

J2ME Web Services (JSR-172) und J2EE Client Provisioning

1. Inhalt

J2ME Web Services (JSR-172) und J2EE Client Provisioning

1. Inhalt

2. J2ME Web Services (JSR-172)

2.1. Einleitung

2.1.1. Definition

2.1.2. Motivation

2.2. Architektur

2.2.1. SOAP

2.2.2. WSDL

2.2.3. UDDI

2.3. Entwicklung von mobilen Web Services

2.4. J2ME Web Services (JSR-172)

2.4.1. Architektur

2.4.2. JAX-RPC Subset

2.4.3. JAXP Subset

2.5. Herausforderungen

2.5.1. Content Awareness

2.5.2. Sessionmanagement

3. J2EE Client Provisioning

3.1. Provisioning Server

3.2. Architektur

3.2.1. Adapters.xml

3.2.2. Devices.xml

3.2.3. Matchers.xml

3.2.4. Provisioning Archiv (PAR)

3.3. Provisioning Client

3.3.1. Discovery

3.3.1. Delivery

3.3.1. Stocking

3.4. Anwendung und Ausblick

3.4.1. Anwendungsbeispiele

3.4.1. Ausblick

4. Quellen

4.1. J2ME Web Services (JSR-172)

4.2. J2EE Client Provisioning

2. J2ME Web Services (JSR-172)

2.1. Einleitung

Web Services erfreuen sich in den vergangenen Jahren immer höherer Beliebtheit, um nicht gar in manchen Zweigen von Boom zu reden. Sowohl in der Industrie als auch bei immer mehr Hochschulen trifft diese Technologie auf großen Anklang. Sie berührt viele Disziplinen der Informatik, wie Verteilte Systeme, Programmiersprachen, Software Engineering und Informationssysteme. Einer der wesentlichen Gründe des großen Interesses an dieser Technologie hat mit dem Zauberwort Web zu tun. Sobald dieses Wort irgendwo auftaucht, werden Benutzer wie auch Entwickler hellhörig. Tatsächlich haben Web Services nur nebenbei mit dem Web, wie die meisten es kennen, zu tun. Beim Web geht es um die Kommunikation von Menschen,

bei Web Services kommunizieren mehr die Maschinen.

Ein anderer Boomfaktor der letzten Jahre ist zweifelsohne der mobile Sektor. Keine Branche hat eine so hohe Umsatzsteigerung erzielt. Ende 2002 gab es über 59 Millionen Handy-Nutzer in Deutschland. Die Zahl der Mobiltelefone hat sich damit bei uns in den letzten zehn Jahren mehr als verundertfacht. Die dritte Mobilfunkgeneration UMTS (Universal Mobile Telecommunications System) ermöglicht seit diesem Jahr umfassende Multimediaanwendungen via Handy. Die bis zu 200-mal höheren Übertragungsgeschwindigkeiten erlauben es, über die gewohnte Nutzung des Telefons hinaus auch Online-Dienste abzurufen, umfangreiche Datenpakete zu übermitteln und Musik und bewegte Bilder per Mobiltelefon auszutauschen.

Mit sinkenden Kosten für das mobile Internet liegt eine Verbindung von Web Services und mobilen Geräten nah. Sun hat dazu eine Spezifikation JSR-172 herausgebracht, um Applikationen für ein mobiles Gerät zu schreiben, die Web Services nutzen.

2.1.1. Definition

Zunächst einmal sollte geklärt werden was Web Services eigentlich sind. Es gibt keine anerkannte Definition des Begriffs, man kann aber in etwa sagen, dass Web Services Internet Applikationen sind, die irgendwo im Internet niedergelegt sind und die durch Standard Internet Protokolle wie HTTP und SMTP von einer Client Software aufgerufen werden können. Web Services können von der Handhabung so leicht wie das Aufrufen einer Internetseite sein, aber auch komplexe Strukturen sind möglich.

2.1.2. Motivation

Ausschlaggebend für die Nutzung von Web Services ist zunächst einmal die alte Vision vom Markt der Komponenten, bzw. jetzt Web Services. Unabhängige Dienstleister bieten dabei die Web Services an und müssen diese nicht einmal lokal lagern, sondern können dafür die Rechner eines Application-Service-Providers (ASP) nutzen.

Als nächstes ist natürlich das Web als Schnittstelle optimal und vor allem verhältnismäßig einfach zu nutzen. Bisher wurden Informationen zumeist über die Eingabe in den Web-Browser weitergegeben. Mit Web Services ist hier eine Automation möglich.

Des Weiteren zeichnen sich Web Services durch einheitliche und verbreitete Protokollen und Schnittstellen aus. Es wird zum Großen Teil HTTP und XML verwendet.

2.2. Architektur

Die wesentlichen Aufgaben für einen reibungsfreien Ablauf von Web Services sind die Beschreibung von Schnittstellen und der Nachrichtenaustausch. Hierzu wurden SOAP, WSDL und UDDI als drei Kerntechnologien entwickelt. Sie basieren alle auf XML und nutzen die Internetprotokolle HTTP und SMTP.

2.2.1. SOAP

SOAP (Simple Object Access Protocol) ist das grundlegende Transportmittel für Web Services. Sein Aufbau in XML Format macht es dem Benutzer einfach diese nachrichtenbasierte Übertragung auf Fehler zu durchsuchen. Die SOAP-Nachrichten werden über ein Trägerprotokoll, wie z.B. HTTP, transportiert. Eine SOAP-Nachricht besteht aus einem Hauptteil und optionalen Attachments, die jedoch im mobilen Bereich nicht unterstützt werden. Der Hauptteil wird durch einen Umschlag (ENVELOPE) charakterisiert. Dieser unterteilt sich wiederum in Kopf (HEADER), der Anwendungsdaten wie XML-Namespaces enthält, und den Rumpf (BODY) für den wichtigen Teil der Nachricht und dessen Verarbeitung.

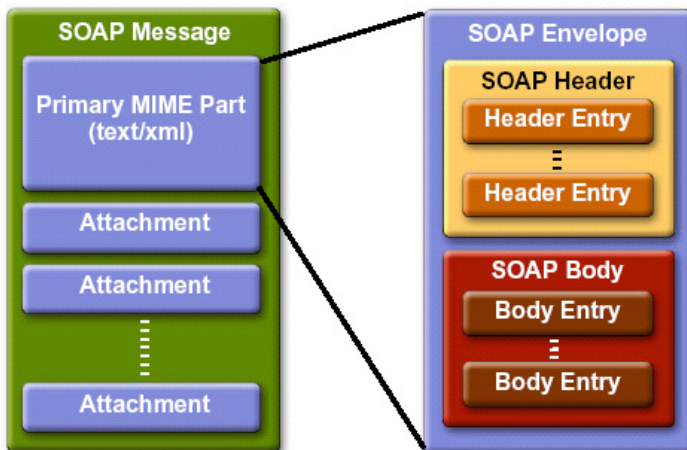


Abbildung 1: SOAP Architektur [Developing Real World Web Services S.11]

Nachfolgend ist eine Anfrage an einen Server zur Übersetzung eines Textes zu sehen.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <getTranslation xmlns="http://localhost/translator/1.0/wsdl/TranslatorService"
      id="o0" SOAP-ENC:root="1">
      <String_1 xmlns="" xsi:type="xsd:string">inputLocale</String_1>
      <String_2 xmlns="" xsi:type="xsd:string">outputLocale</String_2>
      <String_3 xmlns="" xsi:type="xsd:string">inputText</String_3>
    </getTranslation>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

`getTranslation()`, mit drei Übergabeparametern, ist die aufzurufende Methode des Services zur Übersetzung. Wichtig für ein mobiles Gerät ist, dass es die Kodierung UTF-8 auch versteht, denn sonst kann man diese Nachricht nicht verschicken.

2.2.2. WSDL

Im vorhergehenden Teil war von SOAP als Transportmittel die Rede. Um von außerhalb zu verstehen wie die SOAP Nachricht versendet werden soll und was sie für Parameter hat, benötigt man eine Sprache, die ähnlich strukturiert ist und diese Beschreibung vornimmt.

Web Services Description Language (WSDL) wurde zu diesem Zweck und zu weiteren Beschreibungs-

möglichkeiten für Web Services in XML Format eingeführt. Das Wesentliche an einer solchen Beschreibung sind die Endpunkte (ENDPOINTS). Dabei legt WSDL genau fest wo der benötigte Service sich befindet. Zu beachten ist hierbei, dass alle WSDL Beschreibungen abstrakt sind, d.h. es werden keinerlei Funktionen oder Dienste bereitgestellt, nur beschrieben.

```
<definitions>
  <import>*
  <types>
    <schema></schema>*
  </types>
  <message>*
  <part></part>*
  </message>
  <PortType>*
    <operation>*
      <input></input>
      <output></output>
      <fault></fault>*
    </operation>
  </PortType>
  <binding>*
    <operation>*
      <input></input>
      <output></output>
    </operation>
  </binding>
  <service>*
    <port></port>*
  </service>
</definitions>
```

Hier ist der strukturierte Aufbau einer WSDL-Datei zu sehen. Die Sterne stehen wie auch bei regulären Ausdrücken für die mehrmalige Wiederholungsmöglichkeit. Unter <types> werden die verwendeten Daten- und Kommunikationstypen, wie SOAP oder String, beschrieben. In <message> steht wie der Datenaustausch laufen soll und wie viele Parameter verwendet werden. Alles was unter <PortType> zu finden ist, stellt eine Beschreibung der verfügbaren Funktionen des Services dar. <binding> greift diese Funktionen auf und beschreibt dann, wie konkret mit ihnen umzugehen ist, z.B. dass an einer gewissen Stelle SOAP verwendet wird. Der Endpunkt wird dann durch <service> aufgeführt. Die Adresse des Endpunktes könnte z.B. `http://localhost:8080/translator/1.0/wsd/TranslatorService` sein. WSDL Dokumente schreibt man in der Regel nicht selbst. Die Arbeit übernehmen Hilfsprogramme aus dem Java Web Service Development Kit oder eine Entwicklungsumgebung. Oft ist es aber sehr hilfreich solche Dokumente lesen zu können.

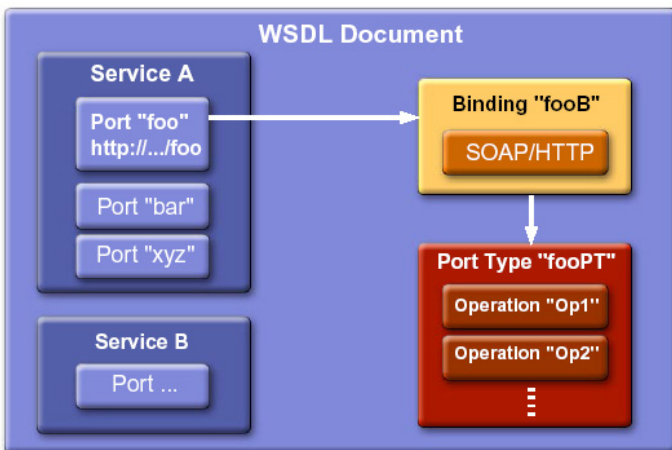


Abbildung 2: WSDL Architektur [Developing Real World Web Services, S.12]

2.2.3. UDDI

Das Universal Description, Discovery & Integration (UDDI) Projekt ist ursprünglich aus einer Zusammenarbeit von Ariba, IBM und Microsoft hervorgegangen. Ziel des UDDI Projektes ist es, die Zusammenarbeit von Firmen über Ihre Internet-Dienste zu beschleunigen, aber auch das Angebot von Internet-Diensten überhaupt erst zu fördern. Dies soll durch eine Standardisierung von Beschreibung, Auffindung und Integration von Geschäften/Unternehmen für das Internet erreicht werden.

Die Vision war eine Registrierdatenbank zur Verfügung zu stellen. Diese Datenbank enthält programmatische Beschreibungen von Internet-Diensten und programmatische Beschreibungen von Unternehmen und den Diensten, die sie unterstützen. Diese Registrierdatenbank ist öffentlich im Internet zugänglich.

Einen Eintrag in einer UDDI-Registry kann man auf zweierlei Arten erzeugen:

- entweder elektronisch mit speziellen "Publish-API"-URLs (z.B. "https://www-3.ibm.com/services/uddi/testregistry/protect/publishapi") durch bestimmte Funktionsaufrufe (z.B. "save_business()", "save_service()", "save_tModel()")
- oder interaktiv im Webbrowser.

Einen Eintrag in einer UDDI-Registry kann man auf zweierlei Arten suchen:

- entweder elektronisch mit speziellen "Inquiry-API"-URLs (z.B. "https://www-3.ibm.com/services/uddi/testregistry/inquiryapi") durch bestimmte Funktionsaufrufe (z.B. "find_business()", "find_service()", "find_tModel()")
- oder interaktiv im Webbrowser.

2.3. Entwicklung von mobilen Web Services

Bei der Entwicklung von Applikation auf mobilen Geräten steht jeder vor der Herausforderung mit einem Gerät arbeiten zu müssen, das einen schwachen CPU und geringe Speicher-Ressourcen bereitstellt.

Außerdem sollte man nicht vergessen, dass sich der Benutzer in Bewegung befindet und somit ein Session-Management notwendig sein wird, das gegebenenfalls Informationen über den Client speichert, damit dieser schnell und unkompliziert wieder den Dienst nutzen kann.

Wichtig ist auch noch der einfache Umgang für den Benutzer. Oft sind es Geschäftsleute, die Web Services nutzen und die mehr darauf bedacht sind, die Programme auch wirklich zu nutzen, anstatt sich mit lästigen Installationen und umständlichen Aktualisierungen herumschlagen.

Des Weiteren kann die bisherige Erfahrung mit Web Services auf dem Desktop Bereich genutzt werden um effizientere Applikationen zu schreiben. Die Herausforderung liegt dabei in der Verkleinerung der Desktop Anwendungen, so dass sie die festgelegten Höchstgrenzen der Dateigröße bei mobilen Geräten nicht überschreiten und dennoch richtig laufen.

Bei Web Services befindet sich der Server zur Übertragung von Informationen auf einem Desktop PC oder Server. Auf mobilen Geräten ist bis heute nur eine Client Anwendung möglich.

2.4. J2ME Web Services (JSR-172)

Sun hat zur Erweiterung der Java 2 Micro Edition Plattform die J2ME Web Services API (JSR-172) herausgebracht um Web Services auch bei mobilen Applikationen zu unterstützen. Die API besteht aus zwei zusätzlichen Packages (optional packages) die durch JAX-RPC eine Ansteuerung von Web Services erlauben und mit JAXP ein XML Parsing System auf mobilen Geräten anbieten.

Um mit der API arbeiten zu können benötigt man ein mobiles Gerät, das entweder Connected Device Configuration (CDC), zumeist PDAs, oder Connected Limited Device Configuration (CLDC 1.0 oder CLDC 1.1), z.B. Handys, unterstützt. JAX-RPC und JAXP sind auch nur mit eingeschränkter Funktionalität, so genannte Subsets, in der Spezifikation definiert. Ziel ist es, grundlegende Unterstützung für Web Services Einbindung und XML Parsing zu geben, so dass Entwickler nicht extra solche Funktionen für jede Applikation schreiben müssen.

2.4.1. Architektur

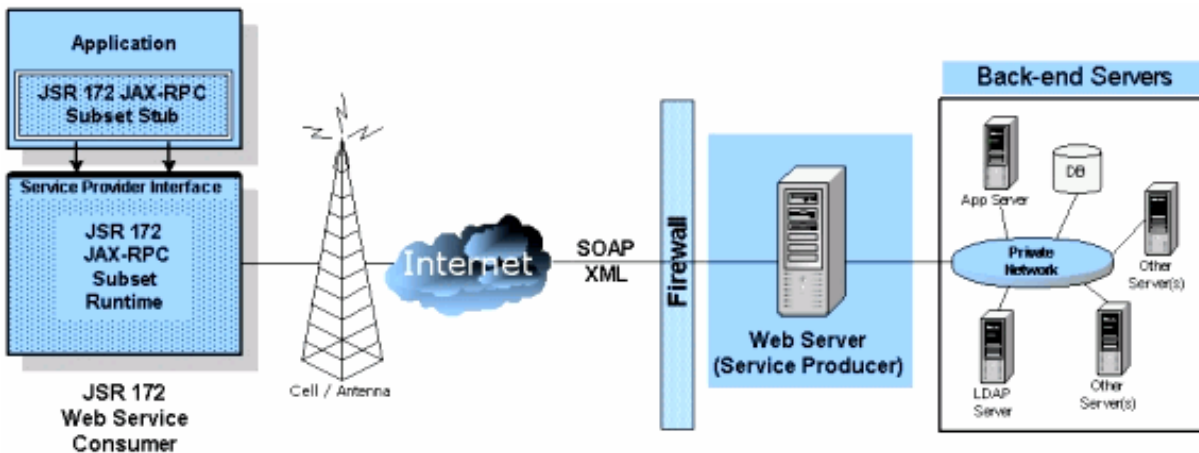


Abbildung 3: JSR-172 Architektur [Introducing to J2ME Web Services]

Diese typische J2ME Web Service Architektur enthält im Wesentlichen drei Stufen. In der ersten Stufe nutzt der JSR-172 Stub die JAX-RPC Runtime um mit einem Netzwerk über das mobile Funk System zu kommunizieren. Als nächstes wird das Internet Protokoll HTTP genutzt, um SOAP Nachrichten zu versenden. In der letzten Stufe befindet sich der eigentliche Web Service, der in vielfältiger Form erscheinen kann.

Weiß man im vornherein noch nicht genau, wo sich der Web Service befindet, so kann man normalerweise mittels UDDI danach suchen. JSR-172 bietet jedoch keine API für eine solche Suche an.

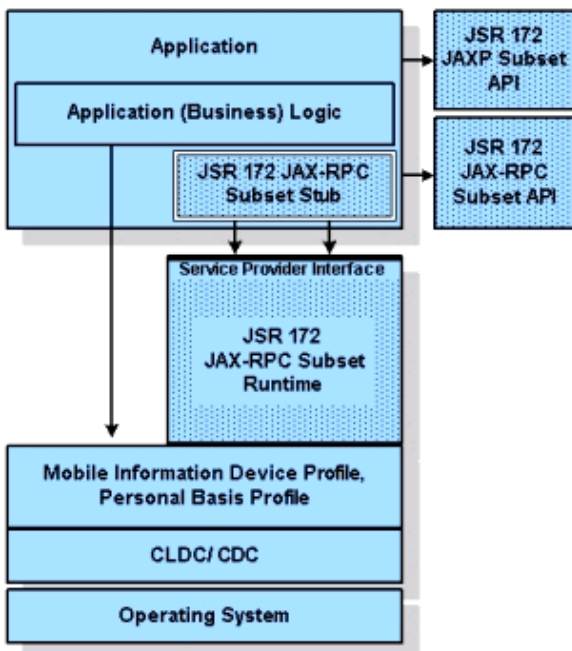


Abbildung 4: JSR-172 Client Architektur [Introducing to J2ME Web Services]

Eine genauere Betrachtung der Applikation auf dem mobilen Gerät zeigt die wesentlichen Elemente des Web Service Client.

Unten steht natürlich das Betriebssystem, z.B. Symbian OS, auf dem dann mit Hilfe der Profile Mobile Information Device Profile (MIDP) oder auch Personal Basis Profile (PBP), der Einschränkungen CLDC/CDC und der Java Virtual Machine zur Ausführung, die eigentliche Applikation gebaut ist. Das zentrale System einer mobilen Applikation zur Nutzung von Web Services mittels Java ist JAX-RPC. Für den Umgang mit XML Dokumenten steht die JAXP API zur Verfügung.

2.4.2. JAX-RPC Subset

JAX-RPC steht für Java API for XML-based Remote Procedure Call und ist für die Erzeugung von Web Services und Clients zuständig, die RPCs nutzen. Ein RPC Mechanismus ermöglicht den Aufruf von Methoden auf anderen Systemen und wird in JAX-RPC als XML-basiertes Protokoll, wie z.B. SOAP, repräsentiert.

JAX-RPC gewährleistet ein absolut plattformunabhängiges System, das auch andere Web Services anwählen kann, die SOAP, HTTP und WSDL nutzen.

Da SOAP Nachrichten sehr komplex werden können, verbirgt JAX-RPC diese Komplexität vor dem Entwickler. So müssen SOAP Nachrichten nicht selbst erzeugt oder geparkt werden. Sie entstehen automatisch, wenn ein Service antwortet oder ein Client anfragt, mit Hilfe des JAX-RPC Laufzeitsystems (JAX-RPC Runtime). Zuständig sind dafür so genannte Proxys, die als Schnittstelle zwischen Client bzw. Server und Internet dienen und für solche Zwecke spezielle Methoden implementiert haben.

Der Proxy auf der Client Seite heißt Stub, der vom Server Tie oder Skeleton. Beide sind für das so genannte Marshalling bzw. Unmarshalling zuständig, d.h. sie wandeln einen Methodenaufruf in eine SOAP Nachricht um oder entschlüsseln eine. Das Interface Stub befindet sich im Package `rpc`.

Package `javax.xml.rpc`

This package contains the core JAX-RPC APIs for the client programming model.

See:

[Description](#)

Interface Summary	
Stub	The interface <code>javax.xml.rpc.Stub</code> is the common base interface for the stub classes.

Die Tie Klasse wird automatisch mit dem Server erzeugt, wohingegen bei der Stub Klasse drei Arten von Erzeugung möglich sind. Die statische Methode kann man sich als normale Code Generierung mittels eines bereitgestellten Tools vorstellen. Die zwei dynamischen Methoden (Dynamic Proxy Invokation und Dynamic Invokation Interface (DII)) werden von JSR-172 nicht unterstützt.

Die statische Methode besteht aus einem Tool das das Interface Stub in eine Klasse implementiert und dabei drei wichtige Methoden verwendet.

```
package javax.xml.rpc;

public interface Stub {
    public void _setProperty(String name, Object value);
    public Object _getProperty(String name);
    public java.util.Iterator _getPropertyNames( );
}
```

Sun's J2ME Wireless Toolkit 2.1 enthält einen solchen Stub Generator und außerdem noch alle nötigen Libraries für die Entwicklung von J2ME Web Services. Das ebenfalls enthaltene Beispiel JSR172Demo demonstriert anschaulich die Nutzung von Web Services.

Package javax.microedition.xml.rpc

Class Summary	
ComplexType	The class <code>ComplexType</code> is a special <code>Type</code> instance used to represent an <code>xsd:complextyp</code> defined in a Web Service's WSDL definition.
Element	The class <code>Element</code> is a special <code>Object</code> used to represent an <code>xsd:element</code> defined in a Web Service's WSDL definition.
Operation	The <code>javax.microedition.xml.rpc.Operation</code> class corresponds to a <code>wsdl:operation</code> defined for a target service endpoint.
Type	The class <code>Type</code> is a type safe enumeration of allowable types that are used to identify simple types defined in a Web Service's WSDL definition.

Das zweite `rpc` Package bietet noch eine Datenumwandlung zwischen Java und WSDL an. Die Umwandlung deckt einfache Datentypen wie `short`, `int`, `long`, `float` und `double` ab, hat aber auch Regeln um Arrays, Enumerations oder noch komplexere Datentypen abzubilden. Dieser Teil der API ist jedoch mehr für Entwickler von Code Generation Tools. Für den normalen Anwender verbirgt sich das Ganze hinter der Stub Datei.

Das letzte wichtige Package `rmi` definiert alle erforderlichen Exceptions, wenn z.B. beim Marshalling ein Fehler auftritt oder kein Web Service gefunden werden konnte.

Package java.rmi

Interface Summary	
Remote	The <code>Remote</code> interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine.

Exception Summary	
MarshalException	A <code>MarshalException</code> is thrown if a <code>java.io.IOException</code> occurs while marshalling the remote call header, arguments or return value for a remote method call.
RemoteException	A <code>RemoteException</code> is the common superclass for a number of communication-related exceptions that may occur during the execution of a remote method call.
ServerException	A <code>ServerException</code> is thrown as a result of a remote method invocation when a <code>RemoteException</code> is thrown while processing the invocation on the server, either while unmarshalling the arguments, executing the remote method itself, or marshalling the return value.

2.4.3. JAXP Subset

Die JSR-172 Spezifikation definiert neben der JAX-RPC API, bei der der Nachrichtenverkehr via SOAP dem Entwickler verborgen ist, auch noch die Möglichkeit SOAP Nachrichten, aber auch allgemein XML Dateien, zu parsen und selbst auszuwerten. Man sollte sich jedoch immer im Klaren sein, dass einem keine SOAP Objekt Klassen für `ENVELOPE`, `HEADER` oder `BODY` zur Verfügung stehen, wie es die Java API for XML Messaging (JAXM) in der allgemeinen Spezifikation für Web Services definiert. JSR-172 stellt lediglich die Java API for XML Proceasing (JAXP) zur Verfügung, die XML nur parsen kann.

In Java stehen zwei Hauptmöglichkeiten für das Parsen von XML zur Verfügung. Zu einen ist es die Document Objekt Model (DOM), die jedoch nicht von JSR-172 abgedeckt wird, und zum anderen die Simple API for XML Parsing (SAX). Bei SAX sollte man beachten, dass es SAX 1.0 gibt, was nicht von JSR-172 unterstützt wird, und die unterstützte Version 2.0, die in der mobilen Welt jedoch ohne XML DTDs auskommen muss. DTDs sind Dateien, die ein spezielles Schema für XML definieren.

Ein wichtiger Bestandteil der API ist zum ersten das Package `javax.xml.parsers`, das den SAXParser an sich enthält.

Package `javax.xml.parsers`

Provides classes allowing the processing of XML documents.

See:

[Description](#)

Class Summary	
DocumentBuilder	Defines the API to obtain DOM Document instances from an XML document.
DocumentBuilderFactory	Defines a factory API that enables applications to obtain a parser that produces DOM object trees from XML documents.
SAXParser	Defines the API that wraps an XMLReader implementation class.
SAXParserFactory	Defines a factory API that enables applications to configure and obtain a SAX based parser to parse XML documents.

zum zweiten das Package `org.xml.sax` mit den wichtigsten Exceptions

Package `org.xml.sax`

Provides the classes and interfaces for the Simple API for XML (SAX) which is a component of the [Java API for XML Processing](#).

Exception Summary	
SAXException	Encapsulate a general SAX error or warning.
SAXNotRecognizedException	Exception class for an unrecognized identifier.
SAXNotSupportedException	Exception class for an unsupported operation.
SAXParseException	Encapsulate an XML parse error or warning.

und zuletzt das Package `org.xml.sax.helpers` mit den wichtigsten Klassen für die Behandlung des Parsing Ergebnisses.

Package `org.xml.sax.helpers`

Provides helper classes for the Simple API for XML (SAX) which is a component of the [Java API for XML Processing](#).

See:

[Description](#)

Class Summary	
AttributeListImpl	Deprecated. <i>This class implements a deprecated interface, AttributeList; that interface has been replaced by Attributes, which is implemented in the AttributesImpl helper class.</i>
AttributesImpl	Default implementation of the Attributes interface.
DefaultHandler	Default base class for SAX2 event handlers.
LocatorImpl	Provide an optional convenience implementation of Locator.
NamespaceSupport	Encapsulate Namespace logic for use by SAX drivers.
ParserAdapter	Adapt a SAX1 Parser as a SAX2 XMLReader.
ParserFactory	Deprecated. <i>This class works with the deprecated Parser interface.</i>
XMLFilterImpl	Base class for deriving an XML filter.
XMLReaderAdapter	Adapt a SAX2 XMLReader as a SAX1 Parser.
XMLReaderFactory	Factory for creating an XML reader.

Es besteht im Allgemeinen die Möglichkeit den Client auf dem mobilen Gerät mit einer Stub Datei auszustatten oder man wertet die SOAP Nachricht mit einem SAX Parsers selbst aus. Eine Klasse des

Clients enthält dabei die Methode

```
public void receiveHttpResponse(byte[] bytes) {}
```

Diese Methode wird beim Empfang der SOAP Nachricht aufgerufen und die Nachricht selbst wird als Byte Array übergeben. An dieser Stelle wird auch der Parser mit seinen nötigen Gehilfen initialisiert.

```
Echo handler = new Echo();
```

```
SAXParserFactory factory = SAXParserFactory.newInstance();
```

```
try {
```

```
    SAXParser saxParser = factory.newSAXParser();
```

```
    saxParser.parse( new InputSource(new ByteArrayInputStream(bytes)), handler );
```

```
    } catch (Throwable t) {}
```

Um den SAXParser zu erhalten benötigen wir zunächst eine SAXParserFactory, die mit der Methode newSAXParser() einen SAXParser zurückgibt.

Das Byte Array wird zuerst in ein ByteArrayInputStream und dann in ein InputSource umgewandelt, denn nur so kann er geparkt werden. Wichtig ist hier ganz besonders der handler. Er übernimmt das eigentliche Parsen. Der handler ist vom Klassentyp Echo, das wie folgt aussieht:

```
public class Echo extends DefaultHandler
{
    ...

    public String soap = "";

    ...

    public void startDocument()
    throws SAXException {}

    public void endDocument()
    throws SAXException {}

    public void startElement(String namespaceURI,
                             String sName, // simple name
                             String qName, // qualified name
                             Attributes attrs)
    throws SAXException {}

    public void endElement(String namespaceURI,
                           String sName, // simple name
                           String qName // qualified name
                           )
    throws SAXException {}

    public void characters(char buf[], int offset, int len)
    throws SAXException {}

    ...
}
```

Die überschriebenen Methoden der Klasse `DefaultHandler` werden vom `SAXParser` in einer bestimmten Reihenfolge aufgerufen und man kann dann selbst entscheiden, was aus der SOAP Nachricht wird. Um jetzt trotz fehlender Gliederungsklassen den wesentlichen Teil der Antwort herauszufiltern kann man nach einer bestimmten Stringfolge suchen, von der man weiß, dass sie vor der Antwort steht. Eine andere Möglichkeit besteht in der Einbindung des Packages `kSOAP`, dass alle benötigten Klassen für die Auswertung von SOAP bereit stellt.

2.5. Herausforderungen

Die wichtigsten Werkzeuge für die Entwicklung von Web Services Clients für mobile Geräte sind nun definiert. Man steht jedoch noch vor der Herausforderung die Applikation so zu programmieren, dass sie sich an veränderte Situationen anpasst und eine Sitzung wieder aufnehmen kann, wenn eine Unterbrechung stattfindet.

2.5.1. Content Awareness

Entwickler sollten mobile Applikationen so schreiben, dass sie sich automatisch an einen (veränderten) Kontext anpassen können. Ein Kontext ist nach Dey und Abowd wie folgt definiert:

...Kontext ist eine Information, welche die Situation einer Entität beschreibt. Die Gesamtheit aller Konfigurationsmerkmale dieser Entität ist ihr Kontext.

...Ein System heißt 'context aware', wenn es den Kontext verwendet, um relevante Informationen und/oder Dienste dem Benutzer zur Verfügung zu stellen, wobei die Relevanz von der Rolle des Benutzers abhängt.

Ein Kontext kann sich dabei auf vielerlei Eigenschaften beziehen. Beispielsweise kann eine Ortsangabe von einem Web Service zu einem mobilen Gerät relativ oder aber auch absolut sein oder die Verbindungsort kann zwischen GPRS, UMTS und WLAN wechseln, aber auch die Zielplattform kann unterschiedliche Formen annehmen, z.B. Linux, Windows, Solaris. Eine Applikation sollte in der Lage sein diese Umstände automatisch anzupassen.

2.5.2. Sessionmanagement

Eine andere Herausforderung ist ein flexibles Session-Management. Man sollte sich immer vor Augen halten, dass die Sitzung zwischen Web Service und mobilen Gerät ständig unterbrochen werden kann, wenn z.B. das mobile Gerät in eine empfangslose Zone kommt oder der Benutzer sein Gerät wechselt und sein PDA gegen ein Handy tauscht. Der Service sollte in der Lage sein gegebenenfalls Information über den Benutzer zu speichern, z.B. mit Cookies, und eine Sitzung schnell wieder aufzunehmen.

3. J2EE Client Provisioning

Hat man die Herausforderungen des Sessionmanagements und der Content Awareness bewältigt, so steht man vor dem Problem die Client Software auf das mobile Gerät zu bringen. Die wachsende Nachfrage nach mobilen Geräten eröffnete da ganz neue Tore. Ein System für „Plug and Play“ Applikationen, also Applikationen die man sich schnell und einfach nur bei Bedarf holt, wäre da eine wünschenswerte Lösung. Mit der Spezifikation für Java 2 Enterprise Edition (J2EE) Client Provisioning bietet Sun eine Möglichkeit dies in Java umzusetzen.

3.1. Provisioning Server

Ein Provisioning Server bietet Clients über ein Web Interface verschiedene Sachen zum Herunterladen an, z.B. Fotos, Programme oder Lieder. Man kann sich das Ganze in etwa wie einen Zigarettenautomat vorstellen: Der Benutzer nähert sich dem Automaten, macht sich klar was dieser ihm anbietet, wählt seine Zigaretten, zahlt dafür und der Automat schickt ihm seine Wahl zu. Ein weiterer Vorteil ist die Möglichkeit auch ganze Archive mit Inhalt Hochladen zu können.

Die Clients können zwischen mobilen Geräten, wie PDAs oder Handys, und ganz gewöhnlichen Desktop Systemen variieren. Die angeforderte Funktionalität ist somit auch unterschiedlich. Man unterscheidet zwischen Simple Clients, die die Web Seiten des Provsioning Servers nutzen, und den Rich Clients, die Multimedia Inhalte, wie Film, Audio usw., unterstützen. In Java können Rich Clients auf der Java 2 Micro Edition (J2ME) oder auf der Java 2 Standard Edition (J2SE) entwickelt werden. Ein J2EE Client Provisioning Server ist jedoch nicht auf Java Clients beschränkt, auch andere Plattform Applikationen, die gewisse Regeln einhalten, werden unterstützt.

3.2. Architektur

J2EE Client Provisioning Server stellen einen zentralen Behälter (Repository) für verschiedene Inhalte, von Client Applikationen bis hin zu Multimedia Inhalten, bereit.

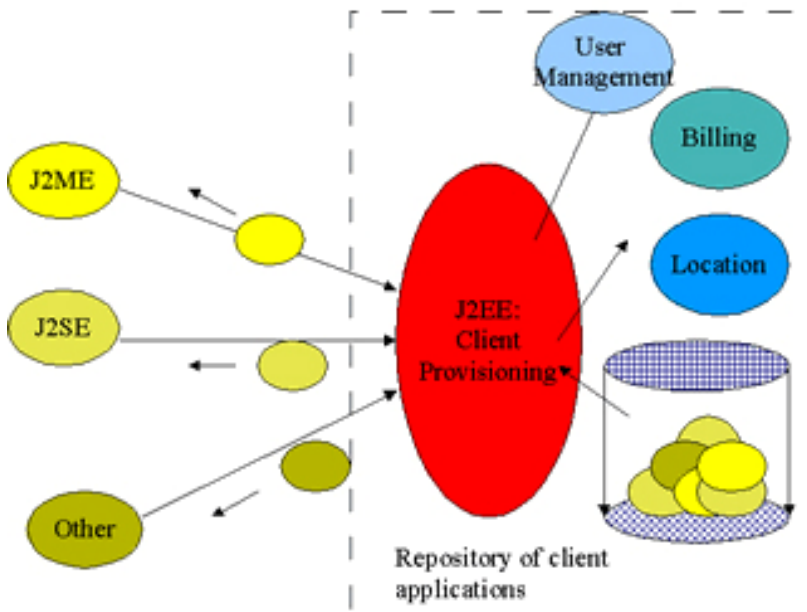


Abbildung 5: J2EE Client Provisioning Architektur [J2EE Client Provisioning Overview]

Die Grafik zeigt die Interaktion zwischen Clients und dem Provisioning Server. Die Clients suchen aus dem Repository etwas aus und erhalten dann ihre Wahl. Es lassen sich auch Inhalte hochladen und der Server ist mit Systemen für Zahlungen (Billing) und einer Benutzer Verwaltung (User Management) ausgestattet.

Die zentralen Elemente eines J2EE Client Provisioning Servers sind zum einen die Konfiguration Dateien `devices.xml`, `adapters.xml` und `matchers.xml` und zum anderen das Provisioning Archiv (PAR). `device.xml` definiert die unterstützten Geräte, `adapters.xml` ist für die unterstützten Plattformen zuständig und `matchers.xml` enthält Regeln zur Überprüfung von gewissen Voraussetzungen oder ob Geräteeinschränkungen eingehalten wurden.

3.2.1. Adapters.xml

Diese Datei beinhaltet spezielle Provisioning Modelle. Ein Provisioning Model ist die Art und Weise wie ein Client mit dem Server interagiert. Es unterstützt spezielle Protokolle zum Herunterladen der Inhalte und legt die Plattform und dessen Voraussetzung fest. `adapters.xml` enthält beispielsweise das MIDP Over The Air (MIDP OTA) Protokoll mit dem mobile Geräte festgelegt werden und das Java Network Launching Protocol (JNLP) für Anwendungen der J2SE.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<adapters
  xmlns="http://java.sun.com/xml/ns/j2ee-cp"
```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee-cp
Adapters_1_0.xsd">

<!-- This is the MIDP OTA adapter. -->
<adapter>
  <adapter-name>midp</adapter-name>
  <adapter-class>com.sun.provisioning.adapters.midp.AdapterMIDP
    </adapter-class>
  <base-uri>/delivery/midp</base-uri>
  <descriptor-file>
    <extension>jad</extension>
    <mime-type>text/vnd.sun.j2me.app-descriptor</mime-type>
  </descriptor-file>
  <fulfillment-duration>0</fulfillment-duration>
  <init-param>
    <param-name>install-notify</param-name>
    <param-value>true</param-value>
  </init-param>
</adapter>
</adapters>

```

Der adapters.xml Code enthält genau einen Adapter für mobile Geräte. <adapter-name> gibt dem Adapter einen eindeutigen Namen, <adapter-class> legt die Klasse fest, die für den Umgang mit den Geräten zuständig ist. Für den Aufruf der jeweiligen Seiten des Provisioning Servers, hier der Teil mit den MIDP Applikationen, braucht man dafür eine eindeutige URI, die an die URL des Servers angehängt wird. Eine URI definiert eindeutig Objekte im Internet, URL ist ein Spezialfall davon. <base-uri> gibt diese an. In <descriptor-file> wird noch festgelegt welche Art von Datei versendet, bzw. empfangen, wird und ansonsten stehen noch einige Tags für spezielle Parameter bereit.

3.2.2. Devices.xml

Devices.xml legt nun genau fest welche Geräte mit welchen Fähigkeiten unterstützt werden, welche Protokolle der adapters.xml dafür verwendet werden und außerdem noch ein Parsing System für HTTP Header zur Feststellung der Geräte Art.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<devices
xmlns="http://java.sun.com/xml/ns/j2ee-cp"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee-cp
Devices_1_0.xsd">

<!-- Motorola i95cl -->
<device>
  <identifier>Motorola/i95cl</identifier>
  <adapter-name>midp</adapter-name>
  <adapter-name>generic</adapter-name>
  <capability>
    <capability-name>HardwarePlatform.ScreenSize</capability-name>
    <capability-value>120x160</capability-value>
  </capability>
  <capability>
    <capability-name>HardwarePlatform.BitsPerPixel</capability-name>
    <capability-value>8</capability-value>
  </capability>

```

```

<capability>
  <capability-name>SoftwarePlatform.JavaPlatform</capability-name>
  <capability-value>MIDP/1.0</capability-value>
</capability>
<capability>
  <capability-name>SoftwarePlatform.JavaPackage</capability-name>
  <capability-value>com.motorola.lwt</capability-value>
</capability>
<capability>
  <capability-name>SoftwarePlatform.JavaProtocol</capability-name>
  <capability-value>comm, socket, https, ssl, datagram, file
  </capability-value>
</capability>
</device>
</devices>

```

Der Code zeigt das `<device>` einer `devices.xml` Datei zur der Festlegung aller unterstützten Geräte. Dabei unterscheidet man zwei Arten von Geräten. Zum einen gibt es Hardware Geräte wie Handys, in dem Fall Motorola i95cl und zum anderen können auch Software Geräte wie ein Web Browser, Java 1.4 oder MIDP 2.0 festgelegt werden. Hier ist nur ein Hardware Gerät aufgelistet, das den eindeutigen Namen in `<identifier>` hat. Der zuständige Adapter steht in `<adapter-name>`. Wichtig sind die `capability` Tags, die alle Fähigkeiten eines Gerätes auflisten.

Nun muss man natürlich auch noch feststellen können um welches Gerät es sich handelt, wenn eine Anfrage über HTTP an den Provisioning Server gesendet wird. Dazu enthält `devices.xml` ein Tag namens `<device-mapping>`, das den HTTP Header nach bestimmten Informationen absucht.

```

<device-mapping>
  <identifier>Motorola/i95cl</identifier>
  <request-mapping>
    <header-name>user-agent</header-name>
    <header-value>Motorola/i95cl</header-value>
  </request-mapping>
</device-mapping>

```

Zusätzlich lassen sich die Übereinstimmungsprüfungen noch mit den Tags `<map-any>`, `<match-all>` und `<match-not>` verfeinern.

3.2.3. Matchers.xml

Die Fähigkeiten eines Gerätes können verschiedenartig gehandhabt werden. `Matchers.xml` arbeitet alle in `devices.xml` vorkommenden Fähigkeiten ab und definiert Regeln für den Umgang.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<matchers
  xmlns="http://java.sun.com/xml/ns/j2ee-cp"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee-cp
  Matchers_1_0.xsd">

<matcher>
  <attribute-name>SoftwarePlatform.CcppAccept-Charset</attribute-name>
  <matcher-class>javax.provisioning.matcher.StringMatcher</matcher-class>
  <init-param>
    <param-name>allMustMatch</param-name>
    <param-value>false</param-value>
  </init-param>
  <init-param>

```

```

        <param-name>caseInsensitive</param-name>
        <param-value>true</param-value>
    </init-param>
    <init-param>
        <param-name>wildcardsOn</param-name>
        <param-value>true</param-value>
    </init-param>
</matcher>
</matchers>

```

<attribute-name> zeigt die jeweilige Fähigkeit und <init-param> legt drei Arten von Unterstützung fest: bei allMustMatch müssen alle Voraussetzung zusammenpassen, caseInsensitive sieht von Groß- und Kleinschreibung ab und akzeptiert auch Verbindungen mit „-“ oder „_“ in dem zu prüfenden String und mit wildcardsOn gehen ungefähre Strings mit einem „*“ durch. Alle Parameter lassen sich aus oder einschalten. Mit String sind hier die einzelnen Informationen aus dem Header der Anfrage gemeint.

3.2.4. Provisioning Archiv (PAR)

Nach der Beschreibung von Geräten und ihren Fähigkeiten, benötigt man nun ein Archiv das alle Dinge lagert, die angeboten werden. Dieses Archiv kann als Ganzes auf den Server geladene werden. Ein Provisioning Archiv ist ein ZIP Archiv, das im Hauptverzeichnis den ganzen Inhalt des Archives (z.B. .jar, .jad, .jnlp, .gif, .jpg oder andere Datentypen) gelagert hat und in dem Unterordner /META-INF mit der Datei provisioning.xml den Inhalt beschreibt.

```

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

<client-bundle>
    <content-id>
        http://java.sun.com/products/j2ee/provisioning/demos/midlets/nim
    </content-id>
    <bundle-type>APPLICATION</bundle-type>
    <descriptor-file>/nim.jad</descriptor-file>

    <user-descriptions>
        <display-name>Nim Game</display-name>
        <description>The game of Nim.</description>
        <icon>/JavaPowered-8.png</icon>
    </user-descriptions>

    <vendor-info>
        <vendor-name>Sun Microsystems, Inc.</vendor-name>
        <vendor-url>http://www.sun.com</vendor-url>
        <vendor-description></vendor-description>
    </vendor-info>

    <copyright>Copyright 2003 Sun Microsystems, Inc. All Rights Reserved
    </copyright>

    <device-requirement>
        <requirement-name>SoftwarePlatform.JavaPlatform</requirement-name>
        <requirement-value>MIDP/1.0+</requirement-value>
    </device-requirement>

    <catalog-property>
        <property-name>Category</property-name>
        <property-value>Game</property-value>
    </catalog-property>
</catalog-property>

```

```

    <property-name>Billing Plan</property-name>
    <property-value>Price List</property-value>
  </catalog-property>
</client-bundle>
</provisioning-archive>

```

Die Datei stellt eine Menge an Informationen über die Applikation (<user-descriptions>), den Hersteller (<vender-info>), Copyright (<copyright>) und über den die Art der Lagerung (<catalog-property>) zur besseren Auffindung der Applikation. Im Tag <device-requirement> findet man die Geräte und deren Fähigkeiten wieder, so wie man sie in adapters.xml, devices.xml und matchers.xml definiert hat. Es wird angegeben was diese Applikation voraussetzt.

Ein PAR Archiv kann mit Hilfe eines Web Interfaces hochgeladen werden, so dass andere Clients dessen Inhalt nutzen können.

3.3. Provisioning Client

Die J2EE Client Provisioning Architektur benötigt für die Kommunikation mit dem Server einen Client. In Java sollte dieser die Protokolle MIDP OTA für das Herunterladen von MIDP Applikationen für mobile Geräte oder JNLP für das Herunterladen von J2SE Applikationen unterstützen. Dem Client stehen dann drei Hauptfunktionalitäten zur Verfügung: Discovery, Delivery und Stocking.

3.3.1. Discovery

Discovery ist eine Funktion des Provisioning Servers, bei der der Client den Server nach Inhalten absucht, die er Herunterladen kann. Der Server stellt dafür J2EE Komponenten, die sich nach der Spezifikation JSR-124 richten und drei Hauptaufgaben erfüllen:

- Festlegung des Eigenschaften des Endbenutzers
- Entwicklung von URIs für die Objekte des Repositorys, sodass der Client gezielt herunterladen kann
- Abfrage des Repositorys mit Hilfe von Listen der Komponenten die heruntergeladen werden können

Es existieren mehrere Möglichkeiten einen Client zu implementieren: Man kann sich dem MIDP OTA Protokoll vom Java Wireless Toolkit 2.1 (WTK) bedienen, das nach der Installation des WTK 2.1 im Startmenü unter OTA Provisioning zu finden ist. Das Protokoll wird dabei mit einem Emulator für mobile Geräte gestartet. JNLP findet man beim Java Web Start in jeder Standardinstallation von J2SE.

Ohne diese zwei Tools besteht auch die Möglichkeit eine Web Browser basierte Anfrage an den Server zu stellen. Dazu schreibt der Provisioning Entwickler eine Java Servlet, ein JSP oder ein JSF, das sind Java Web Technologien, die mit dem Browser aufgerufen werden können. Die Provisioning API definiert ein Interface namens `javax.provisioning.ProvisioningContext` mit dem Entwickler herausfinden können, welche Fähigkeiten das System hat, das die Anfrage an den Server stellt.

3.3.2. Delivery

Nach der Erforschung des Angebotes steht nun die Versendung des gewünschten Inhaltes an (Delivery). Der Discovery Prozess produziert mit der Methode `FullfillmentTask.getDeliveryURI()` eine Liste von gewünschten URIs, die dem Client angezeigt werden. Der `FullfillmentTask` der Provisioning API erledigt hierbei alle nötigen Delivery Aufgaben. Jeder `FullfillmentTask` hat eine eindeutige `fulfillment ID`.

3.3.3. Stocking

Wenn der Entwickler fertig mit der Erstellung des Provisioning Portals, der Aufbewahrungsort für die Repositorys, ist und der J2EE AppServer mit den Provisioning Server hochgefahren wurde, müssen nun die PAR Archive hoch- und runtergeladen werden, damit andere Clients den Dienst auch nutzen können. Dies nennt man Stocking. Das ganze passiert z.B. über Web Interface wie das Beispielprogramm RI-Test der J2EE Client Provisioning Reference Implementation zeigt.

Bundle Repository Administration

Repository Contents

PAR	Uploaded	Bundles	
6	Thu May 15 14:55:51 PDT 2003	17	remove

Upload a PAR File

Upload a PAR file by entering its path and clicking "Upload PAR file". For example, click on Browse and go to .../test/apps/test.par.

PAR file:

Empty the Repository

If you want to start over, you can empty out the repository by clicking on this button:

Abbildung 6: Web Interface für PAR Verwaltung [JSR-124 Reference Implementation]

3.4. Anwendung und Ausblick

Mit diesen drei Hauptfunktionen ist es nun Sache des Clients ein vorgefertigtes Provisioning Archiv mit Software oder anderen Dingen zu erweitern, die Angebote zu nutzen, oder einfach nur zu erforschen was zur Verfügung steht. Ein Angebot wäre beispielsweise die Bereitstellung von Client Software für Web Services, die auf bestimmte Geräte und Server abgestimmt ist.

3.4.1. Anwendungsbeispiele

Die Client Software könnte z.B. ein Routenplanungssystem sein. Ein Routenplaner bietet dem Benutzer eine aktuelle Wetter und Stauinformation und erstellt gegebenenfalls eine neue Route zur Umlenkung, die schneller und unkomplizierter gesendet werden kann, wie mittels eines WAP Zugangs. Liebhaber von interaktiven Spielen können mobile Spiele mittels der Anwahl eines Services online gemeinsam nutzen. Ein Hindernis ist bis jetzt natürlich noch der hohe Preis eines mobilen Internet Zugangs. Weiterhin können Ärzte z.B. auf Patientendaten zurückgreifen, wenn ein Unfall passiert ist. Sie benötigen dazu eine Client Software, die mit dem Krankenhaus oder einer anderen zentralen Stelle für Daten verbunden ist. Der Provisioning Server kann dabei auch auf einem Krankenhaus Server liegen. Dort weiß man am besten welche Voraussetzungen die Software erfüllen muss.

Zentral bei der Anwendung eines Provisioning Servers ist die „On-Demand“-IT, bei der der Benutzer nur die Software herunterlädt und nutzt, die er im Augenblick benötigt. Das erspart ihm die Belegung von unnötigem Speicherplatz auf dem mobilen Gerät.

3.4.2. Ausblick

Für die Erstellung von derartiger Software hat sich neben Sun auch Microsoft mit einem Mobile Internet Toolkit for .NET auf dem Markt für mobile Web Services etabliert. Aber auch hier muss auf einen direkten SOAP Umgang verzichtet werden. Toolkits wie PocketSOAP und kSOAP schaffen da aber Abhilfe und es ist abzusehen das Sun bei zukünftigen Spezifikationen und Wireless Toolkits dieses Feature mit einbauen wird. Auch bei mobilen Geräten mangelt es noch an Unterstützung für das erforderliche MIDP 2.0 und bei manchen Geräten laufen die Erweiterungstoolkits nur teilweise. Das Nokia 7650 hat beispielsweise Problem mit kSOAP. Dieser Markt ist somit noch längst nicht vollständig erschlossen und man kann sich auf ständige interessante Neuerungen freuen.

4. Quellen

4.1. J2ME Web Services (JSR-172)

Java ist auch eine Insel, Christian Ullenboom, Galileo, 2002
 Moderne Konzepte Verteilter Systeme , D Web Services, Franz J. Hauck, Uni Ulm, 2002/2003
 (<http://www-vs.informatik.uni-ulm.de/teach/ws02/mkvs/Uebung/MKVS-Uebung-D.pdf>)
 Developing real world Web Services using J2ME, J2SE, J2EE, Sang Shin and Carol McDonald, Sun, 2003
 (<http://developer.sun.com/events/techdays/presentations/seattle/CodecampRealWorldWebServices.pdf>)
 Spezifikation J2ME Web Services JSR-172, Jon Ellis and Mark Young, Sun, 2003
 (<http://jcp.org/aboutJava/communityprocess/final/jsr172/index.html>)
 Mobile Web Services, Dr. Thomas Wieland, Siemens, 2002
 (http://www.cpp-entwicklung.de/download/MobileWebservices_oop.pdf)
 Java Web Services, David Chappell and Tyler Jewell, O'Reilly, 2002
 Java Web Services Tutorial, Sun, 2003
 (<http://java.sun.com/webservices/docs/1.1/tutorial/doc/>)
 XML Tutorial, 2000
 (http://www.uzi-web.de/xml/xml_toc.htm)
 Evaluation of Mobile Web Services, XML & Web Services Magazine, 2002
 (http://www.fawcette.com/xmlmag/2002_08/magazine/columns/jjurvis/)
 Introducing to J2ME Web Services, Sun, 2003
 (<http://developers.sun.com/techttopics/mobility/apis/articles/wsa/>)
 Web Services registrieren, finden und benutzen, UDDI, Stefan Hubert, 2002
 (<http://www.ai.informatik.fh-furtwangen.de/~hubert/SeminarAusarbeitung/default.htm>)
 SOAP, Torsten Horn, 2004
 (<http://www.torsten-horn.de/techdocs/soap.htm#SOAP>)

4.2. J2EE Client Provisioning

J2EE Client Provisioning Spezifikation JSR-124, Danny Coward and Dave Bowen, Sun, 2003
 (<http://jcp.org/aboutJava/communityprocess/final/jsr124/index.html>)
 JSR-124 Reference Implementation, Sun, 2003
 (<http://developers.sun.com/techttopics/mobility/midp/articles/provisioninggetstart/>)
 J2EE Client Provisioning Overview, Sun 2004
 (<http://java.sun.com/j2ee/provisioning/overview.html>)
 Java Client Provisioning Tutorial, Benoy Jose, Javaboutique, 2003
 (<http://javaboutique.internet.com/ClientProvisioning/>)