JOB SCHEDULER

Tutorial
Job Implementierung

Dokumentation

November 2008

Inhaltsverzeichnis

1 Einleitung 3
2 Überblick 3
3 Job-Script dem Job Scheduler übergeben 4
3.1 Beispiel für Scripting direkt in der XML-Konfiguration 4
3.2 Beispiel für das Referenzieren externer Skripte 5
3.3 Beispiel für das Referenzieren externer Skripte mit anschließendem Methodenaufruf 6
4 Implementierung 6
4.1 Job Implementierung mit Java 6
4.1.1 Minimale Implementierung 6
4.1.2 Verwendung des Java API 7
4.2 Job Implementierung mit Script-Sprachen 7
5 Objekte des Job Schedulers 8
6 Methoden des Job Schedulers 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 Besonderheiten bei Auftrags-Jobs 9
6.8.1 spooler_task.order 9
6.8.2 Rückgabewert von spooler_process() 9
6.8.3 Ausführungsreihenfolge 10
7 Beispiel-Job: Download per FTP 11
7.1 Verbindung zum FTP-Server herstellen 12
7.2 Dateien vom FTP-Server laden 12
7.3 Skript-Parameter in Job-Parameter wandeln 13
7.4 Fehlerbehandlung integrieren 15
7.5 Verwendung der Methoden des Job Schedulers 16
7.6 Skript auslagern und in XML-Konfiguration referenzieren 18
8 Jobs in einer Java IDE debuggen 20

1 Einleitung

Dieses Tutorial beschreibt die Programmierung eines Jobs im JOB SCHEDULER.

Ein Job ist der Inhalt des <job>-Elements in der XML-Konfiguration des JOB SCHEDULERS. Das Element legt für jeden Job den Namen, Titel, Programmcode, Zeitfenster und Startzeitpunkt fest.

Weitere Informationen finden Sie in den Dokumentationen:

- *Technische Dokumentation* (Online Dokumentation)
  Aufrufbar mittels [Installationsverzeichnis des Job Schedulers]/config/html/doc/de/index.html
  oder
  per http://[Host des Job Schedulers]:[Port des Job Schedulers]/doc/de/index.html
- *Java API Dokumentation*.
  Aufrufbar mittels [Installationsverzeichnis des Job Schedulers]/config/html/doc/de/javadoc/index.html
  oder
  per http://[Host des Job Schedulers]:[Port des Job Schedulers]/doc/de/javadoc/index.html

  Dieselben Dokumentationen sind in den Formaten PDF und HTML enthalten in
  [Installation directory of the Job Scheduler]/doc/de/scheduler_api.pdf
  und
  [Installation directory of the Job Scheduler]/doc/de/scheduler_api/sos_help.htm

2 Überblick

Im Kapitel 3 wird beschrieben, wie der Programmcode für einen Job im SCHEDULER aufgebaut wird.

In den Kapiteln 4-6 wird das Scheduler-Objekt mit seinen Methoden und Objekten beschrieben, das in jedem Job verwendet werden kann.

Ab Kapitel 7 wird ein Beispiel-Job vorgestellt. Dieser Job hat die Aufgabe, Dateien von einem FTP-Server abzuholen. Die verwendete Skriptsprache ist JavaScript. Analoge Beispiele können in den Sprachen Java, Perl und VBScript implementiert werden.

3 Job-Script dem JOB SCHEDULER übergeben

Es gibt drei Möglichkeiten dem JOB SCHEDULER den Programmcode eines Jobs bekannt zu geben:

- Skript direkt in die XML-Konfiguration einfügen
- Skript aus externer Datei in der XML-Konfiguration referenzieren
- Skript aus externer Datei in der XML-Konfiguration referenzieren und Funktion des referenzierten Scripts in der XML-Konfiguration aufrufen

Der erste Fall macht die XML-Konfiguration tendenziell unübersichtlich, wenn das Skript umfangreich ist, bei kurzen Skripten ist dies die direkteste Implementierung.

Im zweiten Fall ist die Verarbeitung des Job-Skripts auch ohne zusätzlichen Aufruf einer Funktion möglich, wenn im referenzierten Skript bestimmte Methodennamen des JOB SCHEDULERS verwendet wurden (siehe Kapitel 4).

3.1 Beispiel für Scripting direkt in der XML-Konfiguration

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

3.2 Beispiel für das Referenzieren externer Skripte

```
<?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>
```

alternativ für Java:

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

Um sich den Aufruf diverser Funktionen zu ersparen, können Funktionen in einem externen Skript implementiert sein (siehe Kapitel 4).

Beispielsweise hat "`hello_world.js`" den Inhalt:

```
function spooler_process() {
    spooler_log.info("Hallo Welt!");
    return false;
}
```

Analog kann die Java-Klasse "`HelloWorld.java`" wie folgt implementiert sein:

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

public class HelloWorld extends Job_impl {

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

3.3 Beispiel für das Referenzieren externer Skripte mit anschließendem Methodenaufruf

```
<?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("Hallo Welt!");
                    ]]>
                </script>
            </job>
        </jobs>
    </config>
</spooler>
```

Hier hat `"jobs/log.js"` beispielsweise den Inhalt:

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

4 Implementierung

4.1 Job Implementierung mit Java

Sie können Jobs beliebig mit Java implementieren, der JOB SCHEDULER schränkt Sie nicht in der Implementierung ein. Sie können die Objekte und Methoden des API Interface nutzen, bspw. für Protokollausgaben oder den eMail-Versand von Job-Nachrichten.

4.1.1 Minimale Implementierung

Geben Sie dem JOB SCHEDULER Ihren Job in den folgenden Schritten bekannt:

- Kompilieren Sie Ihren Java Job als class file und fügen Sie ihn einem Java Archiv (.jar) hinzu.
- Passen Sie die Einstellungsdatei `factory.ini` an, die im Verzeichnis `config` des Installationspfads enthalten ist: fügen Sie den Pfad zu Ihrem Java Archiv dem Eintrag `class_path` in der Sektion `[java]` hinzu. Verwenden Sie ; als Trennzeichen zwischen Archiv-Pfaden auf Windows Servern, verwenden Sie : als Trennzeichen zwischen Archiv-Pfaden auf Unix Servern. Beispiel:

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

- Fügen Sie der Konfigurationsdatei `scheduler.xml` im Verzeichnis `config` Ihrer Installation eine Job-Definition hinzu. Eine mimimale Java Job-Definition für eine Klasse `com.example.job.HelloWorld`, die im Java Archiv `/mypath/sample.jar` enthalten ist, könnten etwa so aussehen:

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

- Starten Sie den JOB SCHEDULER neu, um die Job Implementierung zu laden.
- Wenn Sie eine automatische Startzeit für Ihren Job vereinbaren möchten, fügen Sie der Job-Definiiton ein Element `<run_time>` hinzu.. Weitere Informationen zu Job-Definitionen finden Sie in der Dokumentation im Verzeichnis `docs`.

4.1.2 Verwendung des Java API

Ein Java Job erbt von der abstrakten Super-Klasse `sos.spooler.Job_impl`. Die Klasse ist im Archiv `sos.spooler.jar` enthalten, das im Verzeichnis `lib` Ihres Installationspfades liegt. Fügen Sie dieses Archiv bitte Ihrem Java IDE Projekt hinzu und importieren Sie die Klasse `sos.spooler.Job_impl`.

Um den Job dem Job Scheduler bekanntzumachen, folgen Sie bitte den Schritten des vorigen Kapitels.

Sie könnne die abstrakten Methoden der Super-Klasse implementieren, müssen dies jedoch nicht. Diese Methoden (`spooler_init(), spooler_process()` etc.) geben dem JOB SCHEDULER mehr Kontrolle zur Steuerung Ihres Jobs, was in den folgenden Kapiteln näher erläutert wird.

Für Java stehen alle Objekte und Methoden des JOB SCHEDULER API zur Verfügung. Die Dokumentation des API ist in Ihrer Installation im Verzeichnis `docs` in den Formaten XML, PDF und HTML enthalten. Die Online-Dokumentation steht unter http://www.sos-berlin.com/doc/en/scheduler.doc/api/api.xml zur Verfügung.

4.2 Job Implementierung mit Script-Sprachen

Die Script-Sprachen VBScript, Perl und JavaScript können für Implementierungen mit dem API verwendet werden.

- JavaScript ist für alle Plattformen verfügbar, der JOB SCHEDULER enthält die JavaScript Implementierung “spidermonkey”, siehe http://www.mozilla.org/js/.
- PerlScript ist unter Unix und unter Windows mit Hilfe der Portierung von ActiveState (http://www.activestate.com) verfügbar.
- VBScript ist nur für Windows verfügbar.

5 Objekte des JOB SCHEDULERS

Die Objekte des JOB SCHEDULER API sind

- `spooler`: Objekt für Methoden des Job Schedulers
- `spooler_log`: Objekt zur Protokollierung
- `spooler_job`: Objekt des Jobs
- `spooler_task`: Objekt des Prozesses, in dem der Job ausgeführt wird

Beispiele der Eigenschaften und Methoden dieser Objekte werden in diesem Tutorial verwendet und erklärt. Einen vollständigen Überblick erhalten Sie in der entsprechenden Schnittstellen-Dokumentation für Java, COM oder Perl.

6 Methoden des JOB SCHEDULERS

Jede Job-Implementierung kann unabhängig von der Skriptsprache optional Methoden implementieren, die vom JOB SCHEDULER automatisch aufgerufen werden, sofern sie im Skript vorhanden sind. Dies sind:

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

Wann die Aufrufe erfolgen, beschreiben die folgenden Abschnitte. Die Implementierung dieser Methoden ist optional.

6.1 spooler_init()

`spooler_init()` wird einmal nach dem Laden des Skripts gerufen.

Eine Rückgabe von `True`, `1` oder `Empty` (keine Rückgabe) wird als `True` verstanden und lässt die Verarbeitung fortfahren.

Eine Rückgabe von `False`, `0`, `Nothing` oder `Null` wird als `False` verstanden und beendet die Verarbeitung. `spooler_exit()` wird gerufen und das Skript wird geschlossen.

Bei einem Fehler in `spooler_init()` wird mit `spooler_exit()` fortgefahren.

`spooler_init()` eignet sich insbesondere für das Anlegen von Objekten, Datenbankverbindungen etc.

6.2 spooler_open()

Wird zu Beginn einer Task gerufen. Der Rückgabewert wird wie bei `spooler_init()` interpretiert. Ist der Rückgabewert `False` oder es wird ein Fehler ausgelöst, dann wird `spooler_close()` gerufen, anderenfalls wird mit `spooler_process()` fortgefahren.

`spooler_open()` eignet sich insbesondere für das Anlegen einer Objektmenge und/oder das Öffnen einer Verbindung (Datenbank, FTP-Server, etc.).

6.3 spooler_process()

Der Rückgabewert wird wie bei `spooler_init()` interpretiert. `False` lässt die Task mit `spooler_close()` fortfahren, `True` mit einem weiteren Aufruf von `spooler_process()`.

Jeder Aufruf von `spooler_process()` wird in der Scheduler-Oberfläche als Job-Schritt gezählt.

`spooler_process()` eignet sich insbesondere zur schrittweisen Verarbeitung einer Objektmenge, die z.B. im `spooler_open()` angelegt wurde. Die Implementierung von `spooler_process()` bietet den Vorteil den Fortgang der Verarbeitung in der Oberfläche des Job Schedulers verfolgen zu können und die Task bei einem Schrittwechsel kontrolliert beenden zu können.

6.4 spooler_close()

Wird nach einem Fehler oder nachdem die Rückgabe der Methoden `spooler_open()` oder `spooler_process()` `False` liefert, zum Ende einer Task gerufen.

Anschließend wird `spooler_on_success()` oder bei einem Fehler `spooler_on_error()` aufgerufen.

`spooler_close()` eignet sich insbesondere für das Schließen evtl. geöffneter Verbindungen (Datenbank, FTP-Server, etc.).

6.5 spooler_on_success()

Wird nach `spooler_close()` gerufen, wenn kein Fehler vorliegt.

6.6 spooler_on_error()

Wird nach `spooler_close()` gerufen, wenn ein Fehler vorliegt. Eine Task kann auf den Fehler in `spooler_task.error` zugreifen.

6.7 spooler_exit()

Wird unmittelbar vor dem Schließen des Skripts gerufen.

`spooler_exit()` eignet sich insbesondere zum Aufräumen ggf. angelegter Objekte.

6.8 Besonderheiten bei Auftrags-Jobs

Jobs, die in Jobketten verwendet werden sollen, müssen konfiguriert werden, Aufträge zu verarbeiten: `<job order="yes">`. Dies hat einige folgen für die Job Implementierung:

6.8.1 spooler_task.order

In `spooler_process()` ist ein Auftragsobjekt verfügbar.

6.8.2 Rückgabewert von spooler_process()

Der Rückgabewert von Auftragsjobs entscheidet, ob der Auftrag erfolgreich verarbeitet wurde oder nicht.

True: Der Auftrag wird im `next_state` des aktuellen Jobknotens fortgeführt.

False: Der Auftrag wird im `error_state` des aktuellen Jobknotens fortgeführt.

Wird jedoch im Job (mit `spooler_log.error()`) ein Fehler ausgelöst, so wird der Job gestoppt und der Auftrag bleibt am aktuellen Jobknoten stehen. Der Rückgabewert hat in diesem Fall keine Bedeutung.

6.8.3 Ausführungsreihenfolge

Die Ausführungsreihenfolge der Job Methoden bleibt größtenteils gleich. Jedoch kann `spooler_process()` mehrfach aufgerufen werden, falls der Job mehrere Aufträge verarbeitet, ohne sich zwischendurch zu beenden (z.B. weil das `idle_timeout` größer ist als die Zeit zwischen zwei Aufträgen).

Für einen Job der 3 Aufträge verarbeitet ohne sich zwischendurch zu beenden, würden also die folgenden Methoden aufgerufen werden:

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

`spooler_on_success()` und `spooler_on_error()` beziehen sich nicht auf Fehler des Auftrags sondern auf Fehler des Jobs.

7 Beispiel-Job: Download per FTP

Dieser Job hat die Aufgabe, Dateien von einem FTP-Server abzuholen.

Die verwendete Skriptsprache ist JavaScript.

Da JavaScript weder auf einen FTP-Server noch auf das Datei-System Zugriff hat, werden Objekte von entsprechenden Java-Klassen verwendet, um dies zu gewährleisten.

Die Verwendung dieser Java-Klassen ist in einer externen JavaScript-Datei (`"jobs/ftp.js"`) gekapselt. Der Inhalt dieser Datei ist für das Tutorial nicht wesentlich (der Quell-Code ist in der Auslieferung enthalten). Man gehe einfach davon aus, dass ein Objekt der in dieser Datei formulierten Klasse Methoden besitzt, die es ermöglichen z.B. eine Verbindung zu einem FTP-Server aufzubauen oder Dateien in das Datei-System zu schreiben.

Diese JavaScript-Datei wird in der Job-Definition des JOB SCHEDULERS referenziert, anschließend können die enthaltenen Methoden oder Funktionen aufgerufen werden (siehe Abschnitt 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[
 <!----------------Hier kommen die Aufrufe rein----------------------->
                    ]]>
                </script>
            </job>
        </jobs>
    </config>
</spooler>
```

Später wird das XML-Gerüst noch erweitert werden. Bis dahin wird darauf verzichtet das XML-Gerüst wiederzugeben, es werden nachfolgend lediglich die Aufrufe beschrieben.

Mit jedem der folgenden Abschnitte wird das Skript wachsen. Jede neue Skriptzeile wird kommentiert. Die in `unproportionaler` Schrift wiedergegebenen Zeilen sind die Skriptzeilen.

7.1 Verbindung zum FTP-Server herstellen

Im folgenden werden Aufrufe zum Verbinden mit einem FTP-Server vorgestellt.

Verbindungparameter zum FTP-Server festlegen (Hostname des FTP-Servers, Benutzername und Kennwort)

```
var ftp       = null;           // wird später das Objekt der JavaScript FTP-Klasse
var ftp_host  = "localhost";
var ftp_user  = "anonymous";
var ftp_pass  = "anonymous@localhost";
```

Objekt der FTP-Klasse instanziieren mit Hostnamen des FTP-Servers als Argument

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

Verbindung zum FTP-Server auf `ftp_host` mit Benutzernamen `ftp_user` und Kennwort `ftp_pass` herstellen

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

Info ins Log schreiben und Ausgabe für die SCHEDULER-Oberfläche mittels SCHEDULER-Objekten (s. Kapitel 5)

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

Schließen der FTP-Verbindung

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

7.2 Dateien vom FTP-Server laden

Das Skript aus Abschnitt 7.1 wird ergänzt durch Aufrufe zum Navigieren auf dem FTP-Server, dem Einlesen eines FTP-Verzeichnisses und dem Herunterladen von Dateien des gewählten FTP-Verzeichnisses.

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

Weitere Parameter festlegen (FTP-Verzeichnis und lokales Verzeichnis)

```
var ftp_dir   = "/test";
var lokal_dir = "./";     //die Dateien werden im aktuellen Arbeitsverzeichnis gespeichert

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;
```

Übertragungsmodus festlegen ("binary" oder "ascii")

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

Lieferart festlegen ("active" oder "passive")

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

In das FTP-Verzeichnis ftp_dir wechseln

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

Die Dateinamen des FTP-Verzeichnisses ftp_dir sammeln

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

Aktuelles FTP-Verzeichnis mit Anzahl der gefundenen Dateien als Info ins Log schreiben und Ausgabe für die JOB SCHEDULER Oberfläche(s. Kapitel 5).

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

Für jede gefundene Datei einen Debug-Eintrag ins Protokoll mit dem Dateinamen (wird nur geschrieben, wenn der Debug-Level des JOB SCHEDULERS >= 3 ist) und speichern in lokal_dir.

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

Ausgabe für die JOB SCHEDULER Oberfläche

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

ftp.logout();
```

7.3 Skript-Parameter in Job-Parameter wandeln

In diesem Abschnitt werden die Skriptvariablen ftp_host, ftp_user, ftp_pass, ftp_dir, local_dir als Job-Parameter definiert. Hierfür wird innerhalb des <job>-Elements ein <params>-Element eingefügt (siehe Dokumentation zur XML-Konfiguration). Das hat den Vorteil, dass der Job leichter konfiguriert werden kann. Später wird das gesamte Skript in eine externe Datei ausgelagert und in der XML-Konfiguration referenziert.

Falls sich FTP-Verbindungdaten ändern sollten, werden Sie nicht das Skript bearbeiten müssen, sondern lediglich in der XML-Konfiguration die Job-Parameter entsprechend neu einstellen. Überdies eröffnet sich so die Möglichkeit, über eine Oberfläche die Parameter für jeden Job-Lauf editierbar zu machen.

```
<?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[
 <!----------------Hier kommen die Aufrufe rein----------------------->
                    ]]>
                </script>
            </job>
        </jobs>
    </config>
</spooler>
```

Ab hier folgen die Aufrufe. Ergänzend zum vorherigen Abschnitt wird auf die konfigurierten Parameter zugegriffen. Dies erfolgt mittels des SCHEDULER-Objekts `spooler_task` (siehe Kapitel 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 Fehlerbehandlung integrieren

Es werden ergänzend `try`-`catch`-Blöcke mit `finally` eingeführt. Wenn im `try`-Block ein Fehler auftritt, wird der `catch`-Block ausgeführt. Der `finally`-Block wird in jedem Fall als letztes ausgeführt.

Diese Variable muss ausserhalb (global) definiert sein, damit sie in jedem Block bekannt ist.

```
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;
```

Zähler für erfolgreiche 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) {
```

Warnung ins Protokoll schreiben mittels SCHEDULER-Objekt

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

Fehler ins Protokoll schreiben mittels SCHEDULER-Objekt

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

Hier soll unabhängig von etwaigen Fehlern die FTP-Verbindung geschlossen werden.
Die Prüfung `if( ftp != null )` ist nötig, falls bereits die Instanziierung einen Fehler auslöst und damit `ftp` kein Objekt ist, das die Methode `logout()` aufrufen könnte.

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

7.5 Verwendung der Methoden des JOB SCHEDULERS

In diesem Abschnitt werden die Methoden des JOB SCHEDULERS (siehe Kapitel 6) integriert. In der Methode `spooler_open()` wird das FTP-Objekt angelegt, die Verbindungsoptionen gesetzt und die zu verarbeitenden Dateien gesammelt. In `spooler_process()` werden die eingesammelten Dateien schrittweise verarbeitet, d.h. heruntergeladen. In `spooler_close()` wird schließlich die Verbindung zum FTP-Server getrennt.

Auf diese Weise kann die Fehlerbehandlung auf den äußeren `try/catch`-Block des vorherigen Abschnitts verzichten. Insbesondere ermöglicht dies den Job bei Schrittwechseln zu stoppen, fortzusetzen und/oder zu beenden. Die JOB SCHEDULER Web-Oberfläche informiert über den Fortschritt der Verarbeitung durch die Anzeige der aktuell vollzogenen Schritte.

```
var ftp         = null;
```

Es müssen weitere Variablen global bekannt gemacht werden. Diese sind das Array `list`, das die zu verarbeitenden Dateien sammelt, ein Zähler `steps` für die Verarbeitungsschritte und der Zähler `cnt_success` für die erfolgreichen Verarbeitungsschritte.

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

Tritt in `spooler_open()` ein Fehler auf oder liefert `False` als Rückgabe, so wird `spooler_close()` gerufen.

```
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;
```

Ist das FTP-Verzeichnis leer, liefert `spooler_open() false` und damit wird `spooler_process()` nicht aufgerufen, was auch nicht nötig ist, wenn keine Dateien zur Verarbeitung vorliegen.

```
    return (list.length > 0);
}
```

Liefert `spooler_open() true`, dann wird `spooler_process()` aufgerufen und das so oft, bis `spooler_process() false` zurückgibt.

```
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()` wird in jedem Fall nach `spooler_open()` bzw. `spooler_process()` aufgerufen.

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

`spooler_on_error()` wird aufgerufen, wenn ein Fehler aufgetreten ist. Ist der Fehler an den JOB SCHEDULER mit `spooler_log.error()` weitergereicht worden, so steht dieser in `spooler_task.error` zur Verfügung

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

7.6 Skript auslagern und in XML-Konfiguration referenzieren

Speichern Sie die oben beschriebenen Aufrufe in der Datei "jobs/ftp_calls.js", dann können Sie diese Datei für den Job wie folgt bekannt machen.

```
<?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>
```

Die Datei "jobs/ftp_calls.js" hat jetzt folgenden Inhalt:

```
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 );
```

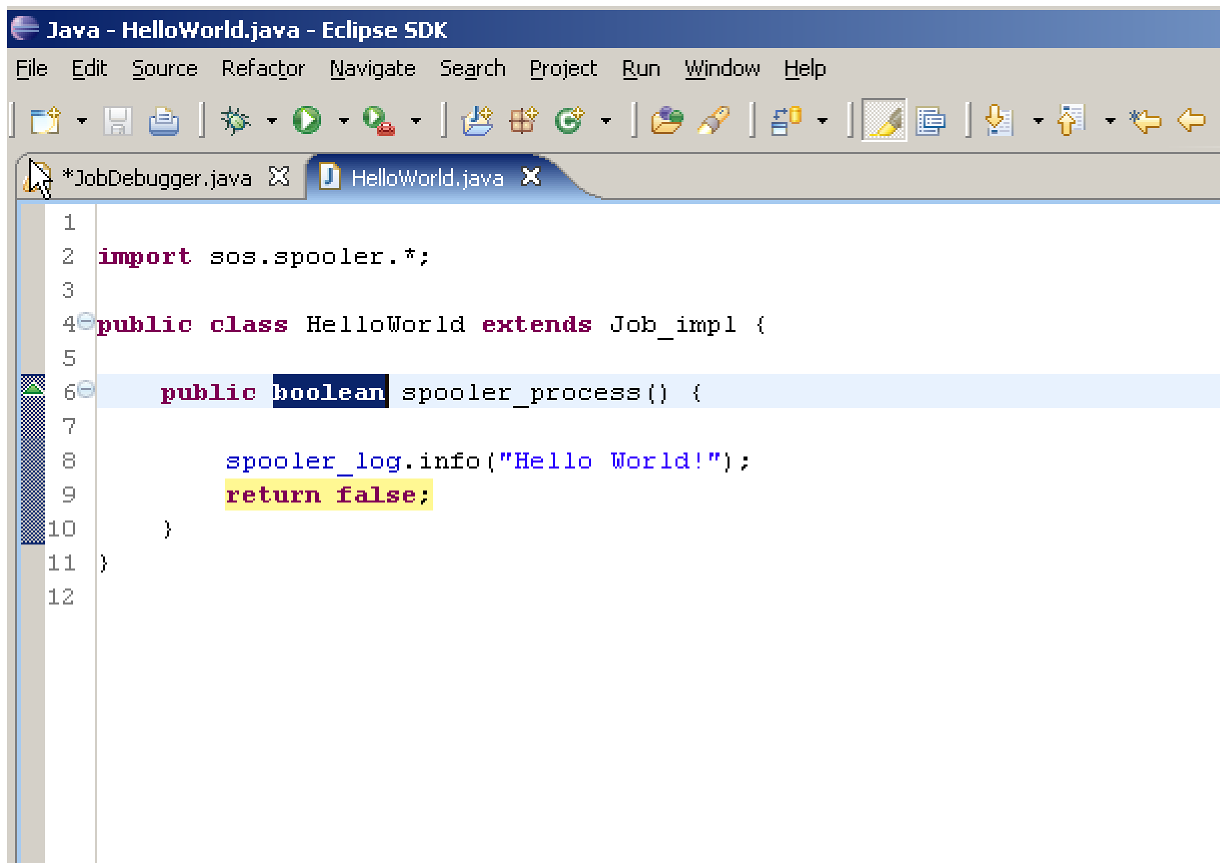
```
    spooler_job.state_text = msg;
    ftp.binary();
    ftp.passive();
    ftp.cd(ftp_dir);
    list = ftp.dir();
    msg = “ftp current working directory: " + ftp.pwd() +
          “ containing “ + list.length + “ files.”);
    spooler_log.info( msg );
    spooler_job.state_text = msg;

    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 Jobs in einer Java IDE debuggen

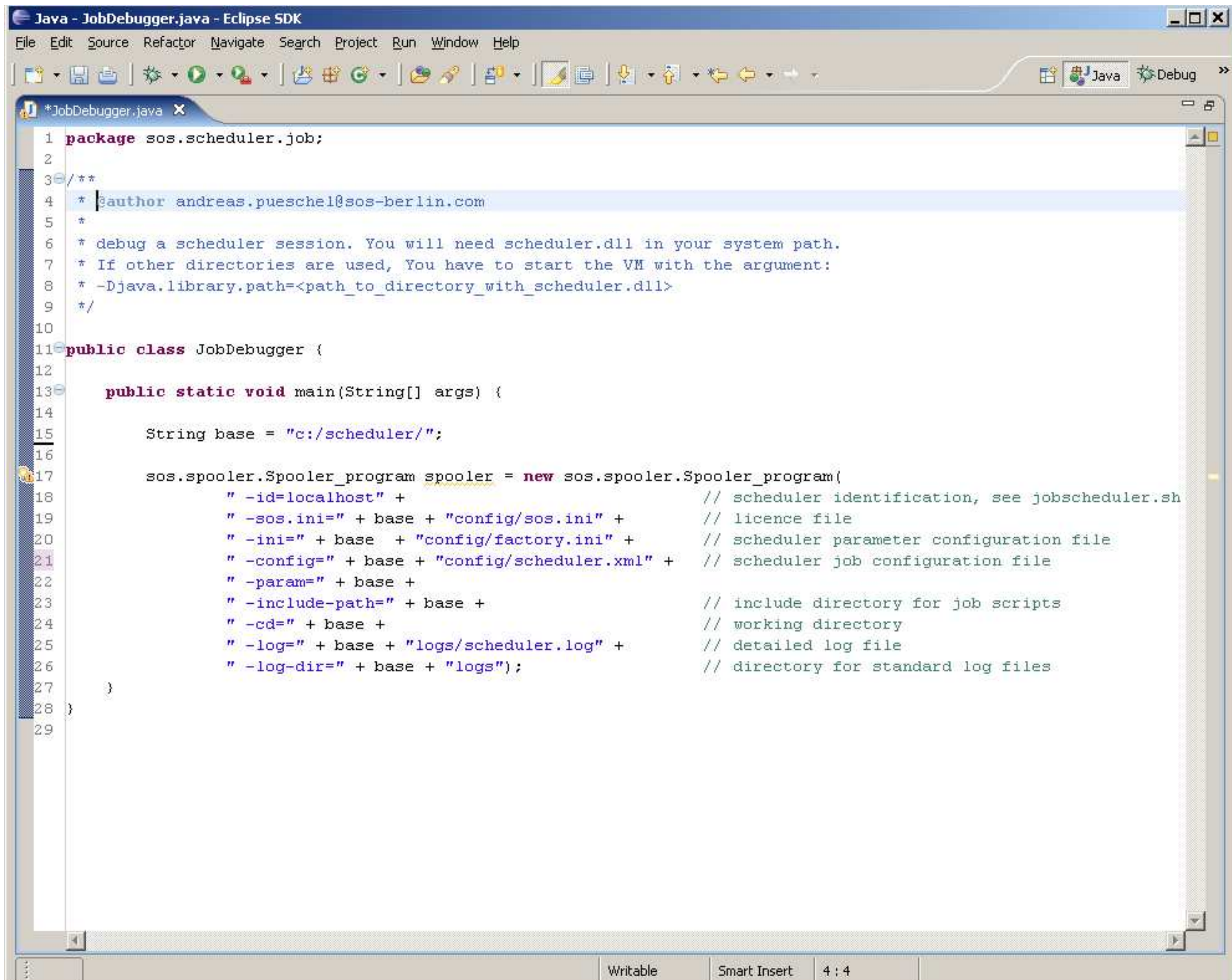
Der JOB SCHEDULER unterstützt das Debuggen von Jobs in Ihrer IDE, z.B. Eclipse. Derzeit ist das Debugging nur für Windows verfügbar. Zum Debuggen folgen Sie bitte diese Schritten im Entwicklungszyklus:

1. Fügen Sie das Java Archiv `sos.spooler.jar` aus dem Installationsverzeichnis `lib` Ihrem IDE Projekt hinzu.

2. Implementieren Sie eine Java Klasse, die von der Basis-Klasse `sos.spooler.Job_impl` erbt:



3. Schreiben Sie eine Java Klasse, die den JOB SCHEDULER in der IDE instantiiert. Der Konstruktor erwartet dieselben Argumente wie sie im Start Skript `jobscheduler.cmd` im Verzeichnis `bin` verwendet werden. Detaillierte Informationen zu den Kommandozeilen-Argumenten des JOB SCHEDULERS finden Sie unter http://www.sos-berlin.com/doc/en/scheduler.doc/command_line.xml. Ihre Java Klasse kann etwa folgendermaßen aussehen:



4. Erzeugen Sie eine launch configuration für Ihre IDE, die die o.g. Klasse ausführt.

5. Kopieren Sie die Datei `scheduler.dll` aus dem Installationsverzeichnis `lib` in Ihren Windows-Pfad oder fügen Sie die folgende Definition als VM Arguments Ihrer Java launch configuration hinzu und passen den Pfad entsprechend an:

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

Beachten Sie bitte, dass der JOB SCHEDULER die xml Konfigurationsdatei lädt (`-config=`*`xmlfile`*) und die Jobs zu dem Zeitpunkt startet, der in der Konfigurationsdatei angegeben ist. Um einen bestimmen Job zu debuggen wird empfohlen eine separate Konfigurationsdatei einzurichten, zum Beispiel mit der Job-Definition:

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

Das Attribut `once = “yes”` des Elements `<run_time>` vereinbart den sofortigen Job-Start.

Fügen Sie bitte das Element `<process_classes ignore="yes"/>` Ihrer Konfigurationsdatei hinzu: um einen Job in Ihrer IDE zu debuggen, darf der JOB SCHEDULER keinen separaten Prozess für den Job starten. Stattdessen muss der Job im selben Prozess wie der JOB SCHEDULER in Ihrer IDE ablaufen, was Sie durch den Parameter `ignore=“yes“` erreichen.