PROKOM Web Service Schnittstelle

Versionsgeschichte

Aktuelle Version: 0.1	Datum: 01.03.2011

Versions nummer	Versions- datum	Zusammenfassung der Änderungen	Änderungen markiert
0.1	01.03.11	Erste Version	Nein

Inhaltsverzeichnis

1 Einleitung 4
2 Technische Bestandteile 5
2.1 PROKOM Object 6
2.2 PROKOM WS Base 10
2.3 PROKOM WS 13
3 Abschlussbemerkung 15

1 Einleitung

Die DeTeMedien liefert ab 2011 Eintragsdaten (Telefon-, Branchenbücher) im sogenannten PROKOM-Format, ein XML-Format, das mit einem XML-Schema (XSD) definiert ist. U.U. kann davon ausgegangen werden, dass Eintragsdaten darüber hinaus zukünftig in diesem Format ausgetauscht werden. Eine umfassende systemtechnische Unterstützung ist damit nutzbringend.

Das XML-Format bietet auch die Gelegenheit, technologisch das Thema Datenaustausch und Kommunikation neu zu bewerten und, insoweit noch nicht geschehen, die Unternehmens-IT Richtung Prozess- und Serviceorientierung weiter zu entwickeln. Die Aufgabenstellung PROKOM-Eintragsdaten kann dafür dann einen exemplarischen Kristallisationskern bilden.

Um diese Daten zu verarbeiten, sie z.B. in eine Datenbank zu importieren, können sie mittels Java-Technologien geparst, in Objekte unmarshalled und zur weiteren Bearbeitung verwendet werden. Das ist eine Möglichkeit, die gleichzeitig erlaubt, bezüglich systemtechnischer Entwicklung Web Services zu etablieren und damit zeitgemäße Standardschnittstellen zu schaffen. Der Begriff „Standard“ ist hierbei berechtigt, weil es sich bei PROKOM um ein unternehmensübergreifendes Format handelt, also mehr als ein Unternehmen dieses beherrschen müssen. (Wobei DeTeMedien nicht als eine Standardisierungs-Instanz verstanden werden kann)

Seitens NeRTHUS (in Kooperation mit der CCDM GmbH) existiert eine Rahmenimplementierung, d.h. alle Aspekte der Formatbearbeitung, vom Parsen, Marshalling, Unmarshalling, Databinding, Web Service-Client bis Web Service-Server, sind ausimplementiert, es fehlt lediglich die unternehmensspezifische Verarbeitung, z.B. das Abspeichern in Transfer-Tabellen oder aber die vollständige Verarbeitung in einem Unternehmens-Datenmodell.

Auf der Grundlage der Quellen können auch leicht weitere, auf andere XML-Formate basierende Schnittstellen effektiv in Java geschaffen werden.

Zur Veranschaulichung wird die Implementierung mit Quellen als ein Verzeichnis „prokom“ geliefert, das vorzugsweise auf das Laufwerk C: kopiert wird. Dann stimmen alle Pfade und es ist schnell und unkompliziert möglich, das System zu präsentieren und die Intention darzustellen.

2 Technische Bestandteile

Die Implementierung basiert auf folgende Bestandteile:

- AXIS2 Web Service (Open Source)
- PROKOM Object (Prokom-Objekte wie Eintrag, Kommnr, Adresse, …)
- PROKOM WS Base (Basis-Klassen Web Service)
- PROKOM WS (konkreter Web Service Prokom-WS)
    - WSDL

Zusätzlich wird noch ein Tomcat mit installiertem AXIS2 und prokom-ws geliefert, Port 8080, um die Implementierung zu testen und erste Erfahrungen zu gewinnen.
Um den Tomcat zu starten (apache-tomcat-6.0.29/bin/startup) muss zuvor eine Umgebungsvariable JAVA_HOME auf z.B. C:\Programme\java\jdkxxx gesetzt werden (alternativ funktioniert auch ein „set" im setclasspath.bat).

Wenn Tomcat erfolgreich gestartet ist, meldet er sich im Browser bei localhost:8080 und mit localhost:8080/prokom-ws/services/listServices sieht man den laufenden Web Service ProkomService, der nun auf Aufrufe wartet (siehe Kapitel PROKOM WS Base).

The Apache Software Foundation
http://www.apache.org/

Available services

ProkomService

Service EPR : http://localhost:8080/prokom-ws/services/ProkomService

Service Description : ProkomService

Service Status : Active
Available Operations

- begin
- handleLieferung
- mirrorEintrag
- end
- mirrorLieferung
- handleEintrag

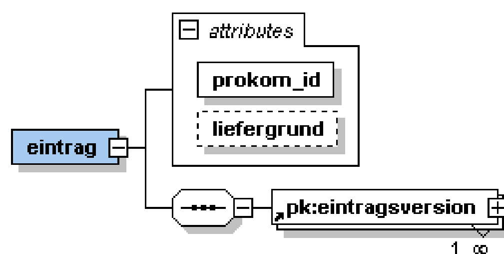
2.1 PROKOM Object

In diesem Source-Modul sind die Klassen implementiert, die den XML-Objekten von PROKOM entsprechen. Das Modul basiert auf dem NeXML-Objectframework, wo die wesentliche Grundfunktionalität programmiert ist, die hier für PROKOM angewendet wird.

Wesentlich sind die Klassen, die das Interface NeXMLInterface implementieren:

```
package com.nerthus.prokom.object;

import java.util.Iterator;
import java.util.Vector;

import com.nerthus.xml.NEXMLContext;
import com.nerthus.xml.NEXMLDataMarshaller;
import com.nerthus.xml.NEXMLInterface;
import com.nerthus.xml.NEXMLNode;

import com.nerthus.prokom.object.type.EnumLiefergrund;

/**
 * BusinessObject for <prokom:eintrag>
 */
public class Eintrag implements NEXMLInterface {

      private String tag = "eintrag";
      protected Long prokom_id;
      protected EnumLiefergrund liefergrund = null;
      protected Vector<Eintragsversion> eintragsversion =
                                    new Vector<Eintragsversion>();
```

Das PROKOM-Objekt „eintrag“ ist im Schema folgendermaßen definiert:



Mittels den Methoden writeTo() und initWith() erfolgt das Schreiben in oder das Lesen aus XML:

```
public NEXMLNode writeTo(NEXMLNode parent, NEXMLContext context,
                  boolean depth) {

      NEXMLNode node = this.createNode(context, parent, namespace, tag);

      node.addAttribute("prokom_id", prokom_id);
      node.addAttribute("liefergrund", liefergrund);
      node.addObjects(eintragsversion, context, depth);

      return node;
}

public void initWith(NEXMLDataMarshaller dm, NEXMLNode node,
                              NEXMLContext context) {

      String nodeName = node.getName();
      if ((node == null) || (node.getNodeType() != NEXMLNode.ELEMENT)
                  || ((!nodeName.equals(tag)
                  && !nodeName.startsWith("eintrag_")))) {
            return;
      }
      this.init();

      if (nodeName.startsWith("eintrag_")) {
            operation = nodeName.substring(8);
      }

      prokom_id = node.getAttributeAsLong("prokom_id");

      …
}
```

Wichtig ist die Zuordnung von XML-Element (name, tag) und Klasse für alle einzeln für sich behandelbare Klassen (in einer PROKOM-XML-Datei bräuchte man nur <prokom:lieferung> zu kennen, um das Dokument in das Objekt Lieferung (mit allen Unterobjekten, komplettes, der Datei entsprechendes Objektmodell) zu überführen. Aber aus Speicherverbrauchs- und Performancegründen ist es effektiver, mittels ObjectParser die Einträge einzeln zu bearbeiten (dabei wird <prokom:lieferung> ignoriert).

Diese Zuordnung erfolgt im DataMarshaller:

```
package com.nerthus.prokom;

public class DataMarshaller extends NEXMLDataMarshaller {

	/** Creates a new instance of DataMarshaller */
	public DataMarshaller() {
		super();
		businessClasses.put("lieferung",
	"com.nerthus.prokom.object.Lieferung");
		businessClasses.put("eintrag",
	"com.nerthus.prokom.object.Eintrag");
		businessClasses.put("eintrag_anlegen",
	"com.nerthus.prokom.object.Eintrag");
		businessClasses.put("eintrag_pflegen",
	"com.nerthus.prokom.object.Eintrag");
		businessClasses.put("eintrag_loeschen",
	"com.nerthus.prokom.object.Eintrag");
	}
```

Das ist im Prinzip alles, was für das XML-Binding mittels NeXML Objectframework zu tun ist, um ein eigenes Objectframework, in dem Fall für PROKOM, zu erstellen. Auf dem Open Source-Markt gibt es noch andere, wie z.B. Castor, JAXB usw.. Diese generieren aber nach meiner Erfahrung ein komplexes Objektmodell, das schwer zu überschauen und wo die Objektmodellierung nur schwer zu kontrollieren ist. Beim NeXML ist das Databinding zum führenden WS-Framework AXIS2 gleich dabei, d.h. in den Web Service wird mit diesen Objekten gearbeitet.

Die Generierung der Klassen erfolgt normalerweise mittels Stylesheet XSD2NeXML.xsl, im Fall PROKOM dauerte die Implementierung per Hand 0,5 PT.

Die Verwendung des PROKOM-Objectframeworks wird in den Klassen:

```
com.nerthus.prokom.sax.Main
com.nerthus.prokom.parser.Main
```

demonstriert. Der Unterschied besteht in dem Umfang des Parsens, bei SAX ist es möglich, nur sequentiell Objekte aus dem XML zu behandeln (Speicherverbrauch, Performance) und beschreibt die zu bevorzugende Methode (die auch für den WS-Client verwendet wurde – Abschnitt PROKOM WS Base). Dazu kommt der ObjectParser zum Einsatz, der dem ObjectHandler mittels der Methode mashalled() informiert, welches Objekt er im XML erkannt und marshalled, damit als Java-Objekt zur Verfügung stellt, hat.

```
objectParser = new NEXMLObjectParser(datamarshaller, this, "lieferung");
objectParser.parseFile(argv[0]);
```

Unter <prokom:lieferung> (Root-Element, XML-Dokument) werden Objekte gesucht, die der DataMarshaller am Tag erkennt.

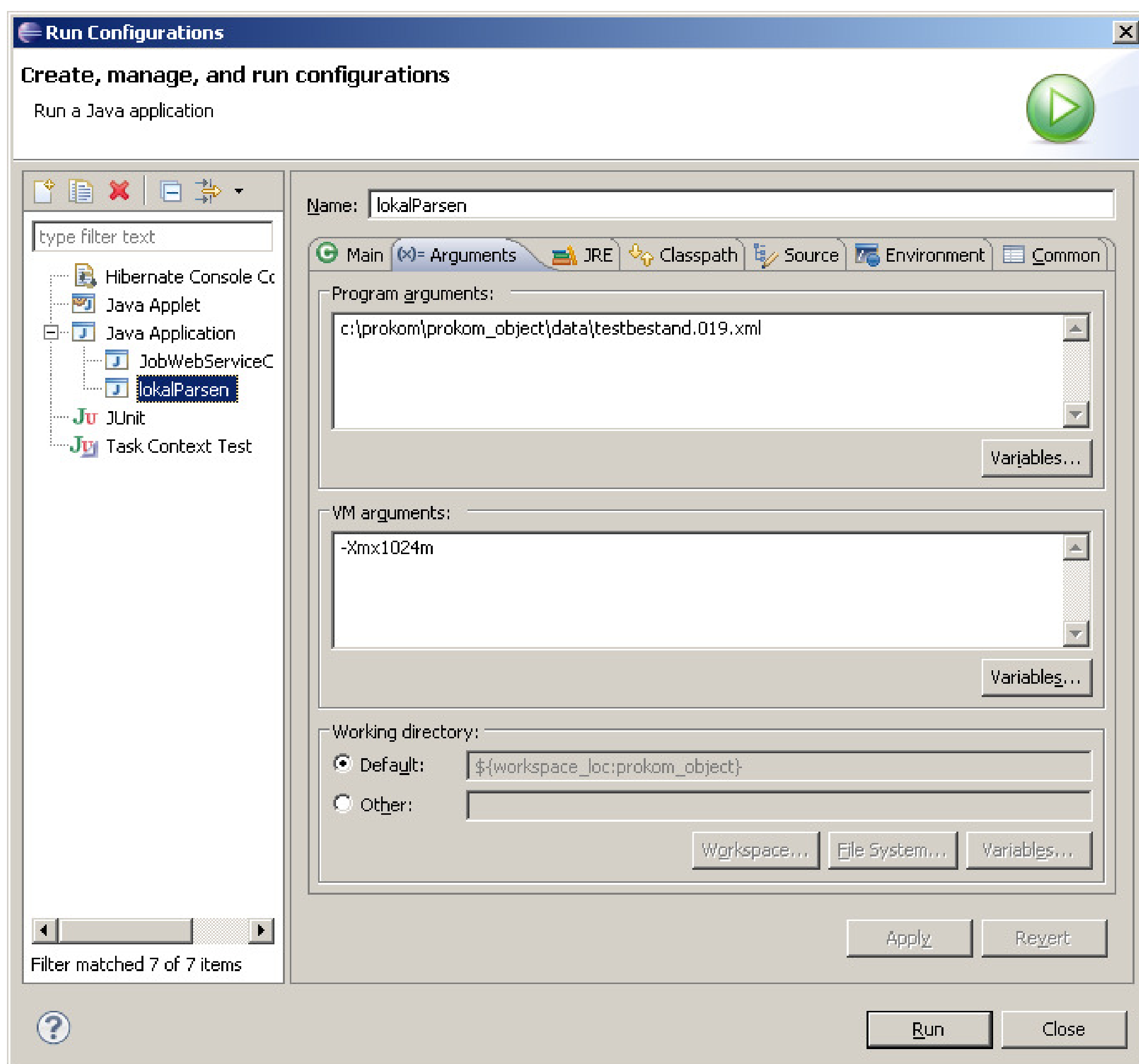
Der Parameter “this” meint das Main-Objekt

```java
public class Main implements NEXMLObjectHandler {
```

das als ObjectHandler fungiert und die Methode unmarshalled() implementiert hat.

```java
	public void unmarshalled(NEXMLInterface obj, NEXMLContext context) {
		if (obj instanceof Eintrag) {
			objectCount++;
			System.out.println(object.getXPath(true));
		} else if (object instanceof Lieferung) {
			objectCount++;
			System.out.println(object.getXPath(true));
		}
	}
```

Für einen Test ist in dem zugehörigen Eclipse-Projekt ein RUN konfiguriert.



Mit Run oder Debug (Breakpoints an interessierenden Stellen) kann man es ausprobieren und z.B. einen Eindruck zur Geschwindigkeit gewinnen.

2.2 PROKOM WS Base

In diesem Modul sind die Basis-Klassen für Web Services in professioneller Umgebung implementiert. Näheres entnehme man der Literatur zu AXIS2, wesentlich sind Parameter in unterschiedlichen Kontexten (Service, ServiceGroup, Configuration), das Session- und Lebenszyklus-Management (z.B. für die Implementierung von Protokollierungen zu Web Service-Sessions, Etablierung von DB-Connections etc.).
Für die PROKOM WS wurde hier ein WebService-Client implementiert

```
com.nerthus.ws.client.batch.JobWebServiceClient
```

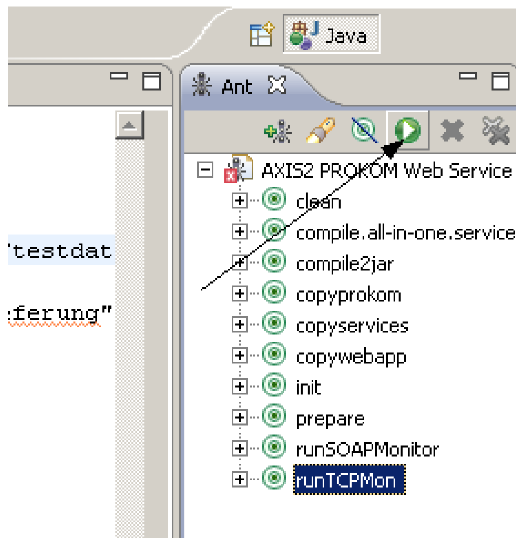
Dieser dient für den Aufruf aus der Kommandozeile oder aber aus einem Scheduler als Batch-Job. Damit werden Dateien aus dem Dateisystem geparst (oder aus einer Mail-Inbox) und die Daten an den Web Service geschickt und ist konfigurierbar.
Die Konfiguration für die Jobs ist eine XML-Datei (imex/conf)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<jobconfig session="true" timeout="60000">
  <!-- config of import -->
  <import name="Eintragsdaten" desc="Import/Update Prokom">
    <file-select format="NULL" source-dir="C:\prokom\ws_base\imex\examples"
      filename-mask="testdaten_rb.xml" source-mask="*"/>
    <jobs>
      <job
        endpoint="http://localhost:8081/prokom-ws/services/ProkomService"
        object="lieferung"
        operation="handleLieferung"
        return="properties"
        split="10"/>
    </jobs>
    <post-processing old-dir="/tmp/old" err-dir="/tmp/err"/>
  </import>
</jobconfig>
```

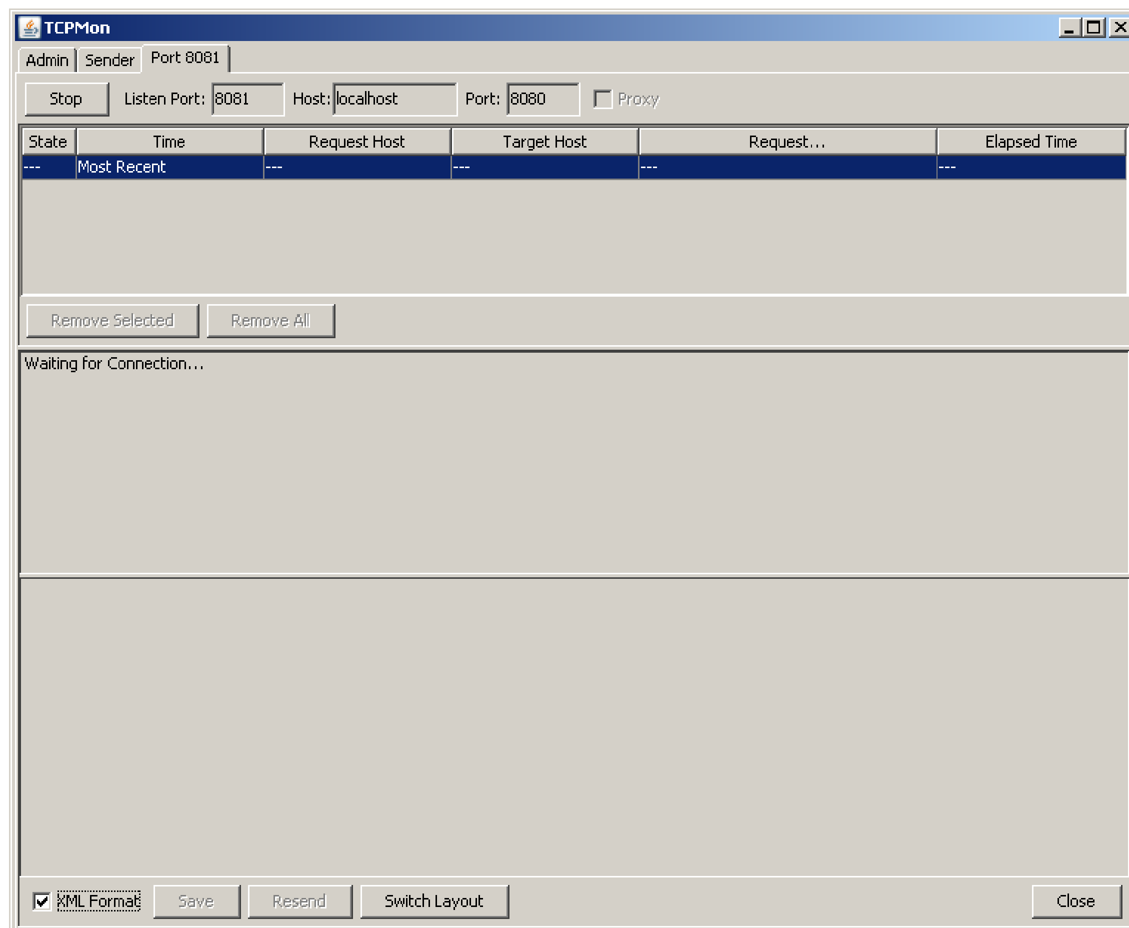
wo angegeben wird, wo die Dateien zu finden sind (source-dir) oder welcher Web Service (endpoint und operation) aufgerufen wird. Die vorhandene import_prokom.xml ist beispielhaft und dient dem Test der vorhandenen Implementierung. Die Struktur ist mit ein wenig Dokumentation dem XML-Schema imex/imex_conf.xsd entnehmbar.

Um das Ganze zu testen, ist zuerst der Tomcat zu starten (JAVA_HOME setzen, startup in bin aufrufen – siehe Einleitung). Wie man oben an der Konfiguration erkennt, sendet der Client an localhost:8081! Das liegt daran, dass wir zur Veranschaulichung der Funktionsweise den TCPMonitor starten, der auf Port 8081 den WS-Aufruf empfängt, an Port 8080 weiter leitet und uns damit den Request und die Response sichtbar macht.

Der TCPMonitor wird im Eclipse via Ant gestartet:

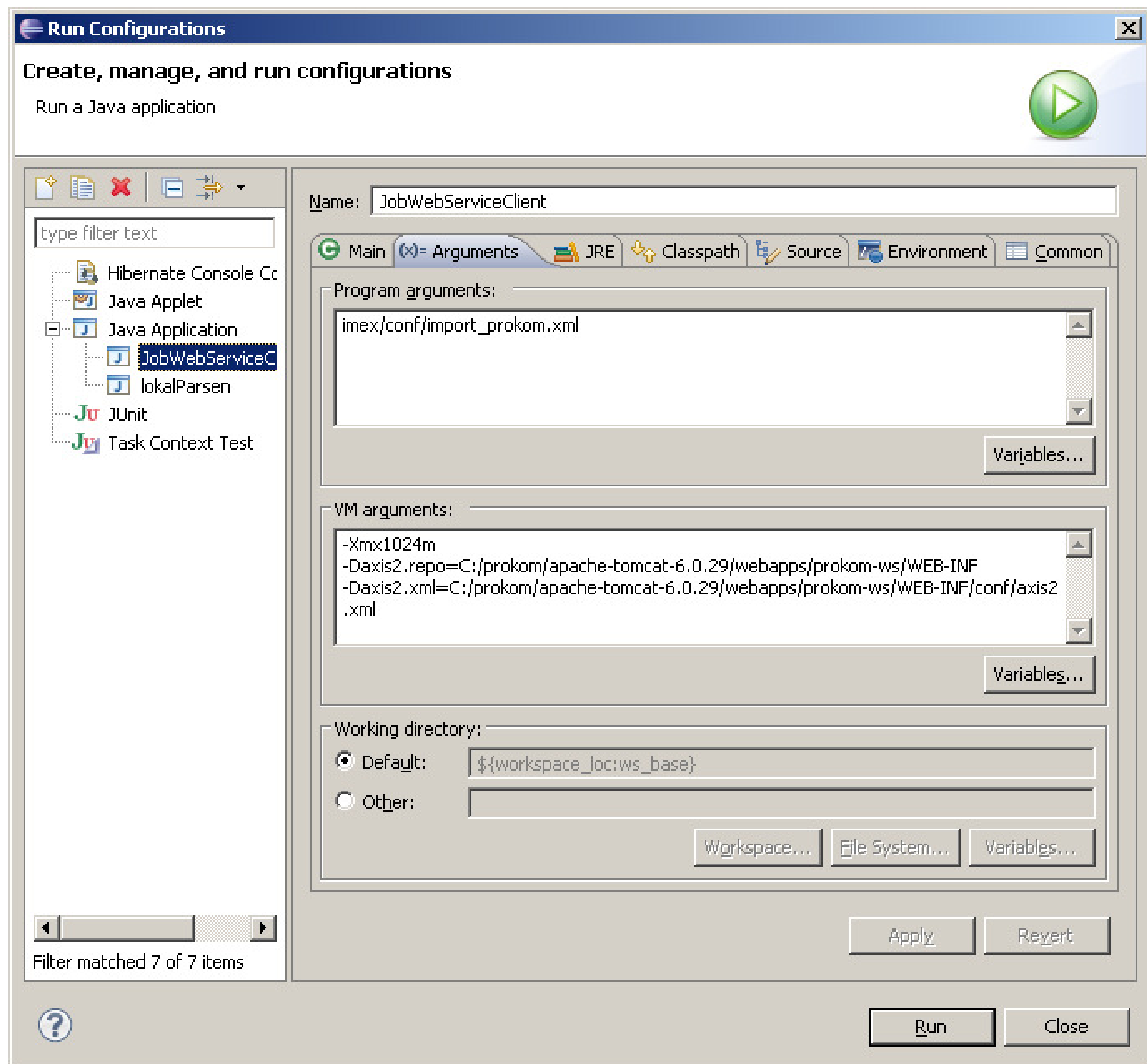


und meldet sich:



(den Haken bei *XML-Format* bitte setzen!)

Im Eclipse ist auch ein RUN für den JobWebServiceClient konfiguriert:

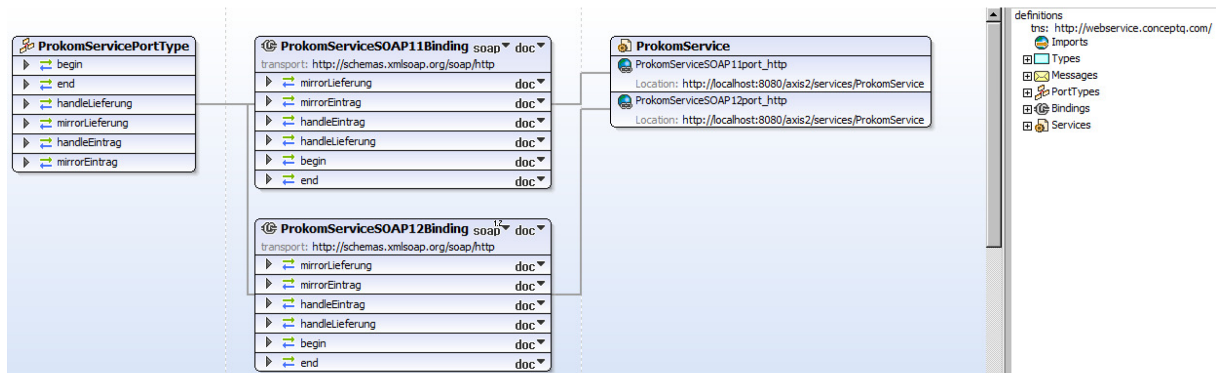


Damit wird der Client gestartet und mit dieser Konfiguration kann auch das Debugging des Clients stattfinden.

2.3 PROKOM WS

In dem Modul ist die Web Service-Klasse implementiert, die mittels service.xml im AXIS2 konfiguriert wird. Alle Bestandteile befinden sich bereits compiliert in dem AXIS2-Server (siehe Literatur und apache-tomcat-6.0.29\webapps\prokom-ws\WEB-INF\services\prokom-ws).

Wesentlich ist die WSDL, die eineindeutig den WebService beschreibt.
(…/WEB-INF/ProkomWebservice.wsdl)



Das Databinding, also das Mappen von XML-Fragment auf zugehörige Klasse, wird bei den Web Services durch eine Ressourcen-Datei konfiguriert, da das Web-Service-Framework unabhängig von dem jeweils implementierten Objectframework ist, d.h. der DataMarshaller des Prokom-Objectframeworks serverseitig keine Verwendung findet. Die Datei befindet sich in …\webapps\prokom-ws\WEB-INF\ und heißt nexml_databinding.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
   <entry key="lieferung">com.nerthus.prokom.object.Lieferung</entry>
   <entry key="eintrag">com.nerthus.prokom.object.Eintrag</entry>
   <entry key="eintrag_anlegen">com.nerthus.prokom.object.Eintrag</entry>
   <entry key="eintrag_pflegen">com.nerthus.prokom.object.Eintrag</entry>
   <entry key="eintrag_loeschen">com.nerthus.prokom.object.Eintrag</entry>
</properties>
```

In der Klasse `com.nerthus.service.prokom.ProkomService` werden die Web Service-Operationen als Methoden implementiert, z.B.

```
public Properties handleLieferung(Lieferung lieferung, Properties props)
                                                        throws AxisFault {

      // HIER: unternehmensspezifische Verarbeitung
      return result;
}
```

In denen ist die Verarbeitung der erhaltenen Daten unternehmensspezifisch zu implementieren. Im Beispiel wird eine Ursprungsdatei in Lieferungen zu je 10 Einträgen gesplittet und an den Web Service mit der Operation handleLieferung() gesendet. Hier erfolgt dann die Verarbeitung z.B. mit einem Spring-Hibernate-Backend Richtung Datenbank. Die Beispielimplementierung zeigt, wie die Daten „angefasst“ werden und wie ein mögliches Result unter Verwendung von Nachrichten <nexml:messages><nexml:msg> aussehen könnte.


```
<properties name="result">
   <property name="service" value="ProkomService" />
   <property name="operation" value="handleLieferung" />
   <property name="datenbestand" value="2010-10-05T00:00:00+02:00" type="xs:datetime" />
   <property name="bestand" value="10" type="xs:integer" />
   <property name="anlegen" value="0" type="xs:integer" />
   <property name="loeschen" value="0" type="xs:integer" />
   <property name="pflegen" value="0" type="xs:integer" />
   <property name="messages">
      <messages>
         <msg object="*/eintrag[@prokom_id='203041060']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203041160']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203043260']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203046060']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203046660']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203051360']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203052060']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203053860']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203055260']" level="I">Bestand OK</msg>
         <msg object="*/eintrag[@prokom_id='203056760']" level="I">Bestand OK</msg>
      </messages>
   </property>
   <property name="state" value="true" type="xs:boolean" />
</properties>
```


Die Verarbeitungszeit eines einzelnen Eintrags, aus einer Datei geparst, an den Prokom-WS gesendet und von diesem in der Verarbeitung bestätigt, d.h. Response gesendet und ausgewertet, beträgt hier auf dem Testsystem ca. 1,2 ms.

```
================================================================
= WebServiceClient: Ende Import!                               =
= Bearbeitungszeit:    354698 ms                               =
= Anzahl Objekte:      303514                                  =
================================================================
```

Unter realen Bedingungen ist noch die Verarbeitungszeit des Backends und ggf. Netzwerk verursachte Zeit hinzu zu rechnen. Aber die Zeit, die das Framework als Grundlast verursacht, ist sicher zu vernachlässigen.

3 Abschlussbemerkung

Dieses Dokument ist nur eine kurze Beschreibung, um erste Einblicke und vielleicht mittels weiterer Implementierungen Erfahrung zu gewinnen. Für den produktiven Einsatz sind neben der unternehmensspezifischen Erweiterung sicherlich ein paar Aspekte genauer zu erläutern und seitens der Mitarbeiter in Besitz zu nehmen.

Der TCPMonitor z.B. streikt nach > 100000 Einträgen, der Service arbeitet am besten im Scope „request" (service.xml) und Sessions allgemein (config.xml) sind für die Verarbeitung von Massendaten auch nicht sinnvoll (eher für verschiedene Operationen, die zu einem gemeinsamen Vorgang gehören, wie Rechnungslegung, Rechnungsversand, Buchung …)

Deshalb der übliche Satz: Für Rückfragen stehe ich gerne zur Verfügung, auch für eine Unterstützung bei der Umsetzung als produktive Schnittstelle oder gar bei der Etablierung von Unternehmens-WS.

Ronald Bohlig