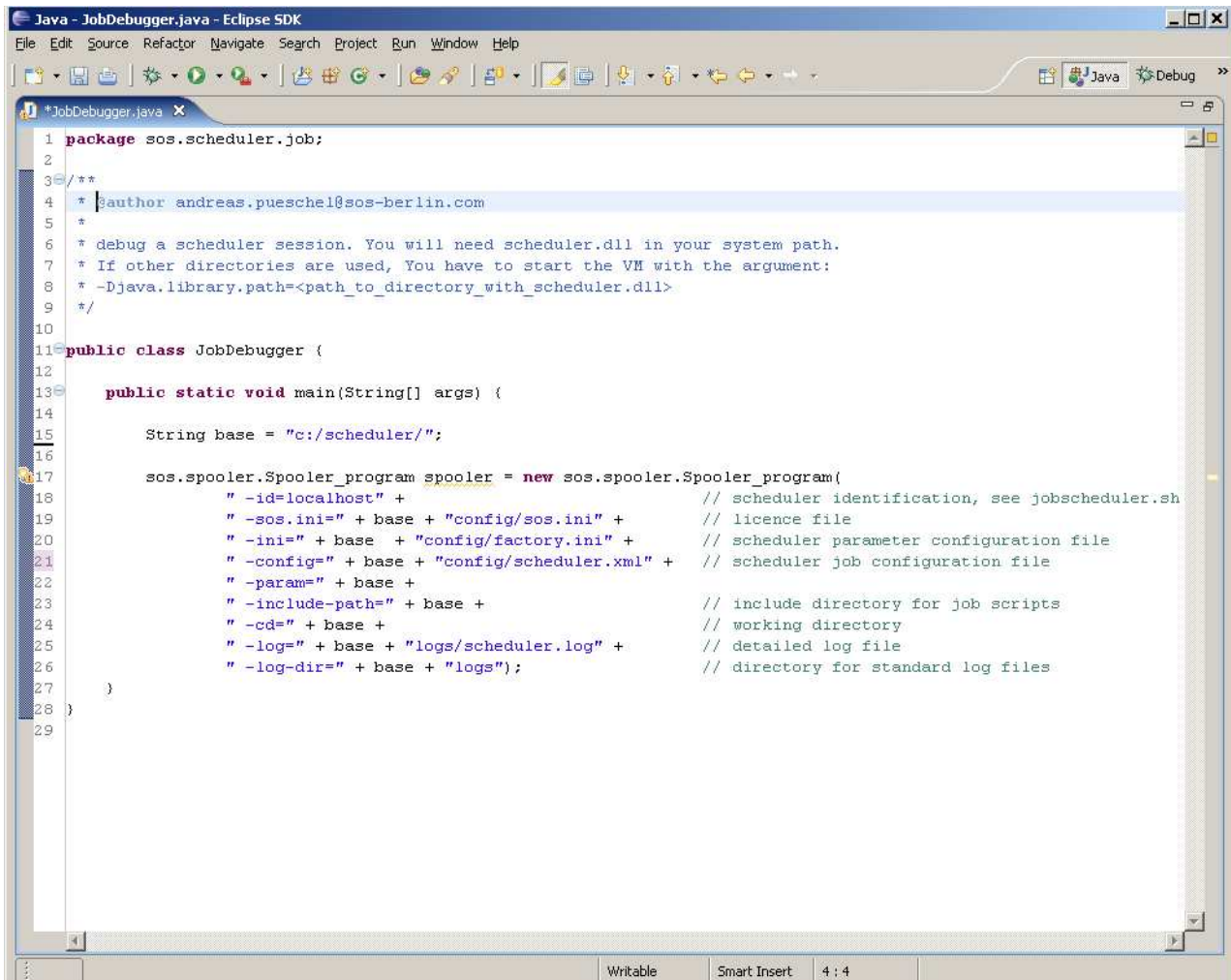
The JOB SCHEDULER supports debugging of Jobs in your IDE, e.g. Eclipse. At the time of writing this support is restricted to Windows Workstations. For debugging follow these steps in your development cycle:

1. Add the Java archive `sos.spooler.jar` from the installation directory `lib` to your IDE project.
2. Create a Java Class that extends the base class `sos.spooler.Job_impl` for your implementation:



```
Java - HelloWorld.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
*JobDebugger.java x HelloWorld.java x
1
2 import sos.spooler.*;
3
4 public class HelloWorld extends Job_impl {
5
6     public boolean spooler_process() {
7
8         spooler_log.info("Hello World!");
9         return false;
10    }
11 }
12
```

3. Create a Java class that instantiates the JOB SCHEDULER in the IDE. The constructor takes all the arguments that are equally used in the start script `jobscheduler.cmd` in the `bin` directory. See http://www.sos-berlin.com/doc/en/scheduler.doc/command_line.xml for a detailed explanation of the command line arguments. Your Java class might look like this:



```

1 package sos.scheduler.job;
2
3 /**
4  * |author andreas.pueschel@sos-berlin.com
5  *
6  * debug a scheduler session. You will need scheduler.dll in your system path.
7  * If other directories are used, You have to start the VM with the argument:
8  * -Djava.library.path=<path_to_directory_with_scheduler.dll>
9  */
10
11 public class JobDebugger {
12
13     public static void main(String[] args) {
14
15         String base = "c:/scheduler/";
16
17         sos.spooler.Spooler_program spooler = new sos.spooler.Spooler_program(
18             "-id=localhost" + // scheduler identification, see jobscheduler.sh
19             "-sos.ini=" + base + "config/sos.ini" + // licence file
20             "-ini=" + base + "config/factory.ini" + // scheduler parameter configuration file
21             "-config=" + base + "config/scheduler.xml" + // scheduler job configuration file
22             "-param=" + base +
23             "-include-path=" + base + // include directory for job scripts
24             "-cd=" + base + // working directory
25             "-log=" + base + "logs/scheduler.log" + // detailed log file
26             "-log-dir=" + base + "logs"); // directory for standard log files
27     }
28 }
29

```

4. Add a launch configuration to your IDE that runs the above class.
5. Copy the file `scheduler.dll` from the installation directory `lib` to your system path or add the following definition to the VM arguments of your Java launch configuration and:


```
-Djava.library.path=c:/scheduler/lib/scheduler.dll
```

Take into account that the JOB SCHEDULER loads the xml configuration file (`-config=xmlfile`) and starts the jobs at the run time given in the configuration file. To run or debug just one job it is recommended to create a separate configuration file for debugging, for example:

```
<process_classes ignore="yes"/>
<job name      = "debug_job">
  <script language = "java"
        java_class = "com.example.job.Helloworld"/>
  <run_time once = "yes"/>
</job>
```

The run time attribute `once = "yes"` starts the job immediately.

Please include the `<process_classes ignore="yes"/>` element in your configuration file: to debug a job in your IDE you have to prevent the JOB SCHEDULER from starting a separate process for your job. Instead the job should run in the same process with the JOB SCHEDULER in your IDE which can be achieved by setting the `ignore="yes"` attribute.