There are still 5 <todo> elements left in this document.

Internal API-Jobs

A Programming Tutorial
June 2012

Contact Information

Software- und Organisations-Service GmbH

Giesebrechtstr. 15
D-10629 Berlin
Germany

Telephone +49 (0)30 86 47 90-0
Telefax +49 (0)30 8 61 33 35
Mail info@sos-berlin.com
Web http://www.sos-berlin.com

Last Updated: 06/12/2012 06:34 PM

This documentation is based on JobScheduler Version 1.3.12.2137.

We would appreciate any feedback you have, or suggestions for changes and improvements; please forward your comments to info@sos-berlin.com.

Table of Contents

1 Introduction

Why do we talk about the "internal" API? Is there an "external" API as well? The answer is yes. **JobScheduler** has an internal as well as an external API. The external API is used to communicate with the **JobScheduler** from other Applications by using Network functionality like TCP/IP or HTTP. This "external" API is described in the documentation xml_commands.

This tutorial describes the implementation and coding of an internal API job with the **JobScheduler**.

A job is the content of a <job> element in a **JobScheduler**'s configuration file. The element specifies the name, title, program code to execute, time slot and start time for a job. The program code of the job is referred in the chapters below as "script", but if we for example are talking about Java it is not a script, it is compiled byte-code. /config/live All configuration files are stored in the `/config/live` /config/live folder or subfolders of the `live`-folder.

Further information can be found in the following documents:

* Technical Documentation (Online Documentation) Can be found at either

```
[Installation directory of the JobScheduler]/config/html/doc/de/index.html
```

or via

```
http://[JobSchedulerhost]:[port]/doc/en/index.html
```

* API Documentation. Can be found at either

```
          [Installation directory of the JobScheduler]/config/html/doc/en/api.xml
```

or via

```
          http://[JobSchedulerhost]:[JobSchedulerTcpPort]/doc/en/api/api.xml
```

The documentation in PDF and HTML format can be found at

```
          [Installation directory of the JobScheduler]/doc/en/scheduler_api.pdf
```

and

```
          [Installation directory of the JobScheduler]/doc/en/scheduler_api/sos_help.htm
```

2 Summary

The organization of the program code for a job for the **JobScheduler** is described in Chapter 3.

In Chapters 4-6, the **JobScheduler** 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 language used for this job is JavaScript™ (see JavaScript). Similar jobs could be written in Java, Java Perl Perl or VBScript VBScript.

3 Communicating the Job Script to the JobScheduler

There are three possible ways of making the **JobScheduler** aware of a job’s program code (sometimes called "the script"):

* incorporation of the script directly in the job configuration file of the job as a text or a script,

* including a reference to a file in the job configuration file. The file contains the code which has to be executed during the run of a job (a task).

* including a reference to the script file in the job configuration file and calling the function of the referenced script in the configuration.

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

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

The 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 an included script

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

Example: JavaScript: HelloWorld

This is the explanation for this example.

4 An example of an external reference to a script

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

Example: including hello_world.js

The "real" location of the file is relative to the **JobScheduler** installation folder.

Alternatively the Java version:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<job >
   <script language   = "Java"
            java_class = "scheduler.job.HelloWorld"/>
</job>
```

Example: including scheduler.job.HelloWorld

It is important to know that no source code is included for Java programs. The program code must be compiled and placed into a .jar-file. The location of this .jar-file has to be defined in the class-path.

Functions can be implemented in external code snippets, 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;
}
```

Example: Using functions

In the same way, the Java class "HelloWorld.java" can be implemented as shown below:

```
package com.sos.api-examples;

import sos.scheduler.job.JobSchedulerJobAdapter;
import com.sos.JSHelper.Exceptions.JobSchedulerException;
import sos.spooler.Spooler;

public class HelloWorld extends JobSchedulerJobAdapter throws Exception  {
    public boolean spooler_process() throws Exception  {
  try {
   super.spooler_process();
   spooler_log.info("Hello, World!");
  }
  catch (Exception e) {
   spooler_log.error(e.getMessage());
   throw new JobSchedulerException(e.getMessage());
  }
  finally {
  } // finally

  return signalSuccess();

    } // public boolean spooler_process()
}  // public class HelloWorld
```

Example: source code of HelloWorld.java

4.1 Reference to an External Script with Subsequent Call of a Method

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

Example: reuse code snippets

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

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

Example: log_info function

5 Implementation

5.1 Job implementation with Java

You are free to implement any job with Java. The **JobScheduler** 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 **JobScheduler** exposes, for example, for logging or mail processing.

5.1.1 Simple implementation

To make the job available to the **JobScheduler** follow these steps:

Um einen Java-API Job zu implementieren gehen Sie am einfachsten wie folgt vor:

Create the Java class file and add it to a Java archive. Put this archive to a folder which are named in the Java class path.

Modify the configuration file factory_ini that is located in the directory `config` of the installation: add the path to the Java archive to the entry class_path in the section `[java]`. Use a semicolon ";" as separator for archive paths on a Microsoft Windows™ platform, use a colon ":" as separator for paths on Unix platform. For more information on this file see factory_ini.

An example:

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

Add the job definition as a file named "HelloWorld.job.xml" to the `live`-folder of the **JobScheduler** installation. You should use **JOE** to create and save this job.

A simple Java job definition for the class `com.example.job.HelloWorld` that is contained in the archive `/mypath/sample.jar` might look like this:

```
   <job >
      <script language    = "Java"
               java_class = "com.example.job.HelloWorld"/>
   </job>
```

If you want the job to be started automatically then add a <run_time> element to the job configuration.

For more information of the job configuration see the documentation that is located in the directory `docs`.

5.1.2 Using the API with Java

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 the **JobScheduler** installation. One have to include this jar-file in the classpath of the Java IDE project and have to import the `sos.spooler.Job_impl`.

For further informations of the deployment of the job follow the steps in the preceding 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 **JobScheduler** more control on the behaviour of your job.

You can use all the objects and methods exposed by the **JobScheduler** API. A detailed documentation is included in your installation directory `docs` in the format XML, PDF and HTML. A reference is the API documentation.

5.2 Job implementation with other languages

The script languages VBScript™, Perl and JavaScript™ can be used to implement an API job.

5.2.1 JavaScript

JavaScript is available for all platforms which are supported by **JobScheduler**. The **JobScheduler** has implemented the JavaScript implementation spidermonkey.

5.2.2 Perl

Perl is available for Unix/Linux platforms and for the Microsoft Windows™ platform that use the ActivePerl™, a perl port, of ActiveState.

5.2.3 VBScript

VBScript is available for Microsoft Windows™ only.

5.3 javax scripting languages

With the package javax.script the Java™ installation provides JSR 223: Scripting for the Java Platform API classes. Beside the Java™ build-in implementation for JavaScript (Rhino-Engine) this Scripting Framework supports a lot of third-party Script-Engines, such as groovy or jython (please refer to https://scripting.dev.java.net/ for details).

JobScheduler has an interface to support this framework. Thus it is able to run scripts of a lot of different scripting languages, but the main target of this feature is to replace the JavaScript implementation of spidermonkey in medium-term (see Differences of the JavaScript implementation for details).

5.3.1 How to define a job using javax.script?

It is very simple to define a job using a script language supported by the javax.script package.

You only have to do two trivial steps:

1. Put the libraries of your preferred script language in your classpath.

   It is not necessary if you use rhino (javascript), because it is build-in in (Oracle/Sun) java. Please refer to Requirements for different script languages for other languages below in this document.

2. Write an internal API-job using the script language.

   The essential declaration to tell **JobScheduler** to use the java-based script execution is the language attribute of the job element. You have to specify the script language you want to use in the form javax.script:< languageid>, where <languageid> is the key of the script language (see Requirements for different script languages).

   Here is an example for the declaration of a JavaScript job:

```
<job>
  …
  <script language="javax.script:javascript">
    Print("hello world");
  </script>
</job>
```

5.3.2 Technical aspects

To create a script for the javax.script interface you have to use an API identical to the API for Java™ jobs. That is an important aspect, because you can not write code like spooler_task.error

```
spooler_task.error = "this is an error"
```

in your script anymore. You have to use the Java™ syntax instead:

```
spooler_task.set_error(“this is an error”);
```

It is recommended that you **not** use *spooler_log* anymore, because it is possible that this object is not supported in future versions of **JobScheduler**. Use *logger* instead (see Logging integration for details).

The Java™ side of the implementation of *javax.script* for the **JobScheduler** is part of the project *com.sos.scheduler.egine.kernel*.

Please refer to package *com.sos.scheduler.engine.kernel.scripting* for details.

5.3.3 Logging integration

The Java™ plugin for running script-jobs produce some log messages for analyzing purposes. Thus we use for logging, it is possible to configure it via a properties file.

The messages of the script itself will be forwarded to this logger instance, too. To point up this fact we provide a special object named logger to each script. It is an instance of *org.apache..Logger*, that means that you can configure your log with all capabilities of appenders.

It is still possible to use the object spooler_log (the internal logging service of JS), but it is **not recommended**.

The assumption for using a properties file for the job scheduler is that it is placed in the classpath of the JVM.

You can configure the classpath in the scheduler configuration file *factory.ini*, e.g.

```
[java]
class_path              = C:\Programme\scheduler\lib\*.*
```

You have to put the .properties file in the folder C:\Programme\scheduler\lib.

If you want to redirect the log messages into the task log of the running job you have to use the special appender *com.sos.JSHelper.Logging.JobSchedulerAppender* build for **JobScheduler**. The following sample will show a configuration for this purpose:

```
# assign the appender for the scripting-api
# please notice: if appender scheduler is used the log-entries of the class
# will be placed in the task-log of the scheduler job
<log4j/>.logger.com.sos.scheduler.engine.kernel.scripting=debug, scheduler

<log4j/>.appender.scheduler=com.sos.JSHelper.Logging.JobScheduler<log4j/>Appender
<log4j/>.appender.scheduler.layout=org.apache.<log4j/>.PatternLayout
<log4j/>.appender.scheduler.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n
```

5.3.4 Differences of the JavaScript implementation

The implementation of spidermonkey is a special implementation for the **JobScheduler**. This solution is inflexible and not up to Date, therefore it is marked as deprecated and will not supported anymore.

The following table gives you an overview of the differences of the Rhino implementation of javascript towards to the spidermonkey implementation:

	Rhino	**Spidermonkey**
used Scheduler API	Java	Javascript

Property assignment	Via setter & getter:
	object.setProperty(“val”)
	val = object.getProperty()
		Direct:
		object.property = val
		val = object.property

Reserved words (see below)	Not possible	Possible
Available objects (direct)	logger spooler spooler_job spooler_task	spooler_log spooler spooler_job spooler_task
Logging service		Scheduler internal

The Rhino implementation of JavaScript does not allow the usage of any reserved word in the script context. For instance the word “delete” is reserved and could not use in a code like this:

```
var file = new java.io.File(files[i]);
file.delete();
```

Please use the following syntax:

```
var file = Packages.java.io.File(files[i]);
file["delete"](); // because delete is reserved in javascript
```

5.3.5 Requirements for different script languages

With exception of javascript every script language needs two libraries. In general they must be available in the classpath you have specified for the **JobScheduler**. In a standard installation it is the lib folder of the scheduler home directory and it is recommended that you put the libraries you need into them.

The first library you need contains the engine and factory implementation for the language and is available in the JS223-engines. The package currently provides engines for 24 different scripting languages.

The second one is the implementation of the script language itself. Please follow the link under “Necessary Libraries” for your preferred script language. If you would like to use a script language that is not described below the java.net homepage is a good choice for inspecting the requirements for them.

5.3.6 Examples

5.3.6.1 Mozilla Rhino (aka JavaScript implementation)

Rhino, Mozilla Language identification: javax.script:rhino
Necessary libraries: (none – provided with the sun jre)
Homepage: http://www.mozilla.org/rhino/

```
<job title="Example Rhino API job (javascript)"
     order="no">
     <params>
           <param name="param1" value="value of param1" />
     </params>
    <script language="javax.script:rhino">
        var cnt;
        function spooler_init() {
            cnt = 0;
           logger.info( "spooler_init called" );
           return true;
        }

        function spooler_open() {
           logger.info( "spooler_open called" );
           return true;
        }

        function spooler_close() {
           logger.info( "spooler_close called" );
           return true;
        }

        function spooler_process() {
           if (cnt < 5) {
              logger.info( "spooler_process called (" + ++cnt + ") times." );

              var params = getParameter();

              if (params != null) {
                var names = params.names().split(";");

                for (var i in names)  {
                    logger.info ("Parameter " + names[i] + " = " + params.value(names[i]));
                }
                // create an additional parameter
                spooler_task.params().set_var ("p" + cnt.toString(), "Value of parameter " + cnt.toString());
              }
              logger.info("----------------------------------------------");
              return true;  // continue run, continue with next call to process
           }
           return false;  // end run, continue process with exit and close
        }

        function spooler_on_success() {
           logger.info("spooler_on_success called");
           return true;
        }

        function spooler_exit() {
           logger.info("spooler_exit called");
           return false;
        }

        function spooler_on_error() {
            logger.error("error during job-run: " + spooler_task.error);
            return true;
        }

        function getParameter () {
      var params = spooler.create_variable_set();
      var taskParams = spooler_task.params();
      if (taskParams != null) {
         params.merge(taskParams);
      }
      var order = spooler_task.order();
      if (order != null) {  // to avoid TypeError: "order has no properties in line 31, column 1,"
         params.merge(order.params);
      }
            return params;
        }
    </script>

<run_time/>
</job>
```

Example: Rhino: Template of an API-Job

```
<monitor name="parseResult" ordering="1">
<script language="javax.script:javascript">
// a monitor prevent executing process multiple times (until return = false). bug or feature?
// if spooler_process_before/after defined this bug is not seen.

    // http://www.sos-berlin.com/doc/en/scheduler.doc/api/Monitor_impl-javascript.xml#method__spooler_task_after
        function spooler_task_after(){
      logger.info("spooler_task_after called");
            var exitCode = spooler_task.exit_code;
            var order = spooler_task.order;
            var result = true;
//          var result = false;
            return result;
        }

        // http://www.sos-berlin.com/doc/en/scheduler.doc/api/Monitor_impl-javascript.xml#method__spooler_task_before
        function spooler_task_before(){
            logger.info("spooler_task_before called");
            var result = false;  // end processing
            var result = true;  // continue processing
            return result;
        }

        // http://www.sos-berlin.com/doc/en/scheduler.doc/api/Monitor_impl-javascript.xml#method__spooler_process_after
        function spooler_process_after(){
            logger.info("spooler_process_after called");
            var exitCode = spooler_task.exit_code;
            var order = spooler_task.order;
            var result = true;
//          var result = false;
            return result;
        }

        //
http://www.sos-berlin.com/doc/en/scheduler.doc/api/Monitor_impl-javascript.xml#method__spooler_process_before
        function spooler_process_before(){
      logger.info("spooler_process_before called");
            var result = false;  // end processing
            var result = true;  // continue processing
            return result;
        }
</script>
</monitor>
```

Example: Rhino: Template of an API-Job

Example: Rhino: Template of an API-Job

Example:

5.3.6.2 Groovy

Language identification: javax.script:groovy
Necessary libraries: groovy-all-1.8.4.jar, groovy-engine.jar
Homepage: http://groovy.codehaus.org/
See also: JSR 223 Scripting with Groovy

Sample:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<job title="Testjob groovy" order="no">
    <script language="javax.script:groovy">
```

```
public cnt;

public boolean spooler_init() {
  cnt = 0;
  logger.info("START of Test -------------------------------------------------");
  logger.info("spooler_init is called by " + spooler_job.name() );
  return true;
}

public boolean spooler_process() {
  if (cnt < 5) {
    cnt++;
    logger.info("spooler_process: iteration no " +  cnt);
    return true;
  }
  return false;
}

public boolean spooler_exit() {
  logger.info("END of Test ---------------------------------------------------");
  return true;
}
    </script>
    <run_time/>
</job>
```

5.3.6.3 Jython

Language identification: javax.script:jython
Necessary libraries: jython.jar , jython-engine.jar
Homepage: http://www.jython.org/

Our test refers to Jython 2.2.1. The jar file you need is provided with the installer jython_installer-2.2.1.jar.

Sample:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<job title="Testjob jython" order="no">
    <script language="javax.script:jython">
global cnt

def spooler_init():
  global cnt
  cnt = 0
  logger.info('START of Test -------------------------------------------------');
  logger.debug('spooler_init is called by ' + spooler_job.name() )
  return True;

def spooler_process():
  global cnt
  if cnt < 5:
    cnt = cnt + 1
    logger.info('spooler_process: iteration no ' +  str(cnt) )
    return True;
  return False;

def spooler_exit():
  logger.info('END of Test ---------------------------------------------------');
  return True;
    </script>
    <run_time/>
</job>
```

5.3.6.4 Beanshell

Language identification: javax.script:bean
Necessary libraries: bsh-2.0b5.jar, bsh-engine.jar
Homepage: http://www.beanshell.org/

Sample:

```
<job title="Testjob beanshell" order="no">
    <script language="javax.script:bsh">
 for (int i=0; i<5; i++)
  print(i);
    </script>
    <run_time/>
</job>
```

6 JobScheduler Objects

The most important objects of the **JobScheduler** internal API are:

* spooler: object for the methods of the **JobScheduler**

* spooler_log: object for logging and protocols

* spooler_job: the implementation of the job object.

* spooler_task: the implementation of the 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 more detailed and complete overview can be obtained in the documentation for the Java, COM or Perl interface.

7 Methods of the Job_Impl class

Every internal API job can implement some methods, independent of the language, if the language supports OO and if **JobScheduler** can process this language in an OO way. These methods are called by the **JobScheduler** when the job has been started and a task is running. These 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 but at least a spooler_process should be implemented.

7.1 spooler_init

spooler_init is particularly suited for setting up of objects, database connections, or something like that.

spooler_init is called once for a task after the task 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().

7.2 spooler_open

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

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.

7.3 spooler_process

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 **JobScheduler** interface and that each task can be monitored on a step by step basis.

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 **JobScheduler**.

7.4 spooler_close

spooler_close is particularly suited for closing connections (a database, FTP server, etc.) which may be opened in other method-calls.

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 by the **JobScheduler** after spoo- ler_close().

7.5 spooler_on_success

spooler_on_success is called after spooler_close, should no error have occurred.

7.6 spooler_on_error

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

7.7 spooler_exit

spooler_exit is called immediately after a script is closed.

spooler_exit is particularly suited for cleaning up remaining objects.

7.8 Rules for Jobs in Jobchains

Jobs that are used in job chains, sometimes names "order jobs", need to be configured to process orders. This must be done by using the attribute `<job order="yes">`. For job implementation this results in the following consequences:

7.8.1 spooler_task.order

A spooler_task.order object, or simply named <order>, is available in spooler_process.

7.8.2 Return value of spooler_process

The return value of order jobs from spooler_process determines if an order was successful executed or not. Value True: The order will be continued with the next_state of the current job chain node. Value False: The order will be continued with 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 does not have any effect.

7.8.3 Execution Sequence

The execution sequence of the job methods basically remains the same. However, spooler_process will be called by the **JobScheduler** repeatedly if the job processes multiple orders without ending in between (for example if the idle_timeout is greater than the idle time between two orders).

For a job which processes three orders without ending inbetween, the methods of Job_impl will be called as shown below:

```
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.

8 Example Job: FTP Download

This is just an example. We suggest to use the Receive-job of the JS Integrated Template Library. This is ready to use and has a lot of more usefull functionality.

This Job downloads files from a 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 (sos_ftp.js). 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 sos_ftp.js file has been referenced in the **JobScheduler** 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"?>
            <job
                 title            = "Get files from ftp server">
                <script language  = "JavaScript">
                    <include file = "sos_ftp.js"/>
                    <![CDATA[
                    <!----------------Calls are inserted here------------------------>
                    ]]>
                </script>
            </job>
```

The configuration 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.

8.1 Establish a Connection to an FTP Server

The following lines establish a connection to a 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 **JobScheduler** user interface using the **JobScheduler** objects (see Chapter 5):

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

Closing of the FTP connection:

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

8.2 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 **JobScheduler** 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 **JobScheduler** 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 **JobScheduler** user interface

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

8.3 Convert Script Parameters into Job-Parameters

The ftp_host, ftp_user, ftp_pass, ftp_dir and local_dir script variables are defined as job parameters in this section. To this end, a <params> element is added into the <job>- element. 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 configuration.

Should, for example, the FTP connection data changed, it is not necessary to make this changes in the coding - instead the job parameters in the configuration merely need to be changed. In addition, this give the opportunity of customizing the parameters for a job via an user interface, like **JOE**.

```
             <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>
```

The calls are presented below. In comparison with the previous section, the configured parameters are now accessed using the spooler_task **JobScheduler** 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();
```

8.4 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.

The variable `ftp` 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 **JobScheduler** 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 **JobScheduler** 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) {}
}
```

8.5 Use of the JobScheduler Methods

The methods of the **JobScheduler** (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 – for example 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 **JobScheduler** 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 **JobScheduler** by way of spooler_log.error():

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

8.6 Create an External Script File Referenced in the 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:

```
<job
     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>
```

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 );
}
```

9 Debugging Jobs in a Java IDE

The **JobScheduler** supports debugging of internal API jobs in your IDE, for example Eclipse™ (read more about Eclipse). At the time of writing this paper debugging of jobs is restricted to the Microsoft Windows™ platform. Debugging on Unix/Linux is not possible in this way.

For to setup a debugging environment follow these steps:

Add the latest version of the Java archive `sos.spooler.jar` from the installation directory `lib` to your IDE project.

Create a Java Class that extends the base class `sos.spooler.Job_impl` for your implementation:

```
package com.sos.JSDebugging;
import sos.spooler.Job_impl;
public class HelloWorld extends Job_impl {
    private final String        conClassName    = "HelloWorld";
    private static final String  conSVNVersion = "$Id$";
    public HelloWorld () {        //    }
    public boolean spooler_process () {
        spooler_log.info ("Hello, world ...");
        return false;
    }
}
```

Create a Java class "JSJobDebugger" that instantiates the **JobScheduler** in the IDE. The constructor takes all the arguments that are used int the same way in the start script `jobscheduler.cmd` `jobscheduler.sh`in the bin directory. See command line arguments for an explanation of the command line arguments. The Java class might look like this:

```
package com.sos.JSDebugging;
import org.apache.log4j.Logger;
import sos.spooler.Spooler_program;
public class JSJobDebugger {
    private final String        conClassName    = "JSJobDebugger";
    private static final String conSVNVersion   = "$Id$";
    private static final Logger logger          = Logger.getLogger(JSJobDebugger.class);
    public JSJobDebugger() {        //    }
    public static void main(String[] args) {
        String base =       "./";
        String strConfigBase =      "./JSDebugging/";
        String[] strArgV = new String[]  {"-id=JSDebugger",                    // JobScheduler ID
                  "-config=" + base + "config/JSDebugger.scheduler.xml" ,    // Configuration file
         "-ini=" + strConfigBase + "config/factory.ini" ,                   // another config-file
         "-sos.ini=" + strConfigBase + "config/sos.ini"  ,                  // licence key-file
         "-include-path=." ,
        "-param=" + base ,
        "-log=" + strConfigBase + "logs/scheduler.log" ,
        "-log-dir=*stderr" ,                                       // redirect the log output to the console window of
eclipse
        "-reuse-port",                                             // to avoid 2 minutes wait for releasing the port
        "-env=SCHEDULER_HOME=" + System.getProperty("user.dir") ,  // set environment variable
        "-env=SCHEDULER_DATA=" + System.getProperty("user.dir") ,
        "-cd=" + base
        } ;
        Spooler_program objSP = new Spooler_program(strArgV);
    }
}
```

Add a launch configuration to your IDE that runs the above shown class.

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
```

One must take into account that the **JobScheduler** load its configuration file (`-config=xmlfile`) and starts the jobs at the run time given in the job configuration file. To run or debug just one job it is recommended to create a separate **JobScheduler**-configuration file for debugging, for example:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<spooler>
    <config mail_xslt_stylesheet="config/scheduler_mail.xsl" port="4210">
        <security ignore_unknown_hosts="yes">
            <allowed_host host="0.0.0.0"  level="all"/>
        </security>
        <process_classes ignore="yes"/>
        <http_server>
            <http_directory url_path="/scheduler_home/" path="${SCHEDULER_HOME}"/>
            <http_directory url_path="/scheduler_data/" path="${SCHEDULER_DATA}"/>
        </http_server>
    </config>
</spooler>
```

One must include the <process_classes> element with the attribute ignore="yes" /> in the `scheduler.xml`: to debug a job in an IDE this element prevent the **JobScheduler** from starting a separate process for each job. The job must run in the same process as with the **JobScheduler**-dll and in the IDE which must be achieved by setting the ignore="yes" attribute as shown below.

```
      <process_classes ignore="yes"/>
```

The job definition below is the job which is the job to 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 after starting the debugging-session in the IDE.

If the job comes to an end the **JobScheduler** is still running until you end the debug session in the IDE.

It is possible to use the **JOC** to restart the job and/or to end the **JobScheduler** session. To achive this you must start the **JOC** in a browser an type the server-name (localhost for example) and the port of the debugging **JobScheduler**.

The JSDebugging project is available for download at JSDebugging-project.zip.

Index