levigo®

jadice server

jadice document platform

Benjamin Geißelmeier

Januar 2010

jadice® server Version 4.2.1.5

Dokumentation für Entwickler

levigo solutions gmbh
Max-Eyth-Straße 30
D-71088 Holzgerlingen
Telefon 0 70 31 / 41 61 - 20
Telefax 0 70 31 / 41 61 - 21
E-Mail: solutions@levigo.de
Web: www.levigo.de

Inhaltsverzeichnis

1. Allgemeines

1.1. Über diese Dokumentation

Der hier vorliegende Leitfaden führt in die technischen Zusammenhänge des jadice server® ein.

Die Dokumentation beschränkt sich im Wesentlichen auf die Bereiche, die für Entwickler interessant sind (im Folgenden auch Integratoren genannt), um den jadice server® in eigene Applikationen zu integrieren.

Eine API-Referenz im javadoc-Format wird als separates Dokument zur Verfügung gestellt.

1.2. Feedback

Sollten Ihnen bei der Verwendung dieser Dokumentation Fehler auffallen oder möchten Sie Verbesserungsvorschläge einbringen, senden Sie bitte eine möglichst detaillierte Nachricht an **solutions@levigo.de**.

Ihr Feedback hilft bei der Weiterentwicklung dieser Dokumentation. Vielen Dank.

1.3. Über die jadice Produktfamilie

Die **jadice document platform** ist eine auf JAVA entwickelte Technologie mit zentraler Dokumentverarbeitungskomponente.

Dank ihrer Flexibilität als integrationsfreundliche Toolbox ist sie unterschiedlich einsetzbar und bietet die Basis für individuelle Archivclient- sowie Server-Lösungen im professionellen Dokumentenmanagement.

Der Dokumentbetrachter **jadice viewer**, das einstige Ausgangsprodukt der jadice Familie, ist ein wesentlicher Bestandteil der jadice document platform geblieben – allerdings mit weiterentwickeltem und erweitertem Funktionsumfang. Kann der **jadice viewer** ein angefordertes Dokument nicht wiedergeben, wird dieses an den **jadice server** weitergeleitet, welcher die passende Anwendung zur Konvertierung in das gewünschte Format ausführt.

Die jadice Produktfamilie bietet für den Dokumentenmanagement-Bereich integrationsfreundliche Java-Komponenten zur Archivierung, Verwaltung, Verteilung, Bearbeitung und Anzeige von Dokumenten an.

2. Der jadice server

Der jadice server ist eine Komponente zur beliebigen serverseitigen Verarbeitung von Datenströmen. Diese für die Weiterverarbeitung von Dokumenten konzipierte Gesamtlösung wird mit einfachen Workflowanweisungen und kaskadierten Kommandos gesteuert. Dabei ist jederzeit eine Änderung des Aufbereitungswegs möglich, denn sowohl die Herkunft des eingehenden Datenstroms als auch die Ausgabe des Verarbeitungsergebnisses sind frei wählbar.

Der jadice server kann dynamisch auf Daten reagieren und verfügt über mächtige Formaterkennungsmodule. Seine offenen Schnittstellen erlauben die Anbindung weiterer Funktionen, beliebiger Drittsoftware und ermöglichen somit die Erweiterung um neue Formate.

2.1. Produktidee

Kernaufgabe des jadice server ist es nicht, selbst ein Dokument vom Typ X nach Typ Y zu konvertieren. Stattdessen greift er unter anderem auf die Funktionalität der zugrunde liegenden **jadice document platform** zurück, die viele Möglichkeiten liefert, die in Langzeitarchiven anzutreffenden Formate zu konvertieren.

Außerdem stellt er eine flexible **Schnittstelle zu Applikationen** bereit, die das Konvertieren übernehmen. Dieser Vorteil beruht unter anderem auf der Plattformunabhängigkeit des jadice servers. Das heißt, dass der jadice server auf verschiedenen Plattformen installierbar ist und so mit einer großen Menge von Applikationen kommunizieren kann.

Dazu bietet der jadice server Schnittstellen zu extern gesteuerter Fremdsoftware. Deren Steuerung kann entweder direkt in der Java-Schnittstelle realisiert werden oder man kann die COM-Schnittstelle nutzen oder eine Batch-Datei schreiben.

Die Konvertierung kann je nach Format des Originals entweder durch eine **Funktion** einer Software (wie z. B. Photoshop), die das Ursprungs-Format anzeigen kann, oder durch die Benutzung eines vorhandenen **Image-Druckertreibers** erfolgen. Wichtig ist in beiden Fällen nur das Definieren von **geeigneten Schnittstellen** des jadice server mit der jeweiligen Software.

Durch die zweistufige Architektur des jadice server kann eine nahezu beliebige Skalierbarkeit erreicht werden. Er kann um beliebig viele Plugins zu verschiedenen Applikationen erweitert werden. Es können also viele Clients von den Programmen profitieren, die auf wenigen Rechnern (auf denen sich ein **jadice server** befindet) installiert wurden.

2.2. Einsatzmöglichkeiten des jadice servers

Die in diesem Kapitel beschriebenen Einsatzmöglichkeiten beschreiben die typischen Anwendungsfälle, die durch den jadice server abgedeckt werden. Für die konkrete technische Realisierung wird dabei jeweils auf die passenden Code-Beispiele im Kapitel „Anwendungsszenarien samt Code-Beispielen“ ab Seite 17 verwiesen.

Natürlich können an dieser Stelle nicht alle Einsatzmöglichkeiten, die der jadice server bietet, beschrieben werden. Außerdem können die jeweiligen Teilaufgaben miteinander kombiniert werden, sodass sie die konkrete Problemstellung passgenau abdecken.

2.2.1. Vereinheitlichung und Langzeitarchivierung

Der jadice server eignet sich für die Arbeit mit Daten, die sich nicht direkt clientseitig anzeigen lassen, wie etwa Office-Formate oder spezielle technische Formate.

Da der Server für die Konvertierung der Dokumente zuständig ist, werden die einzelnen Clients am Arbeitsplatz entlastet und somit auch das Arbeiten an älteren, weniger leistungsstarken Rechnern ermöglicht. Zusätzlich müssen die für die Konvertierung notwendigen Programme nicht auf allen Clients, sondern nur auf dem jadice server installiert sein.

Da diese Daten nun in einem Standardformat vorliegen (z. B. als PDF/A oder TIFF), muss das Programm, mit dem diese Dateien ursprünglich erstellt worden sind, nicht mehr vorgehalten werden. Somit eignen sich diese Dokumente auch für die Langzeitarchivierung.

Passende Code-Beispiele:

- Konvertierung unbekannter Eingabedaten in ein einheitliches Format (PDF) (Seite 23)
- Konvertierung von OpenOffice-Dokumenten nach PDF (Seite 23)
- Zusammenfassen mehrerer PDF-Dokumente (Seite 21)
- Konvertierung nach TIFF (Seite 22)

2.2.2. Kachelung

Ein weiterer Einsatzbereich ist das Ver- und Bearbeiten von Dokumenten mit sehr vielen oder auch besonders großen Seiten.

Bei der Arbeit mit sehr großen Dokumenten im jadice viewer oder im jadice web toolkit bietet jadice server die Möglichkeit, Dokumente zu kacheln, also in einzelnen Ausschnitten zu laden und darzustellen. Dies ist von Vorteil, wenn große Seiten, wie beispielsweise Baupläne, oder auch nur Teile eines hunderte Seiten umfassenden Dokuments dargestellt werden sollen.

2.2.3. Virtuelle Dokumente

Im Gegensatz dazu können durch den jadice server auch mehrere Dokumente zu einem großen, logischen Dokument zusammengefügt werden. So können etwa Personalakten mit den Zeiterfassungsdokumenten, Krankmeldungen und Tankabrechnungen eines Mitarbeiters in einer einheitlichen Datei erfasst werden.

Andererseits ist es auch möglich große Dokumente in einzelne Teildokumente zu trennen, um diese beispielsweise einzelnen Fachabteilungen zukommen zu lassen.

Passende Code-Beispiele:

- Zusammenfassen mehrerer PDF-Dokumente (Seite 21)

2.2.4. Dauerhafte Verankerung von Annotationen

Sind auf Dokumenten Annotationen aufgebracht, kann es notwendig sein, dass sie dauerhaft im Dokument verankert werden müssen, wenn die entsprechenden Dokumente an externe Stellen weitergeleitet werden sollen.

Dies kann aus verschiedenen Gründen notwendig sein. Einerseits ist nicht gewährleistet, dass externe Stellen das Datenformat für Annotationen verarbeiten können, sodass hier auf ein Standardformat (beispielsweise PDF) zurückgegriffen werden muss. Auch ist es denkbar, dass einige Teile des Dokuments mit einer Mask-Annotation „geschwärzt" werden müssen, da diese Geschäftsgeheimnisse oder datenschutzrelevante Daten enthalten. Hierbei muss sichergestellt werden, dass die externe Stelle keine Möglichkeit hat, die „geschwärzten" Informationen wieder lesbar zu machen.

Passende Code-Beispiele:

- Konvertierung nach TIFF (Seite 22)

2.2.5. Extraktion von Metadaten

Für eine schnelle Übersicht in einem Client ist es nicht immer notwendig, dass dieser das komplette Dokument übertragen bekommt. Stattdessen reicht es aus, wenn dieser Metadaten, die das Dokument charakterisieren, erhält.

Diese können z. B. verwendet werden, um clientseitig zu entscheiden, welche weiteren Verarbeitungsschritte ausgelöst werden sollen.

Passende Code-Beispiele:

- Extraktion von Dokument-Informationen (Seite 20)

2.2.6. Vereinheitlichung von E-Mails

Zur rechtssicheren Archivierung von E-Mails ist es nicht nur notwendig, den reinen Text der E-Mail, sondern auch deren Anhänge zu archivieren. Um sicherzustellen, dass die unterschiedlichen Dateiformate auch in Zukunft gelesen werden können, müssen diese jedoch in ein einheitliches Format gebracht werden. Dazu bietet sich der PDF-Standard an.

Außerdem ist es wünschenswert, dass automatisch eine Übersicht der Anhänge erzeugt wird. Beides leistet der jadice server.

Passendes Code-Beispiel:

- Konvertierung von E-Mails nach PDF (Seite 24)

2.2.7. Zentraler Dokumentendruck

Druckjobs können an Sachbearbeiterplätzen erstellt werden. Über die API stehen dafür eine Vielzahl von Einstellmöglichkeiten zur Verfügung. Diese Konfiguration wird an den jadice server übertragen, der als zentraler Druckserver dient. Im Rechenzentrum übernimmt er den Druckauftrag und reicht diesen verlustfrei an den Printcluster weiter.

Da die Aufbereitung durch den Druckertreiber mit – je nach Einstellung und gewünschter Druckqualitität – unter Umständen großen Druckdatenströmen an einer zentralen Stelle erfolgt, kann nicht nur die Netzwerklast verringert werden, sondern auch am Arbeitsplatz des Sachbearbeiters ist das sofortige Weiterarbeiten nach Start eines Druckauftrags möglich.

2.2.8. Verarbeitung gepackter Dateien

Für den Transport von großen Dateien über das Internet ist es sinnvoll, diese als Archivdatei zu packen und zu komprimieren, z. B. als ZIP-Datei. Vor der Verarbeitung muss diese Archivdatei jedoch entpackt werden. Um dies zu automatisieren, kann der jadice server zum Einsatz kommen.

Passendes Code-Beispiel:

- Entpacken von Archivdateien (Seite 22)

2.2.9. Datenvalidierung

Sämtliche Kundendokumente, die in in einem Langzeitarchiv abgelegt werden, müssen den Anforderungen genügen, auch in Jahrzehnten noch problemlos lesbar zu sein. Dazu muss ein geeignetes Dokumentenformat gewählt werden.

Zusätzlich muss aber bei der Archivierung sicher gestellt werden, dass die Dokumente vollständig und intakt sind sowie formal der technischen Spezifikation entsprechen.

Mit jadice server kann im Importprozess eine, von der Erzeugung der Dateien unabhängige Stufe eingeführt werden, die die Dateiformate erkennt und geeignete Validierungsmechanismen gegen definierte Standards aufruft.

Damit können fehlerhafte Dateien schon vorab erkannt und zur Überprüfung und Nachbearbeitung aussortiert werden. Dadurch wird sichergestellt, dass die Dokumente im Langzeitarchiv auch in Jahrzehnten noch zur Verarbeitung zur Verfügung stehen.

3. Systemarchitektur

Der jadice server kann geclustert betrieben werden. Somit lässt sich eine hohe Verfügbarkeit bei gleichzeitig hoher Leistungsfähigkeit realisieren. Pro Serverinstanz wird jadice server in einer JVM ausgeführt und verwaltet jeweils einen Pool von OpenOffice-Prozessen bzw. verschiedene COM-Server. Über ein Messagingsystem (MOM), das als Transportschicht fungiert, fordern Clients die Ausführung von Jobs an. Der Zugriff durch die Clients erfolgt über eine Client-Bibliothek (siehe Kapitel 4.2.).

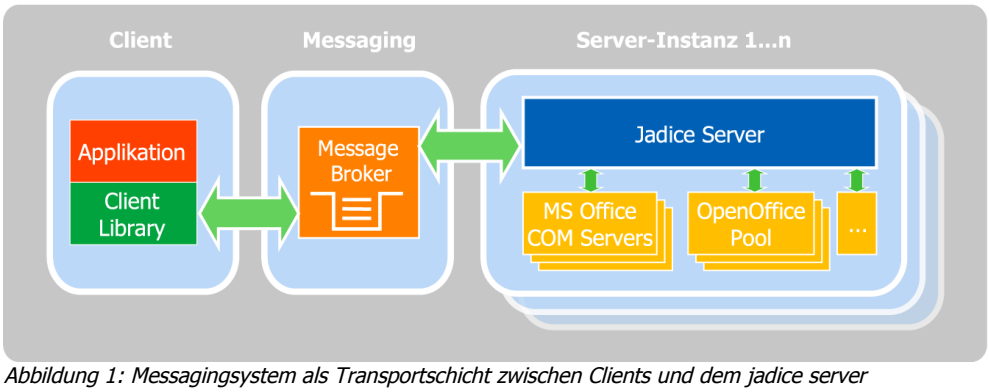


Abbildung 1: Messagingsystem als Transportschicht zwischen Clients und dem jadice server

Da das Messagingsystem über die standardisierte JMS-Schnittstelle (Java Message Service) angebunden wird, können bereits im Unternehmen verwendete Messagebroker in die Systemarchitektur eingebunden werden.

Durch diese Architektur lässt sich mit einem geringen Aufwand ein automatisches Load-Balancing bei gleichzeitig hoher Verfügbarkeit realisieren.

3.1. Arbeitsweise

Der jadice server ist so konzipiert, dass die Verarbeitung von Dokumenten und Dokumentdaten in Aufträge (Jobs[1]) und diese wiederum in einzelne Verarbeitungsschritte (Knoten, Nodes[2]) zerlegt wird, die dadurch einen Workflow beschreiben.

Clients steuern die Ausführung der Aufträge und verteilen sie über das Messaging-

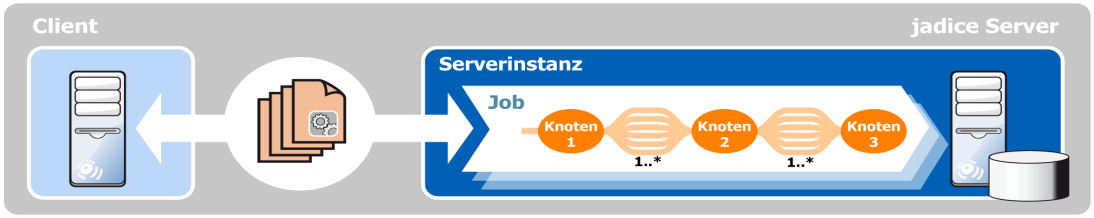


Abbildung 2: Beschreibung eines Jobs durch einzelne Knoten

system an den jadice server.

Knoten sind die einzelnen, individuell definierten Aufgabenschritte, aus denen ein Job besteht. Sie sind untereinander durch Bündel von Datenströmen verbunden, die Nutzdaten sowie Metadaten transportieren. Die Verarbeitungsschritte sind daher in Bezug auf Inhalt und Reihenfolge abhängig vom auszuführenden Job.

Die Knoten unterscheiden sich voneinander in ihrer Aufgabenstellung. So können z. B. in einem Knoten Dokumente mit OpenOffice, MS Office oder den Funktionalitäten der jadice document platform konvertiert werden, während in einem anderen Knoten Dokumente geteilt oder zusammengeführt werden. Weitere Knoten können Metadaten aufbereiten und Daten formatieren. Die Druckaufbereitung von Dokumenten, die Rasterisierung und Bereitstellung von Daten als Tile-Server und die Klassifizierung von Datenströmen sind weitere Beispiele für Knoten. Selbst das ein-

1 com.levigo.jadice.server.Job
2 com.levigo.jadice.server.Node

fache Aus- oder Einpacken von Daten aus oder in Archive geschieht in solch einem Verarbeitungsschritt.

Hierbei ist nicht fest vorgeschrieben, dass jeder Knoten genau einen Vorgänger bzw. einen Nachfolger hat. So gibt es bereits einen vordefinierten Knoten[3], der einen Datenstrom vervielfältigt und an mehrere nachfolgende Knoten weiter gibt sowie einen Knoten[4], der mehrere Vorgängerknoten hat und alle einkommenden Datenströme an nur einen Nachfolger weiterreicht. Einzige Bedingung bei der Konfiguration des Workflows ist nur, dass beim Zusammenstellen der Nodes keine Zyklen entstehen dürfen.

3 com.levigo.jadice.server.nodes.MultiplexerNode
4 com.levigo.jadice.server.nodes.DemultiplexerNode

4. Installation und Konfiguration

4.1. Server

Der jadice server ist ab der Java-Version 1.5 lauffähig. Für die Installation muss zuerst die Distributionsdatei (i. d. R. jadice-server-4.2.x.x-dist.zip) entpackt werden.
Die Installations-/Startdateien befinden sich im Verzeichnis **/bin**.

Um den Windows-Dienst zu installieren, stehen folgende Dateien zur Verfügung:

install.bat	Installiert einen Win32-Dienst.
start.bat	Startet den Dienst.
stop.bat	Stoppt den Dienst.
pause.bat	Dienst wird angehalten.
resume.bat	Dienst wird fortgesetzt.

Alternativ kann der Server mit dem Skript **jadice-server.bat** gestartet werden.
Für Linux basierte Betriebssysteme ist das Skript **jadice-server.sh** zu verwenden.
Beim Start wird eine Konsole geöffnet; mit der Tastenkombination CTRL-C kann der Server beendet werden.

Alle serverseitig benötigten jar-Dateien befinden sich im Verzeichnis **/server-lib**.

4.1.1. Lizenzdatei

Neben der Distributionsdatei von jadice server erhalten Sie eine separate Lizenzdatei (**JadiceServerLicense.properties**). Diese muss im Konfigurationsverzeichnis (**/server-config**) des jadice servers abgelegt werden.

Ist diese Datei nicht vorhanden bzw. die Lizenz abgelaufen oder ungültig, wird beim Start des jadice servers sowie bei jeder Anforderung eines Jobs eine Fehlermeldung in das Server- und das Client-Log geschrieben.
Sollte es sich um eine zeitlich beschränkte Evaluationslizenz handeln, wird nur beim Start des Servers eine Meldung gezeigt. Darüber hinaus gibt es keine Funktionseinschränkungen des jadice servers.

4.1.2. Manueller Download für Silbentrennung

jadice server verwendet Apache FOP zur Konvertierung von XML-Dokumenten und E-Mails nach PDF. Aus lizenzrechtlichen Gründen darf das optionale Paket zur Silbentrennung dem Auslieferungspaket von jadice server nicht beiliegen. Dies ist jedoch frei unter **http://offo.sourceforge.net/** verfügbar. Zur Installation muss lediglich die Datei **hyphenation.jar** in den Ordner **server-lib** kopiert werden.

4.1.3. Konfiguration für JVM 1.6 pre-Update 10

In der SUN Java-VM der Version 6 (1.6.0_*x*), aber vor Update 10, sind Bibliotheken für jaxb[5] vorhanden, die nicht kompatibel mit jadice server sind. Um die von jadice server mitgelieferten Bibliotheken zu aktivieren, sind folgende Einstellungen nötig:

- Entfernen Sie das Kommentarzeichen (#) in der Datei **wrapper/wrapper.conf** in folgender Zeile:

```
#wrapper.java.additional.1=-Djava.endorsed.dirs=../endorsed-lib
```

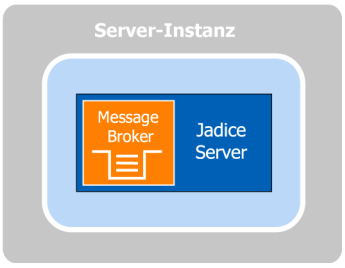
- Kommentieren Sie in der Datei **server-config/jadice-server.options** folgende beiden Zeilen aus (Kommentarzeichen #):

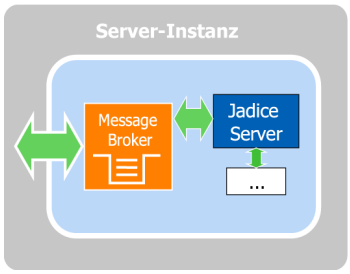
5 Java Architecture for XML Binding

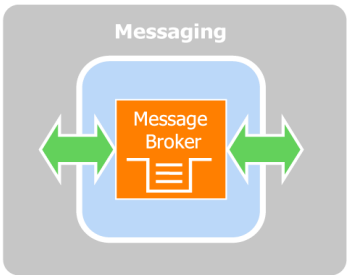
```
--library-dir
../endorsed-lib
```

- Soll jadice server im Multi-VM-Modus laufen (vgl. Kapitel 4.1.12), muss in der Datei **server-config/application/multi-vm-manager.xml** im Abschnitt <property name="instanceJVMOptions"> folgender Eintrag eingefügt werden:

```
<value>
   -Djava.endorsed.dirs=/$pfad/$zu/$JadiceServer/endorsed-lib
</value>
```







4.1.4. Konfiguration des Messagingsystems

Es stehen drei Varianten für den Aufbau des Messagingsystems zur Verfügung.

- Eingebetteter Broker (Standard)
  - Betrieb eines Brokers innerhalb der VM von jadice server
  - Einsatz von Apache ActiveMQ
  - Clusterfähig durch „Network of Brokers"
  - Konfiguration siehe Kapitel 4.1.5.
- Separater Broker
  - Einsatz eines separaten Brokers auf der Infrastruktur des jadice server
  - Einsatz beliebiger kompatibler MOM
  - Ggf. clusterfähig
- Separate MOM-Infrastruktur
  - Einsatz separater MOM-Infrastruktur
  - Einsatz beliebiger kompatibler MOM
  - Verfügbarkeit muss durch Infrastruktur sichergestellt werden

Welche dieser drei Varianten zum Einsatz kommt, wird in der Datei **server-config/application/server.xml** konfiguriert. Im Abschnitt <bean id="jms-connection-factory" …> wird der zu verwendende Konnektor und im Abschnitt <property name="properties" …> unter requestQueueName der Name der Message-Queue und der Port für ActiveMQ konfiguriert, zu der sich der jadice server verbinden soll.

4.1.5. Konfiguration des eingebetteten Messagebrokers

jadice server stellt standardmäßig das zu verwendende Messagingsystem (d. h. den Message-Broker) in der Server-Instanz bereit. In diesem Falle wird Apache ActiveMQ[6] als Messagingsystem verwendet. Es wird in der Datei **server-config/application/activemq-broker.xml** konfiguriert. Um die Übertragung zwischen Client und Server mit SSL zu verschlüsseln, findet sich unter http://activemq.apache.org/ssl-transport-reference.html und http://activemq.apache.org/how-do-i-use-ssl.html eine Anleitung, wie dies realisiert werden kann.

Der Port und der Name der verwendeten Messagequeue werden zentral in der Datei **server-config/application/server.xml** konfiguriert, siehe Abschnitt 4.1.4..

Um diesen internen Broker nicht zu starten, muss der Abschnitt <bean id="broker" …> in der Datei **server-config/application/server.xml** auskommentiert werden.

4.1.5.1. Clustering

Folgende Konfigurationsänderung ist notwendig, um jadice server via ActiveMQ in einem Cluster zu betreiben.

- In der Datei **activemq-broker.xml** die Kommentarzeichen um eins der beiden Elemente <networkConnector …> entfernen.

Der Cluster hat dabei folgenden Aufbau:

6 Siehe http://activemq.apache.org/

- Auf jedem Knoten läuft eine Instanz von jadice server mit jeweils einem eingebetteten Broker unter Apache ActiveMQ.
- Das Clustering basiert auf einem ActiveMQ Network-of-Brokers[7], d. h. der eingebettete Broker eines jeden Knotens nimmt gleichberechtigt an einem verteilten Broker teil.
- Die Broker finden sich entweder gegenseitig über Auto-Discovery[8], das über Multicast realisiert wird, oder das Network-of-Brokers wird statisch definiert.

Damit das Clustering wirksam ist, müssen die Clients entsprechend konfiguriert werden. Finden sich die Broker über Auto-Discovery, so kann man für Clientverbindungen die Methode Multicast-Discovery[9] verwenden. Dazu muss der Gruppenname des Clusters bekannt sein, den die jadice server-Instanzen bilden. Dieser wird in der Datei **/server-config/application/server.xml** unter jadice.server.activemq-group gesetzt und lautet standardmäßig jadice-server.cluster.

Die URL, mit der sich ein Client in diesem Fall mit dem Cluster verbinden kann, lautet (vgl. Codebeispiel in Kapitel 5.3.1.):

```
discovery:(multicast://default)?
  group=jadice-server.cluster&initialReconnectDelay=100
```

Wurde ein statisches Network-of-Brokers definiert, muss die Verbindungs-URL ebenfalls statisch definiert werden:

```
failover:(<URL_Server_A>,<URL_Server_B>)
// Beispiel:
failover:(tcp://serverA:61616,tcp://serverB:61616)
```

4.1.6. Konfiguration Wrapper

Der Server wird über einen betriebssystemabhängigen Wrapper gestartet. Hier werden Java VM und Klassenpfad-Parameter definiert.

Die Wrapper-Konfigurationsdatei **wrapper.conf** befindet sich im Verzeichnis **/wrapper**.

Pfad zur Java VM:

```
# Java Application
wrapper.java.command=java
```

Definition weiterer Klassenpfade:

```
# Java Classpath (include wrapper.jar)  Add class path
# elements as needed starting from 1
wrapper.java.classpath.1=../wrapper/lib/wrappertest.jar
wrapper.java.classpath.2=../wrapper/lib/wrapper.jar
wrapper.java.classpath.3=../bin/server-console.jar
wrapper.java.classpath.4=../msoffice-lib
wrapper.java.classpath.5=<Neuer Klassenpfad>
```

Definition zusätzlicher Parameter der Java VM, z. B. Setzen des temporären Verzeichnisses für die VM:[10]

```
# Java Additional Parameters
wrapper.java.additional.1=-server
wrapper.java.additional.2=-Djava.io.tmpdir=C:\tmp
```

7 Siehe http://activemq.apache.org/networks-of-brokers.html
8 Siehe http://activemq.apache.org/discovery.html
9 Siehe http://activemq.apache.org/discovery-transport-reference.html
10 Wird der jadice server im Multi-VM-Modus gestartet, werden diese Parameter nur für die zentrale Instanz wirksam (vgl. Kapitel 4.1.12.)

Definition Java VM Speichereinstellungen, wrapper.java.initmemory entspricht Paramater -Xms, wrapper.java.maxmemory entspricht Parameter -Xmx:[11]

```
# Initial Java Heap Size (in MB)
#wrapper.java.initmemory=3

# Maximum Java Heap Size (in MB)
wrapper.java.maxmemory=512
```

Außerdem ist es möglich, den Klassenpfad und somit die geladenen Java-Bibliotheken zu ändern, um beispielsweise jadice server um eigene Worker-Implementierungen zu erweitern. Dies geschieht in den Dateien **/server-config/jadice-server.-options** und **/server-config/jadice-server-local.options**. Folgende Einträge sind hier möglich:

`-cp <Jar-Bibliothek>`	Fügt eine einzelne Jar-Bibliothek zum Klassenpfad von jadice server hinzu.
`--classpath <Jar-Bibliothek>`	
`-ld <Verzeichnis>`	Alle Jar-Bibliotheken, die im angegebenen Verzeichnis vorhanden sind, werden in alphabetischer Reihenfolge zum Klassenpfad von jadice server hinzugefügt.
`--library-dir <Verzeichnis>`	

Dabei ist zu beachten, dass zwischen der Option und dem Parameter ein Zeilenumbruch erfolgen muss. Der effektive Klassenpfad wird in folgender Weise gebildet:

1. Einträge unter -cp / --classpath aus jadice-server.options
2. Einträge unter -cp / --classpath aus jadice-server-local.options
3. Einträge unter -ld / --library-dir aus jadice-server.options
4. Einträge unter -ld / --library-dir aus jadice-server-local.options

Des Weiteren sind folgende Optionen möglich:

`-xo <Konfigurationsdatei>`	Bindet eine weitere Konfigurationsdatei ein. Diese muss hier gezeigter Syntax entsprechen.
`--extra-options <Konfigurationsdatei>`	
`-dCL`	Beim Start wird eine Auflistung des effektiven Klassenpfads und der Classloader-Hierarchie angezeigt.
`--debug-classpath`	
`-dX`	Eventuelle Exceptions und Stacktraces, die während des Startens geworfen werden, werden ausgegeben.
`--debug-using-exceptions`	

4.1.7. Konfiguration OpenOffice

Es muss auf das Verzeichnis **<OpenOffice-Verzeichnis>/program** verwiesen werden, damit der Server auf die Programmdatei von OpenOffice zugreifen kann. Die Konfiguration steht in der Datei **/server-config/jadice-server-local.options**, z. B.

```
# OpenOffice Verzeichnis
-cp
C:\Programme\OpenOffice.org 3\program\
```

Für die Verwendung von OpenOffice 3.0.x ist es außerdem notwendig, **<OpenOffice-Verzeichnis>/URE/java/jurt.jar** im Klassenpfad einzubinden.

11 Siehe Fußnote 10

Sind die Pfade nicht richtig konfiguriert, wird bei einer Konvertierung ein Fehler ausgegeben.

Auf Nicht-Windows-Betriebssystemens (Linux, Unix u. ä.) muss außerdem das Paket **Xvfb** (X window virtual framebuffer) installiert sein, damit ein fensterloser und somit automatisierter Betrieb von OpenOffice erfolgen kann.

4.1.8. Konfiguration MS Office

Wird jadice server unter Windows NT, 2000, Server 2003 oder einer früheren Version installiert und soll die Konvertierung mit den MSWord- / MSExcel- /... Nodes erfolgen, so muss zunächst das „Microsoft Visual C++ 2005 SP1 Redistributable Package (x86)"[12] installiert werden.

Unter MS Office 2007 Service Pack 2 ist es möglich, PDFs nativ zu exportieren. In Versionen von MS Office 2007 vor Service Pack 2 muss das „2007 Microsoft Office Add-in: Microsoft Save as PDF"[13] installiert werden, um den nativen PDF-Export nutzen zu können.

Desweiteren muss im Trust Center (deutsch: „Vertrauensstellungscenter") von MS Office eingestellt werden, wie mit Makros umgegangen werden soll. Da im Serverbetrieb keine Benutzerabfrage möglich ist, sind nur die Optionen „Alle Makros ohne Benachrichtigung deaktivieren" und „Alle Makros aktivieren" sinnvoll (siehe Abbildung 3). Ebenso ist mit den ActiveX-Einstellungen zu verfahren.

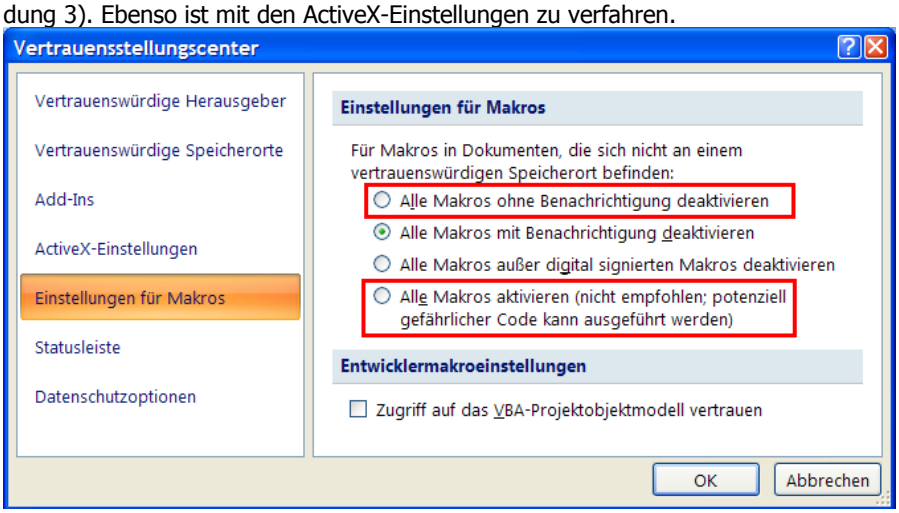


Abbildung 3: Zulässige Einstellungen für Makros in MS Office

4.1.9. Konfiguration MS Outlook

Um die Sicherheitsbestimmung zu umgehen, die es verbieten, dass jadice server auf MS Outlook zugreift, muss das Programm „Advanced Security for Outlook"[14] installiert und konfiguriert werden.

Konfiguration:

1. Loggen Sie sich mit unter dem Benutzer, unter dem jadice server läuft, auf dem Server-Rechner ein.
2. Starten Sie eine Konvertierung, die den MSOutlookNode[15] auslöst.
3. Bestätigen Sie den Dialog von „Advanced Security for Outlook" mit „Allow Access" und „Always perform (...)"

12 Siehe http://www.microsoft.com/downloads/details.aspx?FamilyID=200b2fd9-ae1a-4a14-984d-389c36f85647&displaylang=en
13 Siehe http://www.microsoft.com/downloads/details.aspx?FamilyID=f1fc413c-6d89-4f15-991b-63b07ba5f2e5&displaylang=en
14 Siehe http://www.mapilab.com/de/outlook/security/
15 com.levigo.jadice.server.msoffice.MSOutlookNode

Außerdem müssen unter MS Outlook 2007 die Option „Bei Programmbeendigung Ordner 'Gelöschte Objekte' leeren“[16] aktiviert und die Option „Warnung anzeigen, bevor Elemente endgültig gelöscht werden“[17] deaktiviert sein.

4.1.10. Konfiguration logging

Zur Verwaltung von Log-Einträgen wird auf das Framework log4j[18] zurückgegriffen. Dieses wird in der Datei **/server-config/logging/log4j-appenders.xml** bzw. **log4j-appenders-mvm.xml** im Multi-VM-Modus (siehe Kapitel 4.1.12) konfiguriert. Standardmäßig werden Log-Meldungen auf die Konsole ausgeschrieben sowie in der Datei **/log/jadice-server.log** gespeichert. Weitere Möglichkeiten sind in der Konfigurationsdatei als Kommentare aufgeführt und müssen lediglich aktiviert werden.

4.1.11. Konfiguration Ghostscript

Um den GhostscriptNode[19] verwenden zu können, müssen zunächst folgende Voraussetzungen geschaffen werden:

- Installation von GPL Ghostscript[20], Version 8.64 oder neuer
- Anpassung der Datei **/server-config/ghostscript/ghostscript.xml**. Unter **<bean id="ghostscript" (...)>** muss ein Element eingefügt werden, das folgenden Inhalt hat:
  <property name="executableName" value="*<Ort der Ghostscript-Anwendungsdatei>*" />
  Für die Orte, unter denen Ghostscript unter Windows bzw. Linux standardmäßig installiert wird, sind bereits Vorlagen vorhanden. Hierzu muss nur der passende XML-Kommentar entfernt werden.

4.1.12. Konfiguration Multi-VM-Modus

Durch die Einbindung von externen Bibliotheken ist es möglich, dass diese in einem Fehlerfall die Java Virtual Machine und somit auch den kompletten jadice server zum Absturz bringen.

Um in diesem Fall mit der Bearbeitung von weiteren Jobs fortzufahren, gibt es die Möglichkeit, den jadice server auf einem Rechner in mehreren Instanzen zu starten. Hierbei übernimmt eine zentrale Instanz des jadice servers die Überwachung aller anderen, die die eigentliche Arbeit durchführen. Sollte eine dieser Arbeiter-Instanzen nicht mehr reagieren oder abgestürzt sein, beendet die zentrale Instanz den zugehörigen Prozess und startet automatisch eine neue Instanz des jadice servers.

Um den jadice server in diesem Modus zu starten, muss in der Datei **/server-config/application/server.xml** der auskommentierte Teil wie folgt geändert werden:

```
<!--  <import resource="single-instance.xml"/> -->
      <import resource="multi-vm-manager.xml"/>
```

In der Datei **/server-config/application/multi-vm-manager.xml** können die Arbeiterinstanzen im Abschnitt <bean id="server" …> konfiguriert werden:

Für die Anzahl der zu startenden Arbeiter-Instanzen gibt es folgende Möglichkeiten:

- Feste Anzahl an Arbeiterinstanzen:
  `<property name="fixedVMCount" value="<n>" />`
- *n* Instanzen je Prozessorkern:
  `<property name="perProcessorVMCount" value="<n>" />`

16 Zu finden unter Extras → Optionen → Weitere
17 Zu finden unter Extras → Optionen → Weitere → Erweiterte Optionen
18 Siehe http://logging.apache.org/log4j/
19 com.levigo.jadice.server.ghostscript.GhostscriptNode
20 Siehe http://www.ghostscript.com/

Des Weiteren können unter <property name="instanceJVMOptions"…> der zu startenden Java VM Startparameter wie z. B. die verfügbare Speichergröße mitgegeben werden.

4.1.13. Konfiguration Webservice-Schnittstelle

Um die Schnittstelle, über die Anfragen im XML-Format an den jadice server gestellt werden können, zu aktivieren, muss im der Datei **server-config/application/server.xml** die Kommentarmarkierung um den Abschnitt **<import resource="webservices.xml"/>** entfernt werden.

In dieser Datei werden im Abschnitt <bean id="web-service-provider" …> Endpunkte[21] des Webservices propagiert. Der vorgebene Eintrag

```
<entry key=
  "http://${jadice.server.hostname}:9000/jadiceServer">
```

stellt einen Endpunkt unter dem Namen des Rechners auf Port 9000 bereit.

Zur Verwendung der Webservice-Schnittstelle siehe Kapitel 6.

4.2. Client

Für den clientseitigen Einsatz müssen die jar-Dateien aus dem Verzeichnis

/client-lib (für Java 1.5 und aufwärts) bzw.

/client-lib-jkd14 (für Java 1.4)

in den Klassenpfad der Anwendungs-/ Entwicklungsumgebung eingebunden werden.

4.3. Installation in der Entwicklungsumgebung Eclipse

4.3.1. Server

- Ein neues Java-Projekt anlegen.
- Folgende jar-Dateien zum Klassenpfad hinzufügen:

  Alle jar-Dateien aus dem Verzeichnis /server-lib

  Datei activemq-all-5.x.x.jar aus dem Verzeichnis /apache-activemq-5.x.x

  Dateien spring-###-2.5.5.jar aus dem Verzeichnis /apache-activemq-5.x.x/lib/optional
- Alle Konfigurationsdateien aus dem Verzeichnis /server-conf in das Source-Verzeichnis (src) kopieren.
- Zum Schluss die Klasse JadiceServerControl aus der jar-Datei server-core starten.
- Der Server ist nun betriebsbereit.

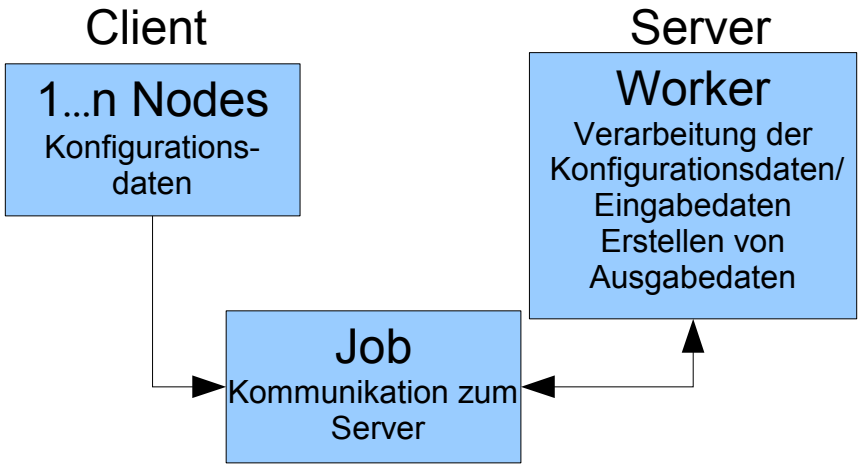
4.3.2. Client

- Ein neues Java-Projekt anlegen.
- Alle jar-Dateien aus dem Verzeichnis /client-lib (für Java 1.4: /client-lib-jdk14) zum Klassenpfad hinzufügen.

21 Siehe javax.xml.ws.Endpoint

5. Anwendung / Funktionalität

Worker sind serverseitige Implementierungen, die bestimmte Aufgaben erledigen, die in der Regel rechenintensiv und ressourcenaufwändig sind. Dazu gehört z. B. die Generierung von großen Dokumenten, das Erstellen von Datenströmen für die Anzeige / Drucken / Seitenvorschau / -auswahl usw. Die Worker werden von einem



Client durch korrespondierende Nodes angesprochen und mit Daten bedient.

Die Superklasse für eine Node-Implementation ist die Klasse **com.levigo.jadice.server.Node**, für eine Worker-Implementation **com.levigo.jadice.server.core.worker.NodeWorker**.

Eine ausführliche Beschreibung der in jadice server vordefinierten Nodes kann der beigelegten Javadoc-Dokumentation entnommen werden. Wie eigene Nodes und Worker implementiert werden können, um die Funktionalität des jadice servers zu erweitern, wird im Kapitel 5.4. anhand eines Beispiel beschrieben.

5.1. Aufgabenstellung clientseitig

Clientseitig wird ein Job erstellt, welchem dann eine oder mehrere Node-Implementationen mit den benötigten Daten (Konfiguration / Datenströme) gesetzt werden. Dieser Job nimmt dann Kontakt zum Server auf und setzt dort die Nodes in eine Warteschlange. Durch die asynchrone Kommunikationsschnittstelle wird der Client während der Abarbeitung durch den Server nicht blockiert.

Die Nodes bilden einen gerichteten, azyklischen Graphen und definieren dadurch den Workflow (z. B. lädt Node 1 Daten und Node 2 bearbeitet diese dann, Node 3 sendet die Daten schließlich an den Client zurück). Wie dies zu implementieren ist, wird in Kapitel 5.3. anhand einiger typischer Beispiele verdeutlicht.

5.2. Aufgabenstellung serverseitig

Der jadice server erstellt aus den vom Job übermittelten Nodes einen Workflow. Dabei werden die miteinander verketteten Nodes nacheinander abgearbeitet, indem die korrespondierenden Worker gestartet werden. Die erzeugten Daten werden über StreamBundle[22]-Objekte an den jeweils nächsten Worker weitergereicht.

5.3. Anwendungsszenarien samt Code-Beispielen

Die Anwendungsszenarien und Konfigurationsmöglichkeiten für den jadice server sind ebenso zahlreich wie vielfältig.

Deshalb sollen hier nur die gängigsten Szenarien aufgegriffen werden, die als Ausgangspunkt für eigene Implementierungen dienen können.

22 com.levigo.jadice.server.shared.types.StreamBundle

In den meisten Fällen sind diese nach dem Schema

- Datei vom Client empfangen
- Serverseitige Verarbeitung
- Ergebnis zum Server zurücksenden

aufgebaut. Dies ist der Einfachheit der Beispiele geschuldet. In realen Anwendungen werden die Ergebnisse üblicherweise nicht direkt nach der Verarbeitung zurückgesendet, sondern in kaskadierten Schritten verarbeitet. Auch die Quelle und das Ziel der Datenströme sind nicht zwingenderweise der Client, sondern können beispielsweise ein zentraler Datei-, Mail-, Archivserver o. ä. sein.

5.3.1. Erstellen eines Server-Jobs

Der Server-Job wird clientseitig erstellt, dem Job können 1…n Nodes angehängt werden.

Konfigurationsmöglichkeiten des Server-Jobs:

– Timeout und ähnliche Limits[23]
– JobListener-Implementation
– Workflow (= azyklischer Graph, der durch verkettete Nodes definiert wird)

Job erstellen (im Beispiel mit ActiveMQ als Messagebroker):

```
public Job createServerJob() {
  // Job-Factory erstellen mit den Parametern „Server-
  // Url“ und „Queue-Name“ (vgl. 4.1.5.)
  JMSJobFactory jobFactory = new JMSJobFactory(
    new ActiveMQConnectionFactory("tcp://<Broker_IP>
      :<Broker-Port>"), "<Queue-Name>");
  // Server-Job erstellen
  Job job = jobFactory.createJob();
  return job;
}
```

Code Beispiel

Job konfigurieren und ausführen:

```
// Job erstellen
Job job = createServerJob();
// Timeout setzen (60 Sekunden)
job.apply(new TimeLimit(60, TimeUnit.SECONDS));
// JobListener registrieren (vgl. 5.3.2.)
job.addJobListener(<JobListener-Implementation>);
// Workflow definieren (siehe unten)
job.attach(<Node-Workflow>);
// Job an Server senden
job.submit();
```

Code Beispiel

5.3.2. Erstellung eines JobListeners

Mit dem JobListener[24] können Zustände des Server-Jobs und serverseitige Fehlermeldungen verarbeitet werden.

Beispiel einer JobListener-Implementation MyJobListener:

```
public class MyJobListener implements JobListener {
 public void stateChanged(Job job, State old,
   State new) {
   dump("stateChanged", job, old, new, null, null);
```

23 com.levigo.jadice.server.Limit
24 com.levigo.jadice.server.JobListener

```
 }

 public void executionFailed(Job job, Node node,
   String messageId, String reason, Throwable cause) {
   dump("executionFailed", job, node, messageId,
     reason, cause);
 }

 public void errorOccurred(Job job, Node node,
  String messageId, String message, Throwable cause) {
  dump("errorOccurred", job, node, messageId, message,
    cause);
 }

 public void warningOccurred(Job job, Node node,
   String messageId, String message, Throwable cause) {
   dump("warningOccurred", job, node, messageId,
     message, cause);
 }

 private void dump(String ctx, Job job, Object
   arg1, Object arg2, Object arg3, Object arg4) {
   System.err.println("Context:    " + ctx);
   System.err.println("Job:        " + job.toString());
   System.err.println("            " + arg1);
   System.err.println("            " + arg2);
   System.err.println("            " + arg3);
   System.err.println("            " + arg4);
 }
}
```

Code Beispiel

5.3.3. Identifikation unbekannter Eingabedaten

Der jadice server bietet mächtige Module zur Erkennung unbekannter Dateiformate. Diese werden in den Modulen zur automatischen Konvertierung unbekannter Dateien bzw. E-Mails eingesetzt (siehe Kapitel 5.3.8. und 5.3.10.).

Darüber hinaus ist es auch möglich, diese Module durch den StreamAnalysisNode[25] anzusprechen und für eigene Zwecke zu verwenden.

```
// Server-Job erstellen (siehe 5.3.1.)
Job job = createServerJob();
// Nodes erstellen:
// 1. Dateneingabe-Node
StreamInputNode siNode = new StreamInputNode();
// 2. Analyseknoten
StreamAnalysisNode saNode = new StreamAnalysisNode();
// 3. Ausgabe-Node
StreamOutputNode soNode = new StreamOutputNode();
// Workflow erstellen
job.attach(siNode.
  appendSuccessor(saNode).
  appendSuccessor(soNode));
// Job ausführen
job.submit();
// Dokument-Datenstrom senden
siNode.addStream(<InputStream>);
// Dateneingabe beenden
siNode.complete();
```

25 com.levigo.jadice.server.jadice4x.StreamAnalysisNode

```
// Auf Antwort vom Server warten[26]
for (Stream stream : soNode.getStreamBundle()) {
      // Lesen der beschreibenden Daten
      StreamDescriptor descr = stream.getDescriptor();
      String mimeType = descr.getMimeType();
}
```

5.3.4. Extraktion von Dokument-Informationen

Die JadiceDocumentInfoNode[27]-Implementation schickt ein Dokument an den Server. Dieser lädt das Dokument und liefert dokumentspezifische Informationen[28] an den Client zurück.

Die DocumentInfoListener-Implementation:

```
public class DocumentInfoListener implements
 IDocumentInfoResultListener {
 // Dokumentinfo, wird vom serverseitigen Worker
 // erstellt.
 private IDocumentInfo documentInfo = null;
 private boolean documentInfoReceived = false;

 public void documentInfoRecieved(IDocumentInfo info) {
  // Wenn der Worker fertig ist, wird die DokumentInfo
  // hier übergeben
  documentInfo = info;
  documentInfoReceived = true;
 }

 public void waitForDocumentInfo() {
  // Blockieren, bis der Worker fertig ist
  while (!documentInfoReceived) {
   try {
    Thread.sleep(250);
   } catch (Exception e) {
   }
  }
 }

 public IDocumentInfo getDocumentInfo() {
  return documentInfo;
 }
}
```

Code Beispiel

Erstellen und Ausführen eines Jobs über die JadiceDocumentInfoNode-Implementation:

```
// Server-Job erstellen (siehe 5.3.1.)
Job job = createServerJob();
// Listener erstellen
DocumentInfoListener documentInfoListener =
  new DocumentInfoListener();
// InfoNode erstellen, Listener hinzufügen
JadiceDocumentInfoNode infoNode =
```

26 Die Methode getStreamBundle() blockiert, bis der Server die Abarbeitung beendet hat. Eine asynchrone Verarbeitung ist durch die Verwendung einer JobListener-Implementierung zu realisieren (vgl. Kapitel 5.3.2.).
27 com.levigo.jadice.server.jadice4x.JadiceDocumentInfoNode
28 com.levigo.jadice.server.jadice4x.IDocumentInfo

```
  new JadiceDocumentInfoNode();
infoNode.addInfoResultListener(documentInfoListener);
// InputNode erstellen, ist der 1. Node im Workflow
StreamInputNode siNode = new StreamInputNode();
// Infonode als Nachfolger hinzufügen.
// Serverseitig wird zuerst der InputNode verarbeitet
// zum Laden des Dokuments, dann wird der InfoNode
// ausgeführt, der das geladene Dokument analysiert.
siNode.appendSuccessor(infoNode);
// 1. Node dem Job übergeben...
job.attach(siNode);
// ...und Job starten
job.submit();
// Erst nach dem Start des Jobs kann der Datenstrom dem
// Inputnode übergeben werden
siNode.addStream(new FileInputstream("<Dateiname>"));
// Dateneingabe beenden
siNode.complete();
// Auf Abarbeitung durch Server warten (siehe oben)
documentInfoListener.waitForDocumentInfo();
// DokumentInfo holen und Daten ausgeben
IDocumentInfo documentInfo =
  documentInfoListener.getDocumentInfo();
System.out.println("Format         : " +
  documentInfo.getFormat(0));
System.out.println("Anzahl Seiten  : " +
  documentInfo.getPageCount());
System.out.println("Grösse (Pixel) : " +
  documentInfo.getSize(0).width + "x" +
  documentInfo.getSize(0).height);
System.out.println("Auflösung (dpi): " +
  documentInfo.getResolution(0));
```

Code Beispiel

5.3.5. Zusammenfassen mehrerer PDF-Dokumente

Mit dem PDFMergeNode[29] ist es möglich, mehrere PDF-Dokumente zu einem einzigen zusammenzufassen.

```
Job job = createServerJob(); // (siehe 5.3.1.)
// Nodes erstellen:
// 1. Dateneingabe-Node
StreamInputNode siNode = new StreamInputNode();
// 2. Zusammenfassen von Eingabedaten (1…n zu 1)
PDFMergeNode pmNode = new PDFMergeNode();
// 3. Ausgabe-Node
StreamOutputNode soNode = new StreamOutputNode();
// Workflow erstellen
job.attach(siNode.
  appendSuccessor(pmNode)
 .appendSuccessor(soNode));
//Job ausführen
job.submit();
// PDF-Dokument-Datenstrom senden
siNode.addStream(<InputStream_PDF_1>);
siNode.addStream(<InputStream_PDF_2>);
(…) // evtl. weitere Datenströme
// Dateneingabe beenden
siNode.complete();
```

29 com.levigo.jadice.server.pdfmerge.PDFMergeNode

```
// Auf Antwort vom Server warten
for (Stream stream : soNode.getStreamBundle()) {
  // Lesen der Daten
  InputStream is = stream.getInputStream();
}
```
Code Beispiel

5.3.6. Konvertierung nach TIFF

Die meisten Konvertierungsvorgänge (z. B. OpenOffice, Shaper) erzeugen PDF. Es ist jedoch möglich, durch das Einfügen des JadiceShaperNode[30] das Ergebnis weiter nach TIFF zu konvertieren.

Im folgenden Beispiel wird der Workflow aus Kapitel 5.3.5. verändert; anstelle des PDFMergeNode wird eine Konvertierung nach TIFF mit anschließender Aggretation angehängt:

```
(…)
JadiceShaperNode shaperNode = new JadiceShaperNode();
// gewünschtes Zielformet
shaperNode.setTargetMimeType(„image/tiff“);
// alle eingehenden Streams zusammenfassen
shaperNode.setOutputMode(OutputMode.JOINED);
// Workflow erstellen, Tiff-Konverter-Node einfügen
job.attach(siNode.
  appendSuccessor(shaperNode).
  appendSuccessor(soNode));
(…)
```
Code Beispiel

5.3.7. Entpacken von Archivdateien

Um die Netzwerklast zu verringern, werden Dateien häufig komprimiert. Diese können vor der Weiterverarbeitung durch den jadice server entpackt werden. Dies geschieht je nach Dateiformat in unterschiedlichen Nodeklassen:

Dateiformat	Nodeklasse	Bemerkung
ZIP	com.levigo.jadice.server.archive.UnZIPNode	
RAR	com.levigo.jadice.server.archive.UnRARNode	
GZIP	com.levigo.jadice.server.archive.UnGZIPNode	.tar.gz-Dateien müssen zuerst durch den GZIP-, anschließend den TAR-Node
TAR	com.levigo.jadice.server.archive.UnTARNode	

Wie dies für den UnZIPNode aussieht, zeigt folgendes Codebeispiel:

```
Job job = createServerJob(); // (siehe 5.3.1.)
// Nodes erstellen:
// 1. Dateneingabe-Node
StreamInputNode siNode = new StreamInputNode();
// 2. Entpacken von ZIP-Archiven
UnZIPNode unzipNode = new UnZIPNode();
// 3. Ausgabe-Node
StreamOutputNode soNode = new StreamOutputNode();
// Workflow erstellen
job.attach(siNode.
  appendSuccessor(unzipNode).
  appendSuccessor(soNode));
// Job ausführen
job.submit();
// Dokument-Datenstrom senden
```

30 com.levigo.jadice.server.jadice4x.JadiceShaperNode

```
siNode.addStream(<zipped_Inputstream>);
// Dateneingabe beenden
siNode.complete();
// Auf Antwort vom Server warten
for (Stream stream : soNode.getStreamBundle()) {
  // Lesen der Daten (1 Stream je Datei im Archiv)
  InputStream is = stream.getInputStream();
}
```

Code Beispiel

5.3.8. Konvertierung unbekannter Eingabedaten in ein einheitliches Format (PDF)

Eine Vereinheitlichung von Dokumenten ist besonders im Bereich der Langzeitarchivierung von Nutzen. Der Zugriff auf die Datenquelle, die automatische Analyse von Daten, eine zielvorgabenorientierte, dynamische Weiterverarbeitung und eine abschließende Archivierung ins Archiv bringt folgende Vorteile:

Die aufrufende Anwendung braucht keinerlei Kenntnis über Quelldateien und Formate. Es besteht keine Gefährdung durch bösartige Daten oder Dokumente. Darüber hinaus ist eine Minimierung des Netzwerktransfers die Folge. Durch seine Struktur ermöglicht es der jadice server, zu jeder Zeit das Konvertierungsergebnis flexibel zu steuern.

```
Job job = createServerJob(); // (siehe 5.3.1.)
// Nodes erstellen:
//  1. Dateneingabe-Node
StreamInputNode siNode = new StreamInputNode();
//  2. Analyse-Node
DynamicPipelineNode dpNode = new DynamicPipelineNode();
dpn.setRulesetName("default");
   //  3. Zusammenfassen von Eingabedaten (1…n zu 1)
PDFMergeNode pmNode = new PDFMergeNode();
//  4. Ausgabe-Node
StreamOutputNode soNode = new StreamOutputNode();
// Workflow durch Verkettung der Nodes erstellen
job.attach(siNode.
  appendSuccessor(dpNode).
  appendSuccessor(pmNode).
  appendSuccessor(soNode));
// Job ausführen
job.submit();
// Dokument-Datenstrom senden
siNode.addStream(<InputStream>);
// Dateneingabe beenden
siNode.complete();
// Auf Antwort vom Server warten
for (Stream stream : soNode.getStreamBundle()) {
  // Lesen der Daten
  InputStream is = stream.getInputStream());
}
```

Code Beispiel

5.3.9. Konvertierung von OpenOffice-Dokumenten nach PDF

```
Job job = createServerJob(); // (siehe 5.3.1.)
// Nodes erstellen:
//  1. Dateneingabe Node
StreamInputNode siNode = new StreamInputNode();
//  2. OpenOffice-Konvertierung Node[31]
OOfficeConversionNode oocNode =
```

31 Der Klassenpfad muss wie in Kapitel 4.1.7. beschrieben gesetzt werden.

```
  new OOfficeConversionNode();
// 3. Zusammenfassen von Eingabedaten (1…n zu 1)
PDFMergeNode pmNode = new PDFMergeNode();
// 4. Ausgabe Node
StreamOutputNode soNode = new StreamOutputNode();
// Workflow erstellen
job.attach(siNode.
  appendSuccessor(oocNode).
  appendSuccessor(pmNode).
  appendSuccessor(soNode));
// Job ausführen
job.submit();
// Dokument Datenstrom senden
siNode.addStream(is);
// Dateneingabe beenden
siNode.complete();
// Auf Antwort vom Server warten
for (Stream stream : soNode.getStreamBundle()) {
  // Lesen der Daten
  InputStream is = stream.getInputStream());
}
```

Code Beispiel

Hinweis: Dokumente im Word2007-Format (Dateiendung .docx) müssen vor der Konvertierung mit OpenOffice durch den StreamAnalysisNode (vgl. Kapitel 5.3.3.).

5.3.10. Konvertierung von E-Mails nach PDF

Bei der E-Mailkonvertierung wird die E-Mail direkt vom Mailserver geholt. Hierzu müssen die entsprechenden Zugriffsdaten angegeben werden.

Der Vorgang ist ähnlich der dynamischen Konvertierung (siehe Kapitel 5.3.8.). Die E-Mail wird analysiert, eventuelle Anhänge wie z. B. Office-Dokumente, Bilder usw. werden alle konvertiert, in einer Übersicht zusammengefasst und an den E-Mail-Text angehängt.

Archivdateien werden dabei ausgepackt und deren Inhalt in den Konvertierungsvorgang eingebunden.

```
Job job = createServerJob(); // (siehe 5.3.1.)
// Nodes erstellen:
//  1. Eingabe-Node, hier wird serverseitig ein
//     Mailserver angesprochen.
JavamailInputNode jiNode = new JavamailInputNode();
// Mailserver-spezifische Daten setzen
jiNode.setStoreProtocol(<Protokol>); // POP3 oder IMAP
jiNode.setHostName(<Server>);
jiNode.setUsername(<Benutzer>);
jiNode.setPassword(<Passwort>);
jiNode.setFolderName(<E-Mail Ordner>);
jiNode.setImapMessageUID(<E-Mail ID>);
//  2. Analyse-Node mit Skript zur E-Mail-Konvertierung
ScriptNode scNode = new ScriptNode();
scNode.setScript(new URI("resource:
  email-conversion/EmailConversion.groovy")));
// 3. Zusammenfassen von Eingabedaten (1…n zu 1)
PDFMergeNode pmNode = new PDFMergeNode();
// 4. Ausgabe-Node
StreamOutputNode soNode = new StreamOutputNode();
// Workflow erstellen
job.attach(jiNode.
  appendSuccessor(scNode).
  appendSuccessor(pmNode).
```

```
  appendSuccessor(soNode));
// Job ausführen
job.submit();
// Auf Antwort vom Server warten
for (Stream stream : soNode.getStreamBundle()) {
  // Lesen der Daten
  InputStream is = stream.getInputStream();
}
```
Code Beispiel

Sollen E-Mails nicht mit dem JavamailInputNode über einen IMAP- oder POP3-Account abgerufen, sondern z. B. als eml-Datei eingelesen werden, muss zusätzlich der MessageRFC822Node[32] zwischengeschaltet werden, der die Abtrennung von E-Mail-Header und -Body vornimmt:

```
Job job = createServerJob(); // (siehe 5.3.1.)
// Nodes erstellen:
//  1. Eingabe-Node, hier als Datei vom Client.
StreamInputNode siNode = new StreamInputNode();
//  2. Trennung von E-Mail-Header und -Body
MessageRFC822Node msgNode = new MessageRFC822Node();
//  3. Analyse-Node mit Skript zur E-Mail-Konvertierung
ScriptNode scNode = new ScriptNode();
scNode.setScript(new URI("resource:
  email-conversion/EmailConversion.groovy")));
// Rest s.o.
```
Code Beispiel

In der oben gezeigten Konfiguration wird standardmäßig zu jedem Dateianhang eine Trennseite generiert, die die Metadaten des jeweiligen Anhangs enthält. Sind keine Trennseiten gewünscht, können diese mit der folgenden Konfiguration des ScriptNode für alle Dateianhänge deaktiviert werden:

```
(…)
scNode.getParameters().put
  ("showAttachmentSeparators", false));
(…)
```
Code-Beispiel

Eine weitere Konfigurationsmöglichkeit betrifft formatierte E-Mails. Wurden diese sowohl im HTML- als auch Plaintext-Format gesendet, wird standardmäßig der HTML-Teil konvertiert. Soll stattdessen der Plaintext-Teil konvertiert werden, ist folgende Konfiguration des ScriptNode vorzunehmen:

```
(…)
scNode.getParameters().put
  ("preferPlainTextBody", true));
(…)
```
Code-Beispiel

Davon unabhängig ist es möglich, denjenigen Teil, der normalerweise nicht konvertiert wird, als zusätzliches Attachment an die E-Mail anzuhängen. Somit kann die konvertierte E-Mail sowohl im HTML- als auch im Plaintext-Format dargestellt werden. Die nötige Konfiguration ist:

```
(…)
scNode.getParameters().put
  ("showAllAlternativeBody", true));
(…)
```
Code-Beispiel

Um zu verhindern, dass jadice server Bilder und andere in E-Mails referenzierte Dateien von unbekannten Quellen nachlädt, kann dies über folgende Einstellung verhindert werden:

32 com.levigo.jadice.server.javamail.MessageRFC822Node

```
(…)
scNode.getParameters().put
  ("allowExternalHTTPResolution", false));
(…)
```
Code-Beispiel

Die Behandlung von Attachments, deren Format nicht erkannt wurde oder die nicht durch den jadice server konvertiert werden können, kann über den Parameter **unhandledAttachmentAction** gesteuert werden:

```
(…)
scNode.getParameters().put
  ("unhandledAttachmentAction", "failure"));
(…)
```
Code-Beispiel

Folgende Werte werden hierbei akzeptiert:

Wert	Bedeutung
warning	Es wird eine Warnung in das Log geschrieben.
error	Es wird ein Error in das Log geschrieben (Standardwert).
failure	Der zugehörige Job bricht mit einem Fehler ab.

Zur Kenntlichmachung von Bilddateien, die in einer E-Mail referenziert sind, aber nicht konvertiert wurden, werden anstelle dieser folgende Platzhalter eingefügt:

Wert	Bedeutung
http://example.com/restricted.jpeg	Das Bild wurde aufgrund der Einstellung „allowExternalHTTPResolution" nicht geladen (s. o.)
cid://failed.jpeg	Die Bilddatei konnte nicht geladen werden.

5.3.11. Ansteuerung externer Programme

Durch den ExternalProcessCallNode[33] ist die Ansteuerung externer Programme sehr einfach möglich. Dabei kümmert sich der jadice server automatisch darum, dass ein- und ausgehende Datenströme automatisch zu temporären Dateien umgewandelt und diese nach der Verarbeitung durch das externe Programm gelöscht werden.

Einzige Voraussetzung ist, dass das Programm auf dem Server über Kommandozeile angesprochen werden kann:

```
Job job = createServerJob(); // (siehe 5.3.1.)
// Nodes erstellen:
//  1. Dateneingabe-Node
StreamInputNode siNode = new StreamInputNode();
//  2. Externer Prozess
ExternalProcessCallNode epcNode =
  new ExternalProcessCallNode();
// Konfiguration:
//    Programmname (Backslashes müssen escaped werden!)
epcNode.setProgramName(
  "C:\\Programme\\MyConverter\\MyConverter.exe");
//    Kommandozeilen-Parameter
//    ${infile} und ${outfile} ersetzt jadice server
epcNode.setArguments(
  "-s -a ${infile} /convert=${outfile}");
//    Dateiendungen, falls vom Programm benötigt
epcNode.setInfileExtension(".foo");
epcNode.setOutfileExtension(".pdf");
```

33 com.levigo.jadice.server.external.ExternalProcessCallNode

```
//    Flag, dass Ausgabedatei erzeugt wird
epcNode.setOutputStreamsExist(true);
// 3. Ausgabe-Node
StreamOutputNode soNode = new StreamOutputNode();
// Workflow erstellen
job.attach(siNode.
  appendSuccessor(epcNode).
  appendSuccessor(soNode));
job.submit();
// Auf Antwort vom Server warten
for (Stream stream : soNode.getStreamBundle()) {
  // Lesen der Daten
  InputStream is = stream.getInputStream();
}
```

5.4. Implementierung eigener Nodes / Worker

In diesem Kapitel soll anhand eines einfachen Beispiels gezeigt werden, wie der jadice server um eigene Nodes bzw. Worker erweitert werden kann, um damit neue Verarbeitungsschritte realisieren zu können.

Dazu sind im Wesentlichen zwei Schritte notwendig: Zunächst muss eine Node-Klasse implementiert werden, die sowohl auf Client- als auch auf der Serverseite vorhanden ist (siehe Kapitel 5.4.1.). Danach muss die korrespondierende Worker-Klasse implementiert werden (siehe Kapitel 5.4.2.). Diese muss nur auf der Serverseite vorhanden sein.

5.4.1. Node-Klasse

Die neu zu erstellende Nodeklasse muss von der abstrakten Superklasse Node[34] erben. Sie muss einen parameterlosen Konstruktor („default constructor") besitzen.

Außerdem kann die Methode getWorkerClassName() überschrieben werden. Als standardmäßigen Rückgabewert liefert diese als Rückgabewert den voll qualifizierten Klassenname des Node, wobei „Node" durch „Worker" ersetzt wird sowie „worker" als zusätzliche Namespace-Ebene eingefügt wird (Bsp: com.acme.jadiceserver.ExampleNode.getWorkerClassName() liefert „com.acme.jadiceserver.worker.ExampleWorker").

Haben Sie eine andere Paketstruktur gewählt, so kann diese Methode überschrieben werden, um den voll qualifizierten Klassennamen der korrespondierenden Workerklasse zu liefern:

```
package com.myCompany.jadice.client;
import com.levigo.jadice.server.Node;

public class DemoNode extends Node {
  public String getWorkerClassName() {
    // Klassenname der Workerklasse aus Bsp unten
    // Standardrückgabewert wäre
    // "com.myCompany.jadice.client.worker.DemoWorker"
    return "com.myCompany.jadice.worker.DemoWorker";
  }
}
```

Code-Beispiel: Implementierung eines Nodes

Soll es möglich sein, dass dem Worker zur Laufzeit Parameter übermittelt werden, so kann dies durch weitere Methoden in der Node-Implementierung erfolgen. Dabei ist zu beachten, dass alle Objekt- und statischen Attribute das Interface Serializable[35] implementieren müssen, da diese über JMS serialisiert und transportiert werden (siehe Kapitel 4.1.5.).

34 com.levigo.jadice.server.Node
35 java.io.Serializable

```
  public String getMyParameter() {
    // Sollte z.B. über Setter-Methode gesetzt werden
    return "a Parameter";
  }
```

Code-Beispiel: Erweiterung des Nodes aus obigem Bsp. um einen Parameter

Der selbst implementierte Node muss sowohl client- als auch serverseitig im Klassenpfad eingebunden werden und kann genau wie die im Kapitel 5.3. gezeigten Nodes in eigene Workflows eingebettet werden.

5.4.2. Worker-Klasse

Die Workerklasse, in der die Konvertierung durchführt wird, erbt von der abstrakten, generischen Superklasse NodeWorker<N>[36], wobei der Typ-Parameter <N> für die zugehörige Node-Klasse steht.

Hier ist die abstrakte Methode work() zu implementieren, in der die serverseitige Konvertierung durchführt wird.

```
package com.myCompany.jadice.server;

// Hier nur die wichtigen Imports
import com.levigo.jadice.server.core.NodeWorker;
import com.myCompany.jadice.client.DemoNode;

public class DemoWorker extends NodeWorker<DemoNode> {

  protected void work() throws Throwable {
    // Im Bsp oben definierter Parameter
    String myParam = getNode().getMyParameter();

    // Abholen der Eingabedaten
    for (Stream stream : getInputBundle()) {
      InputStream unprocessedIS =
        stream.getInputStream();
      // Metadaten des empfangenen Datenstroms
      StreamDescriptor unprocessedSD =
        stream.getDescriptor();

      // Methode, die den Datenstrom verarbeitet
      // (nicht im Listing gezeigt)
      InputStream processedIS =
        process(unprocessedIS, myParam);

      // Metadaten des verarbeitenden Datenstroms
      // unprocessedSD wird als „Parent" gesetzt
      StreamDescriptor processedSD =
        new StreamDescriptor(unprocessedSD);
      processedSD.setDescription("<Beschreibung>");
      processedSD.setMimeType("<MIME Type>");
      processedSD.setFileName("<Dateiname>");

      // Verknüpfen von Ergebnis und Metadaten
      Stream result =
        new BundledStream(processedIS, processedSD);
      // Weitergabe des Ergebnisses
      getOutputBundle().addStream(result);
    }
}
```

Code-Beispiel: Implementation eines Workers

36 com.levigo.jadice.server.core.NodeWorker<N>

Der auf diese Weise implementierte Worker muss nur im Klassenpfad des jadice servers eingebunden werden und wird bei Verwendung des zugehörigen Nodes aus dem vorherigen Kapitel automatisch aufgerufen.

6. Webservice-Schnittstelle

Um die volle Funktionalität, die jadice server bietet, Clients unabhängig von ihrer Implementierungssprache anbieten zu können, wurde in Version 4.2.0.0 eine Webservice-Schnittstelle eingeführt.

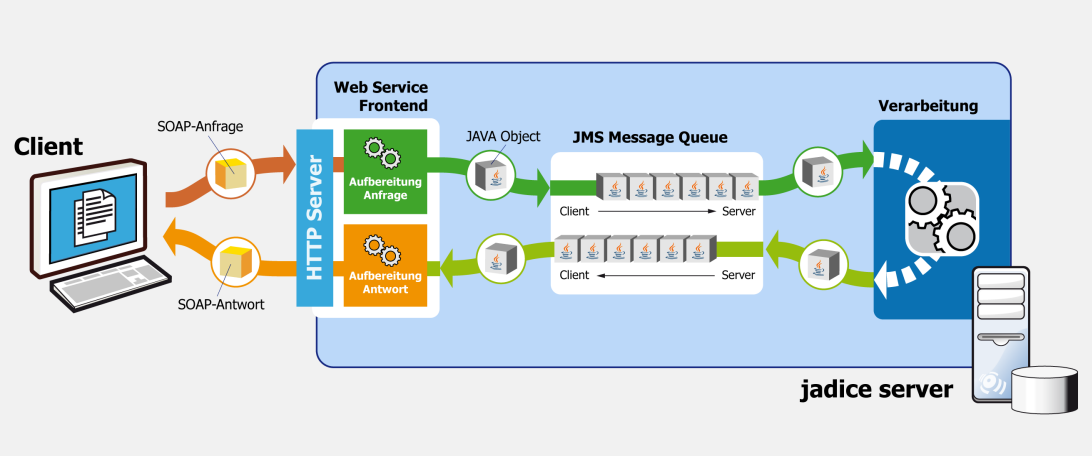


Abbildung 4: Schematischer Aufbau der Webservice-Schnittstelle

Die Kommunikation mit jadice server – beziehungsweise dem Webservice-Frontend – findet dabei über das HTTP-Protokoll mittels SOAP-Nachrichten im XML-Format statt. Die Übertragung von Dateien, die konvertiert werden sollen oder als Ergebnis an den Client zurückgesendet werden, ist als MTOM-Attachment[37] realisiert.

Zur Entwicklung und zum Debugging von SOAP-Anfragen empfehlen wir soapUI[38].

6.1. Aufbau einer SOAP-Nachricht

Nachdem wie in Kapitel 4.1.13. beschrieben die Webservice-Schnittelle aktiviert wurde, kann unter http://<url>?wsdl (z. B. bei Konfiguration wie im o. g. Kapitel: http://localhost:9000/jadiceServer?wsdl) die formale Beschreibung der Schnittstelle im WSDL-Format[39] heruntergeladen werden. Damit kann in vielen Webservice-Frameworks Code generiert werden, um den Webservice des jadice servers anzusprechen. Bei einer SOAP-Anfrage kann jadice server auf unterschiedliche zwei Arten angesprochen werden:

- Der Workflow wird anhand eines Templates, das zuvor serverseitig abgelegt wurde, vorkonfiguriert
- Der Workflow wird zur Laufzeit innerhalb der SOAP-Anfrage definiert.

Die beiden Möglichkeiten werden in den beiden folgenden Kapiteln dargestellt.

6.1.1. Anfrage anhand eines Templates

Es ist möglich, serverseitig eine in XML kodierte Jobbeschreibung abzulegen, sodass ein Client bei einer SOAP-Anfrage auf diese verweisen kann und der Job nicht zur Laufzeit konfiguriert werden muss.

Der Aufbau dieser Nachricht soll anhand des folgenden Beispiels erläutert werden:

```
<soapenv:Envelope xmlns:soapenv=
      "http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:ws="http://ws.server.jadice.levigo.com/">
 <soapenv:Header/>
 <soapenv:Body>
```

37 Siehe http://www.w3.org/TR/soap12-mtom/
38 Siehe http://www.soapui.org/
39 Siehe http://www.w3.org/TR/wsdl20-primer/

```
  <ws:run>
   <job templateLocation=
      "resource:/jobtemplates/x2pdf.xml">
    <property name="dp.rulesetName">default</property>
    <property name="dp.targetMimeType">
      application/pdf</property>
    <property name="dp.timeout">8000</property>
    <stream mimeType="unknown/*" uuid="123456789"
      nodeId="node0">
     <documentData>cid:0001</documentData>
    </stream>
   </job>
  </ws:run>
 </soapenv:Body>
</soapenv:Envelope>
```

Beispiel einer SOAP-Anfrage mit Job-Template

Neben den vom SOAP-Standard vorgegebenen Elementen für Header und Body gibt es das spezifische Element <run>, das die vom Webservice angebotene Methode run adressiert.

Darin wird ein Job (siehe Kapitel 3.) definiert, der über ein Template vordefiniert ist (siehe Kapitel 6.3.). Im Attribut **templateLocation** steht dabei der Ort, an dem das jeweilige Template serverseitig zu finden ist. Sind im Template Variablen definiert, so können diese mittels **property**-Elementen konfiguriert bzw. deren Standardwert überschrieben werden. Optional ist das Attribut **messageID**. Dies kann der Client frei vergeben und wird ggf. in einer Antwort des Servers übernommen.

Zu verarbeitende Datenströme werden über **stream**-Elemente in der SOAP-Nachricht referenziert. Die Angabe einer eindeutigen ID (**uuid**) und des MIME-Typen sind optional. Sollte der MIME-Type nicht bekannt sein, aber dennoch angegeben werden, so ist dort „unknown/*" anzugeben.

Sind in der Templatedatei mehrere StreamInputNodes[40] definiert, so muss eine eindeutige Zuordnung getroffen werden, welcher Datenstrom an welchen StreamInputNode gesendet wird. Dies erfolgt durch das Attribut **nodeId**. Dies verweist auf die ID, die dem StreamInputNode innerhalb des Templates (Attribut **id**) gegeben wurde.

Die eigentlichen Daten folgen in einem multipart/related-Container, der die hier angegebene CID (content ID) besitzt.

6.1.2. Jobdefinition innerhalb der SOAP-Nachricht

Soll der Client keine serverseitig vordefinierte Jobkonfiguration verwenden, ist es möglich, diese in der SOAP-Anfrage einzubetten. Das Format ist hierbei das selbe, wie innerhalb eines separaten Job-Templates (vgl. Kapitel 6.3.). Anstelle des Root-Elements **job** wird hierbei die Definition als **configuration**-Element in die SOAP-Anfrage eingebettet.

Das folgende Beispiel soll dies veranschaulichen:

```
<soapenv:Envelope xmlns:soapenv=
      "http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:ws="http://ws.server.jadice.levigo.com/">
 <soapenv:Header/>
 <soapenv:Body>
  <ws:run>
   <job messageID="4711">
    <configuration>
     <nodes>
      <node class=
       "com.levigo.jadice.server.nodes.StreamInputNode"
       id="input1" />
      <node class=
```

40 com.levigo.jadice.server.nodes.StreamInputNode

```
        "com.levigo.jadice.server.nodes.StreamInputNode"
       id="input2" />
       <node class=
      "com.levigo.jadice.server.nodes.DemultiplexerNode"
       id="demux" />
       <node class=
       "com.levigo.jadice.server.nodes.StreamOutputNode"
       id="out" />
      </nodes>
      <connections>
       <connect from="input1" to="demux" />
       <connect from="input2" to="demux" />
       <connect from="demux" to="out" />
      </connections>
     </configuration>
     <stream nodeId="input1">
      <documentData>cid:abc</documentData>
     </stream>
     <stream nodeId="input2">
      <documentData>cid:def</documentData>
     </stream>
    </job>
  </ws:run>
</soapenv:Body>
</soapenv:Envelope>
```

Beispiel einer SOAP-Anfrage mit eingebetteter Job-Definition

In diesem Beispiel werden zwei StreamInputNodes miteinander über einen DemultiplexerNode[41] gekoppelt und die Eingabedaten unverändert an den Client zurückgesendet.

Die Definition der Nodes und welchen Workflow-Graphen sie bilden, ist innerhalb des Blocks **configuration** beschrieben.

Hier ist außerdem zu sehen, wie es möglich ist, bestimmte Eingabeströme an einen StreamInputNode zu binden: Das erste Dokument (cid:abc) wird an den ersten (nodeId input1), das zweite Dokument (cid:def) an den zweiten StreamInputNode (nodeId input2) gebunden.

6.2. Aufbau einer SOAP-Antwort

Der Aufbau einer Antwort, die dem Client als Antwort auf eine Anfrage geschickt wird, ist auch in der oben genannten WSDL spezifiziert.

Eine mögliche Antwort könnte so aussehen:

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <ns2:runResponse xmlns:ns2=
     "http://ws.server.jadice.levigo.com/">
   <return>
    <stream>
     <documentData>56ea1e8d-114e-4265@apache.org
     </documentData>
    </stream>
    <status>COMPLETED</status>
   </return>
  </ns2:runResponse>
 </soap:Body>
</soap:Envelope>
```

Beispiel einer SOAP-Antwort

41 com.levigo.jadice.server.nodes.DemultiplexerNode

Neben einer (eventuell leeren) Menge von Ergebnisströmen, die jeweils über eine eindeutige ID in einem multipart/related-Container referenziert werden, ist dort eine Status-Meldung vorhanden. Folgende Werte sind möglich:

Wert	Bedeutung
COMPLETED	Job wurde durchgeführt.
FAILED	Job konnte nicht durchgeführt werden.

In beiden Fällen kann das **return**-Element eine Menge an **log-entry**-Elementen enthalten, die Hinweise über den Fehlschlag der Verarbeitung oder Meldungen, die während der Verarbeitung aufgetreten sind, enthalten (vgl. Kapitel 5.3.2. „Erstellung eines JobListeners"). Die Fehlermeldung, die gezeigt wird, wenn eine nicht vorhandene Jobtemplate-Datei referenziert wird, zeigt das folgende Beispiel:

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
   <ns2:runResponse xmlns:ns2=
      "http://ws.server.jadice.levigo.com/">
    <return messageID="12345">
     <log-entry timeStamp="31.12.2009 22:33:44">
      <level>ERROR</level>
      <id>JS.WEBSERVICE-EXCEPTION</id>
      <message>java.io.FileNotFoundException: Could not
        locate resource: does_not_exit.xml</message>
     </log-entry>
    <status>FAILED</status>
   </return>
  </ns2:runResponse>
 </soap:Body>
</soap:Envelope>
```

Code-Beispiel einer Fehlermeldung

6.3. Definition von Job-Templates

Durch die Definition von Job-Templates wurde die Möglichkeit geschaffen, dass Clients nun nicht mehr die internen Schritte des jadice servers, die für eine Konvertierung notwendig sind, kennen müssen. Diese werden für die Webservice-Schnittstelle an zentraler Stelle vorgehalten werden. Dadurch muss der Client nunmehr nur noch die Webservice-Methode „run" sowie den Ort des auszuführenden Templates kennen (diese liegen i.d.R. im Unterordner server-config/).

Im Ordner server-config/jobtemplates liegt die XSD-Definition für diese Templates.

Ein Beispiel, wie solch ein Template aussehen kann, ist das im Lieferumfang enthaltene Template x2pdf.xml, das ähnlich dem Beispiel auf Kapitel 5.3.8. unbekannte Eingabedaten identifiziert und in das PDF-Format umwandelt:

```
<job
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="jobtemplate.xsd">
 <properties>
  <property name="PROPERTY_KEY_COMMUNICATION_TIMEOUT">
   ${communication_timeout:22000}
  </property>
 </properties>
 <nodes>
  <node class="com.levigo.jadice.server.
    nodes.StreamInputNode" id="node0">
   <property name="timeout">${timeout:6000}</property>
  </node>
  <node class="com.levigo.jadice.server.
```

```
    nodes.DynamicPipelineNode" id="node1">
   <property name="rulesetName">
    ${rulesetName:default}</property>
   <property name="targetMimeType">
    ${targetMimeType:application/pdf}</property>
   <property name="timeout">${timeout:6000}</property>
  </node>
  <node class="com.levigo.jadice.server.
    nodes.StreamOutputNode" id="node2">
   <property name="timeout">${timeout:6000}</property>
  </node>
 </nodes>
 <connections>
  <connect from="node0" to="node1" />
  <connect from="node1" to="node2" />
 </connections>
</job>
```

Beispiel eines Job-Templates

Wie man erkennen kann, ist ein Template in drei Blöcke gegliedert:

- Eigenschaften, die den kompletten Job betreffen (Timeouts etc)
- Definition von einzelnen Nodes (Element <node>) und deren Eigenschaften
- Verkettung der Nodes zu einem Workflow

Da die einzelnen Nodes den Konventionen für java beans entsprechen, können die jeweiligen Eigenschaften hier einfach über deren Namen gesetzt werden. Außerdem ist es möglich, diese variabel zu machen, wie im obigen Beispiel zu sehen ist. Dies geschieht nach folgendem Muster:

${<Bezeichner>}** oder **${<Bezeichner>:<Standardwert>}

wobei der Bezeichner zwingend mit einem Buchstaben beginnen muss und als weitere Zeichen Buchstaben, Ziffern, „_" (Unterstrich), „-" (Bindestrich) und „." (Punkt) haben darf.

Durch diesen Bezeichner können diese Werte beim SOAP-Aufruf als Eigenschaft (**Element <property name="bezeichner">wert</...>**) gesetzt bzw. überschrieben werden. Werden Variablen, die keinen Standardwert haben, nicht gesetzt, so führt dies beim Aufruf zu folgendem Fehler:
„com.thoughtworks.xstream.converters.ConversionException: Pattern refers to undefined variable <Bezeichner> for which there is no default".

Die einzelnen Node-Elemente müssen eine für das jeweilige Template eindeutige ID einhalten. Durch diese werden sie im <connections>-Block zu einem Workflow verkettet.

Sollen über den SOAP-Aufruf, der zu diesem Template gehört, Daten vom Client zum Server übertragen werden, ist es notwendig, mindestens einen StreamInputNode zu definieren. Werden mehrere StreamInputNodes definiert, so müssen die einzelnen stream-Elemente in der SOAP-Anfrage den jeweils zutreffenden Node über das Attribut **nodeId** referenzieren. Dies entfällt, wenn es genau einen StreamInputNode gibt.

Die Datenströme, die aus StreamOutputNodes resultieren, werden als MTOM-Attachments in der SOAP-Antwort an den Client zurückgesendet. Dabei ist es auch möglich, mehrere StreamOutputNodes zu definieren. Dabei ist die Reihenfolge, in der die StreamOutputNodes abgefragt werden, um ihre Datenströme an die SOAP-Antwort anzuhängen, zufällig.

Um Job-Templates in eine SOAP-Anfrage einzubetten (siehe Kapitel 6.1.2.), muss das Root-Element job entfernt werden; der Inhalt wird stattdessen an das Element **configuration** in der SOAP-Anfrage eingehängt.

6.4. Generierung von Webservice-Clients

Da die Webservice-Schnittstelle durch die WSDL klar definiert ist, können frei verfügbare Webservice-Bibliotheken diese Definition verarbeiten und daraus Proxy-Klassen generieren, die die notwendigen SOAP-Anfragen kapseln und dadurch eine effiziente Entwicklung von Clientanwendungen ermöglichen. In diesem Kapitel wird dies anhand Suns Referenzimplemenentierung von JAX-WS und der Bibliothek Apache Axis2 gezeigt.

6.4.1. JAX-WS Referenzimplemenentierung

Im Lieferumfang des Java Development Kit (JDK) Version 1.6 befindet sich das Kommandozeilentool **wsimport**, mit dem Proxy-Klassen generiert werden können. Ist der Webservice von jadice server wie in Kapitel 4.1.13. beschrieben aktiviert worden, werden mit folgendem Aufruf die notwenden Clientklassen erzeugt:

```
<jdk1.6.0>\bin\wsimport
  -keep
  http://localhost:9000/jadiceServer?wsdl
```

Der Schalter „-keep" bewirkt, dass nicht nur die Klassen, sondern auch deren Quelltexte gespeichert werden. Zur weiteren Entwicklung empfehlt es sich, mit diesen weiter zu arbeiten. Abbildung 5 zeigt die generierten Klassen, die in die Entwicklungsumgebung eingebunden werden können.

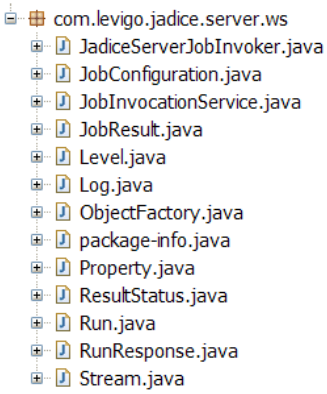


Abbildung 5: Von JAX-WS generierte Klassen

Einstiegspunkte für eine Clientanwendung sind die Klassen **JadiceServerJobInvoker** und **JobInvocationService**, über die ein Zugriff auf die SOAP-Schnittstelle erfolgt, und die Klasse **JobConfiguration**, über die der Aufruf konfiguriert wird. Eine minimale Implementierung kann so aussehen:

```
// Abstraktion der SOAP-Schnittstelle:
JadiceServerJobInvoker invoker =
  new JadiceServerJobInvoker();
JobInvocationService service =
   invoker.getJobInvocationServiceImplPort();

// Konfiguration des Jobs
JobConfiguration job = new JobConfiguration();
job.setTemplateLocation(
  "resource:/jobtemplates/x2pdf.xml");

// Optional: Setzen einer Property (Timeout als Bsp.)
Property timeout = new Property();
timeout.setName("timeout");
timeout.setValue("20000");
job.getProperty().add(timeout);

// Anhängen der Eingabedaten
// (sofern Template einen StreamInputNode besitzt)
Stream inputStream = new Stream();
```

```
inputStream.setDocumentData(…);   // Byte-Array
job.getStream().add(inputStream);

// Absetzen der SOAP-Anfrage (Methode blockiert)
JobResult result = service.run(job);

// Ergebnis-Status abfragen
ResultStatus status = result.getStatus();

for (Log log : result.getLogEntry()) {
  // Logeinträge auswerten…
}

for (Stream stream : result.getStream()) {
  // Ergebnis als Byte-Array
  byte[]  data = stream.getDocumentData();
}
```

Implementierung eines SOAP-Clients mit der JAX-WS Referenzimplemenentierung

6.4.2. Apache Axis2

Apache Axis2 ist unter **http://ws.apache.org/axis2/** verfügbar und steht unter der Apache License. Ist der Webservice von jadice server wie in Kapitel 4.1.13. beschrieben aktiviert worden, werden mit folgendem Aufruf die notwendigen Clientklassen erzeugt:

```
<AXIS2_HOME>\bin\wsdl2java
  -o generatedCode
  -p com.levigo.jadice.server.ws.client.axis2.stub
  -d jaxbri
  -uri http://localhost:9000/jadiceServer?wsdl
```

Die Verwendung der Schalter „-o" für das Ausgabeverzeichnis und „-p" für den zu verwendenden Paketnamen sind optional. Der Schalter „-d" bestimmt, welches Databinding für die Wandlung nach / von XML verwendet werden soll. Standard ist das Apache Axis Data Binding (ADB). Dieses hat in der aktuellen Fassung jedoch Probleme mit der Deserialisierung von SOAP/MTOM-Attachments, sodass stattdessen auf die JAX-B Referenzimplemenentierung (jaxbri) zurückgegriffen werden sollte.

Einstiegspunkte für eine Clientanwendung sind die Klassen J**adiceServerJobInvokerStub** und **Run**, über die ein Zugriff auf die SOAP-Schnittstelle erfolgt, und die Klasse **JobConfiguration**, über die der Aufruf konfiguriert wird. Eine minimale Implementierung kann so aussehen:

```
// Abstraktion der SOAP-Schnittstelle
JadiceServerJobInvokerStub invoker =
 new JadiceServerJobInvokerStub();
Run run = new Run();

// Konfiguration der Anfrage
run.setJob(…); // Typ JobConfiguration (s.o.)

// Absetzen der SOAP-Anfrage (Methode blockiert)
RunResponse response = invoker.run(run);

// Ergebnisobjekt abholen
JobResult jobResult = response.getReturn();

// Verarbeitung des Ergebnisses vgl. oben
```

Implementierung eines SOAP-Clients mit Apache Axis2

7. Monitoring

Durch die Unterstützung der Java Management Extensions (JMX) ist es möglich, jadice server im Betrieb zu überwachen und einige Einstellungen zur Laufzeit zu verändern. Wichtige Kernkomponenten sind hierbei als Managed Beans (MBeans) ausgelegt, sodass diese genau überwacht werden können.

Um die JMX-Schnittstelle zu aktivieren, müssen in der Datei **<jadice-server>/wrapper/wrapper.log** folgende Einträge ergänzt werden:

```
wrapper.java.additional.1[42]=-Dcom.sun.management.jmxremote.port=61619
wrapper.java.additional.2=-Dcom.sun.management.jmxremote.authenticate=false
wrapper.java.additional.3=-Dcom.sun.management.jmxremote.ssl=false
```

Bitte beachten Sie, dass in diesem Beispiel keine Authentisierung vorgeschrieben wird, sodass auch Unberechtigte über diese Schnittstelle auf jadice server zugreifen und den Betrieb beeinträchtigen könnten. Im Java SE Monitoring and Management Guide[43] ist beschrieben, wie die Authentisierung aktiviert werden kann.

Verbinden Sie sich über die Tools **JConsole** oder **Java VisualVM**, die Suns Java Runtime Environment beiliegen, mit jadice server. Die wichtigen Komponenten im Zweig „com.levigo.jadice.server" sind (siehe Abbildung 6):

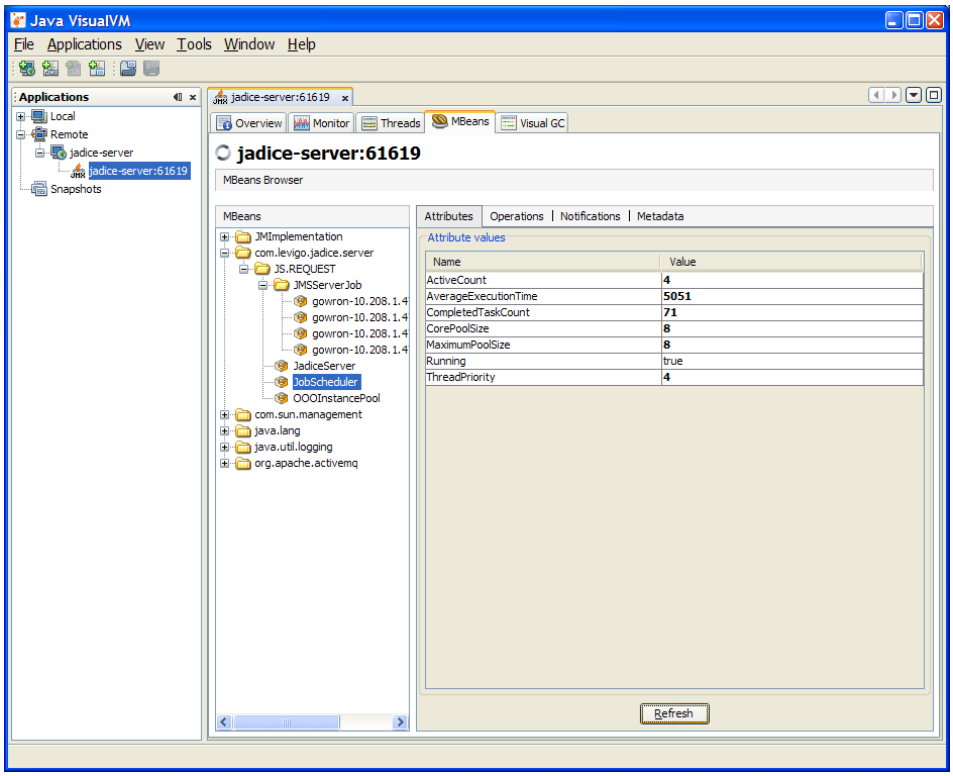


Abbildung 6: MBeans-Ansicht einer laufenden jadice server Instanz

- JobScheduler:
  Hier können statistische Daten zu abgearbeiteten und gerade laufenden Jobs angezeigt werden.
- Pools für MS Office und OpenOffice:
  Außer der Anzeige der gerade aktiven Office-Instanzen können die Pools

42 Die Nummerierung muss fortlaufend sein und darf sich nicht mit bereits vorhandenen Einträgen überschneiden.

43 Siehe http://java.sun.com/javase/6/docs/technotes/guides/management/toc.html

geleert und dadurch die Office-Instanzen beendet werden. Werden anschließend Jobs gestartet, die Office-Instanzen benötigen, werden jeweils neue gestartet und die jeweiligen Pools wieder gefüllt. Außerdem kann die Anzahl der maximal gestarteten Instanzen verändert werden[44].

- Unter JMSServerJob wird jeder gerade laufende Job gezeigt. Diese können über die Operation „abort" abgebrochen werden. Außerdem können statistische Daten über die verwendeten Nodes, die zugehörigen Threads und wie lange die einzelnen Jobs bereits laufen abgefragt werden.

44 Diese Angabe wird nicht persistiert. Bei einem Neustart von jadice server wird der in den Konfigurationsdateien angegebene Wert geladen.

8. Migration von früheren Versionen

Bei der Umstellung auf jadice server 4.2. wurden einige Änderungen an der API vollzogen, um eine klarere Struktur zu erhalten. Dadurch sind einige Anpassungen an Clients, die gegen die API früherer Versionen programmiert wurden, durchzuführen.

8.1. Von Version 4.1.x nach 4.2.0.0

8.1.1. Clientseitige Anpassungen

- Bibliotheken, die clientseitig eingebunden werden, müssen durch aktuelle Versionen ersetzt werden (Verzeichnis **/client-lib**)
- In den Nodes wurden folgende Methoden ersetzt[45]:

Klassenname	Alte Methode	Neue Methode
ScriptNode	setScriptURL(String scriptURL)	setScript(URI location)[46]
XSLFOFormatterNode	setStyleSheet(String styleSheet)	setStyleSheet(URI location)
SeparatorPageNode	setStylesheetURL(String stylesheetURL)	setStyleSheet(URI location)
	setOutputMimeType(String outputMimeType)	setTargetMimeType(String mimeType)
OOfficeConversionNode	setOutputMimeType(String outputMimeType)	setTargetMimeType(String mimeType)
URLInputNode	setMimeType(String mType)	— *entfällt* —
MS###Node (außer Outlook)	setExportFilter(String configuration)	setTargetMimeType(String mimeType) (vgl. Kapitel 8.1.2.)
MSVisioNode	setPassword(String)	— *entfällt* —
MSOutlookNode	setPassword(String)	— *entfällt* —
	setExportFilter(String)	— *entfällt* —

- Folgende Nodes wurden deprecated:

Veraltete Nodeklasse	Ersatzklasse	Bemerkungen
FopNode	XSLFOFormatterNode	
JadiceToTiffNode	JadiceShaperNode	setTargetMimeType(„image/tiff“)
OutputStreamWriterNode	URLOutputNode	
TIFFPageAggregatorNode	TIFFMergeNode	

- Folgende Nodes wurden ersetzt:

Weggefallene Nodeklasse	Ersatzklasse	Bemerkungen
HTMLToPDFRendererNode	HTMLRendererNode	
MSOutlook2003Node	MSOutlookNode	
MSWord2003Node	MSWordNode	
MSWord2003TextNode	MSWordNode	setOpenFormat(WdOpenFormat.WD_OPEN_FORMAT_TEXT) (vgl. Kapitel 8.1.2.)

45 In der Tabelle sind nur die jeweiligen Setter-Methoden aufgeführt; Getter-Methoden wurden analog ersetzt.

46 Der Methodenaufruf myScriptNode.setScriptURL(„<location>“) kann einfach durch myScriptNode.setScript(new URI(„location“)) ersetzt werden.

8.1.2. Änderungen an Konfigurationsdateien

- Die Konfiguration der MSOffice-Nodes erfolgt nun zentral über die Konfigurationsdatei **server-config/ms-office/export-configuration.xml**. Zu beachten ist hierbei, dass
  - der Zieltyp nun als MIME-Type angegeben werden muss.
  - PDF nun das Standardzielformat ist.

Bisher: MS###Node.setExportFilter(...)	Neu: MS###Node.setTargetMimeType(...)	Bemerkungen
pdf	application/pdf	Neues Standardzielformat
default	image/tiff	Bisheriges Standardzielformat
afp	application/afp	
— nicht vorhanden —	application/vnd.ms-xpsdocument	XML Paper Specification (XPS)

- Die Seitenkonfiguration für Plaintext-Dateien, die über den MSWordNode konvertiert werden, erfolgt nun zentral in **server-config/ms-office/style-configuration.xml**.
- XSL:FO-Stylesheets in **server-config/stylesheets/** entfallen und werden durch gleichnamige Stylesheets in **server-config/email-conversion/** ersetzt.

8.1.3. Änderungen eigener Nodes / Worker

- In der abstrakten Klassen Node wurde die Methode getNodeClassName() in getWorkerClassName() umbenannt. Diese ist nicht mehr abstrakt, sondern hat das Standardverhalten, das als Rückgabewert der voll qualifizierte Klassenname des Node geliefert wird, wobei „Node" durch „Worker" ersetzt wird sowie „worker" als zusätzliche Namespace-Ebene eingefügt wird. (Bsp: com.acme.jadiceserver.ExampleNode.getWorkerClassName() liefert „com.acme.jadiceserver.worker.ExampleWorker")
- Um sicherzustellen, dass diese Umbenennung bei der Integration nicht vergessen wird, was zu einem falschen Verhalten führen kann, wurde eine finale Methode mit dem alten Namen in der Klasse Node eingeführt. Dadurch wird ein Compilerfehler erzeugt, der beseitigt werden muss.

8.2. Von Versionen 4.2.0.0 bis 4.2.1.2 nach 4.2.1.3

8.2.1. Änderungen von MessageIDs im Client-Log

- Es entfallen alle MessageIDs, die im Zusammenhang mit der Konvertierung von HTML-E-Mails auftreten, bisheriger Prefix **JS.HTML.MAIL-**. Sie werden mit den MessageIDs für die Konvertierung von HTML-Dateien zusammengeführt, Prefix **JS.HTML-**.
- Die MessageID **CID_REFERENCE_RESOLVE_FAILURE** entfällt. Sie wird durch **CID_REQUEST_FAILED** ersetzt.

8.2.2. Deprecation

Die Methoden **Node.setTimeout(long)** bzw. **getTimeout()** wurden deprecated. Verwenden Sie stattdessen **Node.apply(new TimeLimit(...))**.

8.3. Von Versionen 4.2.0.0 bis 4.2.1.4 nach 4.2.1.5

Clients, deren jadice server-Bibliotheken aktualisiert werden, dürfen nicht mehr die Methode **GhostscriptNode#setSplitPages()**[47] verwenden. Stattdessen muss die

47 com.levigo.jadice.server.ghostscript.GhostscriptNode

Methode **GhostscriptNode#setOutputMode()** verwendet werden. Clients, die weiterhin alte Bibliotheken verwenden, aber mit der neusten Version von jadice server kommunizieren, sind von dieser Änderung jedoch nicht betroffen.

9. Troubleshooting

In diesem Kapitel werden einige der typischen Fehlersituationen, die beim Betrieb von jadice server auftreten können, gezeigt.

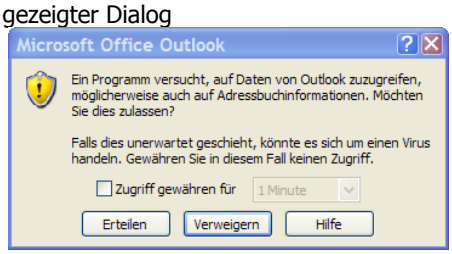
- Fehler bei der Konvertierung via MS Office
  - Fehlermeldung im Server-Log:

```
Exception in thread "main" java.Lang.UnsatisfiedLinkError: (…)\msoffice-lib\jacob-1.14M7-
x86.dll: Diese Anwendung konnte nicht gestartet werden, weil die Anwendungskonfiguration
nicht ordnungsgemäß ist. Zur Problembehandlung sollten Sie die Anwendung neu installieren.
```

  - Ursache
    Fehlende C++-Runtime
  - Lösung
    Installation des „Microsoft Visual C++ 2005 SP1 Redistributable Package (x86)", siehe Kapitel 4.1.8.
- Fehler bei der Konvertierung via MS Office
  - Fehlermeldung im Server-Log:

```
Processing failed: MSWordNode/MSWordWorker:
JS.SERVER-WORKER_FAILED: Verarbeitung für Knoten MSWordNode fehlgeschlagen wegen
com.jacob.com.ComFailException: Invoke of: exportAsFixedFormat
Source: Microsoft Word
Description: Fehler beim Exportieren, weil dieses Feature nicht installiert ist.
   at com.jacob.com.Dispatch.invokev(Native Method)
   at com.jacob.com.Dispatch.invokev(Dispatch.java:858)
   at com.jacob.com.Dispatch.callN(Dispatch.java:455)
   at com.levigo.jadice.server.msoffice.MSWordConverter.convert(MSWordConverter.java:103)
   at com.levigo.jadice.server.msoffice.CommandReceiver.run(CommandReceiver.java:110)
   at com.levigo.jadice.server.msoffice.MSWordConverter.main(MSWordConverter.java:31)
```

  - Ursache
    Kein nativer PDF-Export in MS Office 2007 installiert
  - Lösung
    Installation von „2007 Microsoft Office Add-in: Microsoft Save as PDF", siehe Kapitel 4.1.8.
- Dialog bei der Konvertierung via MS Outlook
  - gezeigter Dialog



  - Ursache
    Sicherheitsbestimmungen von MS Outlook verbieten Zugriff durch jadice server
  - Lösung
    Installation von „Advanced Security for Outlook", siehe Kapitel 4.1.9.
- Fehler bei der Konvertierung via MS Office
  - Fehlermeldung im Server-Log

```
java.lang.Exception: com.jacob.com.ComFailException: Invoke of: Execute
Source: Microsoft Word
Description: Druckerfehler.

   at com.jacob.com.Dispatch.invokev(Native Method)
   at com.jacob.com.Dispatch.invokev(Dispatch.java:858)
   at com.jacob.com.Dispatch.callN(Dispatch.java:455)
   at com.jacob.com.Dispatch.call(Dispatch.java:533)
   at com.levigo.jadice.server.msoffice.MSWordConverter.convert(MSWordConverter.java:152)
   at com.levigo.jadice.server.msoffice.CommandReceiver.run(CommandReceiver.java:110)
   at com.levigo.jadice.server.msoffice.MSWordConverter.main(MSWordConverter.java:31)
```

```
(…)
```

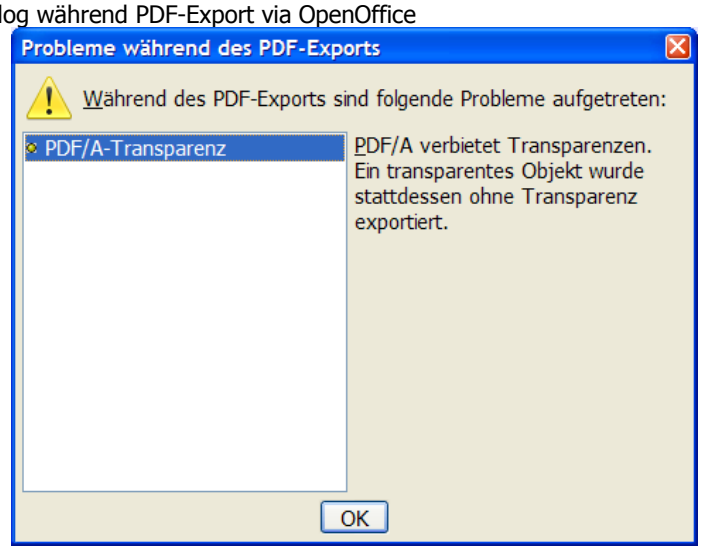
- Ursache
  Der Druckertreiber, über den der Export angestoßen wird, ist nicht korrekt konfiguriert. Eventuell gibt es auf dem Server-System keinen Drucker mit dem angegebenen Namen.
- Lösung
  Überprüfen Sie die Angaben in der Datei **server-config/ms-office/export-configuration.xml** und gleichen Sie diese mit den auf dem System installierten Druckern ab

- Fehler bei der Konvertierung via OpenOffice
  - Fehlermeldung im Server-Log

```
INFO  [OOOInstancePool] Creating an instance
ERROR [ManagedOOOInstance] Could not bootstrap
com.sun.star.comp.helper.BootstrapException: no office executable found!
   at com.levigo.jadice.server.ooffice.server.ManagedOOOInstance.
      launchOOOProcess(ManagedOOOInstance.java:102)
   at com.levigo.jadice.server.ooffice.server.ManagedOOOInstance.
      <init>(ManagedOOOInstance.java:86)
(…)
```

  - Ursache
    OpenOffice nicht ordnungsgemäß konfiguriert
  - Lösung
    Datei jadice-server-local.options anpassen, siehe Kapitel 4.1.7.

- Dialog während PDF-Export via OpenOffice



  - gezeigter Dialog
  - Ursache
    Dieser Dialog ist normal, wenn jadice server auf dem selben Rechner läuft, auf dem parallel die Client-Entwicklung stattfindet und dabei eine weitere Instanz von OpenOffice mit Fenstern offen ist.
  - Lösung
    Beenden Sie alle Instanzen von OpenOffice (soffice.exe / .bin im Task-manager) sowie des OpenOffice-Schnellstarters. Bei einer erneuten Konvertierung wird OpenOffice headless gestartet; der Dialog tritt nicht mehr auf.

- Konvertierung mit Nodes, die nicht standardmäßig im Produkt enthalten sind
  - Fehlermeldung im Server-Log

```
javax.jms.JMSException: Failed to build body from bytes.
   Reason: java.io.IOException: <Node-Klassenname>
   at org.apache.activemq.util.JMSExceptionSupport.create(JMSExceptionSupport.java:35)
   at org.apache.activemq.command.ActiveMQObjectMessage.
      getObject(ActiveMQObjectMessage.java:183)
   at com.levigo.jadice.server.core.JMSServerJob.<init>(JMSServerJob.java:267)
```

```
   at com.levigo.jadice.server.core.ThreadPoolJobScheduler$SchedulerThread.
      handleMessage(ThreadPoolJobScheduler.java:203)
   at com.levigo.jadice.server.core.ThreadPoolJobScheduler$SchedulerThread.
      run(ThreadPoolJobScheduler.java:122)
Caused by: java.io.IOException:
(…)
```

- Ursache
  Der vom Client erzeugte Job referenziert eine Node-Klasse, die nicht im Klassenpfad des Servers vorhanden ist.
- Lösung
  Überprüfen Sie den Klassenpfad des Servers und fügen Sie die fehlende Bibliothek hinzu.

- Fehlermeldung bei XML-Verarbeitung
  - Fehlermeldung im Server-Log

```
Exception in thread "(…)" java.lang.LinkageError: JAXB 2.0 API is being loaded from the
bootstrap classloader, but this RI (from jar:file:
(…)!/com/sun/xml/bind/v2/model/impl/ModelBuilder.class) needs 2.1 API. Use the endorsed
directory mechanism to place jaxb-api.jar in the bootstrap classloader. (See
http://java.sun.com/j2se/1.5.0/docs/guide/standards/)
   at com.sun.xml.bind.v2.model.impl.ModelBuilder.<clinit>(ModelBuilder.java:173)
   at com.sun.xml.bind.v2.runtime.JAXBContextImpl.getTypeInfoSet(JAXBContextImpl.java:422)
   at com.sun.xml.bind.v2.runtime.JAXBContextImpl.<init>(JAXBContextImpl.java:286)
   at com.sun.xml.bind.v2.ContextFactory.createContext(ContextFactory.java:139)
   at com.sun.xml.bind.v2.ContextFactory.createContext(ContextFactory.java:117)
   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
   at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
   at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
   at java.lang.reflect.Method.invoke(Unknown Source)
   at javax.xml.bind.ContextFinder.newInstance(Unknown Source)
   at javax.xml.bind.ContextFinder.find(Unknown Source)
   at javax.xml.bind.JAXBContext.newInstance(Unknown Source)
   at javax.xml.bind.JAXBContext.newInstance(Unknown Source)
(…)
```

- Ursache
  In der gestarteten JVM ist eine inkompatible Version von jaxb geladen.
- Lösung
  Installieren Sie eine aktuelle Sun-JVM oder passen Sie die Konfiguration wie in Kapitel 4.1.3 beschrieben an.

10. Technische Daten

Unterstützte Formate (Auszug):

- IBM AFP und MO:DCA-Familie
  - PTOCA
  - PTOCA
  - IOCA
  - GOCA
- HTML und E-Mails
- TIFF, JPEG, PDF
- Archiv-Formate
  - ZIP
  - GZIP
  - RAR
  - TAR
- Plaintext, XML, XSL:FO

Erforderliche Voraussetzungen:

Clientseitig:

- jadice Client Erweiterungen für Serverkommunikation
- JRE 1.5 oder neuer (in Ausnahmefällen JRE 1.4)
- Einsatz als Applet, Applikation oder eingebettet

Serverseitig:

- JRE 1.5 oder neuer
- Hauptspeicher: ab 2 GB, empfohlen 8 GB
- Prozessor: Pentium4 oder neuer, empfohlen 3 GHz oder schneller, Dualprozessor
- Festplattenspeicher: 80 MB Software, 2 GB Cache

Kommunikation zwischen Server und Clientkomponenten

- Java Messaging Services (JMS)
- Multiserverbetrieb und Load Balancing wird unterstützt

Vorhandene Konvertierungswerkzeuge

- Office-Plugins für TIFF-Export
- Office-Plugins für PDF-Export
- jadice shaper

11. Versionshistorie

Version	Datum	Autor	Änderungen (Syntax: * geändert, + neu, – entfallen)
4.1.x	16.03.09	B. Geißelmeier	Komplett überarbeitet
4.2.0.0	02.07.09	B. Geißelmeier	+ Migration von 4.1.x zu 4.2.0.0 * Beispiele in Kapitel 5.3 an geänderte API angepasst + Beschreibung Webservice-Schnittstelle
4.2.1.0	30.07.09	B. Geißelmeier	* Beschreibung WS-Schnittstelle + Kapitel „Generierung von Webservice-Clients" + Kapitel „Konfiguration MS Office" + Kapitel „Konfiguration Ghostscript"
4.2.1.1	31.08.09	B. Geißelmeier	* Beschreibung WS-Schnittstelle + Kapitel „Jobdefinition innerhalb der SOAP-Nachricht" * Korrektur Beschreibung ExternalProcessCallNode
4.2.1.3	27.11.09	B. Geißelmeier	* Konfiguration MS Office + Konfiguration MS Outlook * Kapitel „Konvertierung von E-Mails nach PDF" + Kapitel „Troubleshooting" + Migration nach Version 4.2.1.3, Kapitel 8.2.
4.2.1.5	22.01.10	B. Geißelmeier	+ Kapitel „Monitoring" * Kapitel „Konfiguration des eingebetteten Messagebrokers" * Kapitel „Konfiguration MS Office" * Kapitel „Konfiguration Wrapper" + Migration nach Version 4.2.1.5, Kapitel 8.3.