Seminar-Ausarbeitung: Web Services

Reinhold Rumberger

Institute of Architecture of Application Systems (IAAS),
University of Stuttgart
rumberrd@studi.informatik.uni-stuttgart.de

Inhaltsverzeichnis

Abstract. Large present-day businesses often use service oriented architectures (SOAs) to implement their business processes. One way to implement a SOA is through web services. This paper will provide an introduction into the basics of web services. It will then focus on the most important technologies used to implement web services in the Java programming environment.

1 Einleitung

In der heutigen Software-Welt wird zunehmend auf Service-Orientierte Architekturen (SOAs) gesetzt. Das bedeutet, dass Anwendungen aus einzelnen Diensten aufgebaut werden, die sehr lose gekoppelt sind und über ein Netzwerk miteinander kommunizieren können. Durch diese Eigenschaften lassen sich auf SOA basierende Anwendungen schnell und effizient an neue Marktbedingungen anpassen.

SOAs können durch Web Services implementiert werden. Für Java-Umgebungen spielen hier vor allem JAX-WS 2.0 (Java API for XML-Based Web Services, JSR-224), WS-Metadata 2.0 (JSR-181) und JAXB 2.0 (Java Architecture for XML Binding, JSR-222) eine Rolle. Diese Spezifikationen werden den Kern der Web Services in DecidR bilden.

2 SOA

Eine „SOA“ (Service Oriented Architecture) ist eine Software-Architektur, bei der lose gekoppelte Dienste miteinander kommunizieren. Die SOA definiert die verfügbaren Kommunikationsmethoden und die Methoden zum Auffinden von Diensten. So können verschiedene Dienste miteinander verbunden werden, um einen bestimmten Geschäftsprozess zu implementieren. Mehrere Dienste werden kombiniert um eine Applikation zu realisieren.

Die lose Kopplung der Dienste macht die durch sie implementierten Applikationen sehr flexibel. So kann relativ schnell auf veränderte Umweltbedingungen reagiert werden. Wenn beispielsweise Dienste zufällig aus einer Menge geeigneter Dienste ausgewählt werden, kann man einer gesteigerten Nachfrage durch Hinzufügen weiterer Dienste begegnen. Dabei kann es sich sowohl um neue Instanzen vorhandener Dienste handeln, als auch um komplett neu entwickelte Dienste, die die gleiche Aufgabe erledigen.

3 Web Services Grundlagen

Das W3C definiert einen „Web Service“ folgendermaßen:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). [...][4]

Diese Definition enthält drei Haupteigenschaften von Web Services:

interoperable Web Services sind so ausgelegt, dass sie unabhängig von ihrer Implementierung und der Infrastruktur zusammenarbeiten können. Dadurch sind Web Services prinzipiell plattformunabhängig, da sie jederzeit auf einer anderen Plattform implementiert werden können.

machine-to-machine interaction Web Services sind nicht konzipiert, um mit Menschen zu interagieren. Jegliche Kommunikation über das Netzwerk findet ausschließlich zwischen Web Services statt. Dadurch ist ein aus Web Services bestehendes Softwaresystem prinzipiell vollständig automatisiert. Es ist selbstverständlich auch möglich, dass ein Web Service Aufgaben an einen Menschen weiterleitet und das System so nur teilautomatisiert ist.

over a network Web Services kommunizieren über ein Netzwerk. Das ermöglicht eine Lastverteilung und den zielgerichteten Einsatz spezialisierter Hard- und Software. Es wirft aber auch ein Problem auf: Während der Implementierung ist nicht immer bekannt, wo sich der Web Service befinden wird. Deshalb existieren Dienste, die eine Liste der bekannten Web Services und ihrer Kontaktinformationen bereitstellen.

Es gibt zur Zeit zwei relevante Ausprägungen von Web Services: die „message-oriented Web Services“ und die „RESTful Web Services“. In der Vergangenheit stellten RPC-basierte Web Services die einzige Ausprägung dar. Diese Web Services bauten auf „remote procedure calls“ auf. Heute ist diese Ausprägung nicht mehr relevant und wird hier ignoriert.
Es folgt eine kurze Vorstellung der relevanten Ausprägungen:

RESTful Web Services: Diese Ausprägung orientiert sich stark an HTTP. Deshalb wird die Schnittstelle dieser Sorte Web Services auf wenige, von HTTP bereitgestellte, Operationen beschränkt (z.B. GET, PUT und DELETE). Durch die Beschränkung der Operationen ist eine bessere Integration mit dem HTTP-Protokoll möglich. So kann der Aufwand verringert werden, der für die Serialisierung/Deserialisierung von Konstrukten einer Hochsprache in ein XML-Format zu Transportzwecken nötig ist. Allerdings lassen sich so mit vertretbarem Aufwand nur sehr einfache Web Services implementieren. Komplexere Web Services würden einen hohen Kommunikations- und Wartungsaufwand erfordern.
Diese Ausprägung ist nicht standardisiert, was dazu führt, dass nicht ganz klar ist, was einen RESTful Web Service ausmacht. Somit ist diese Ausprägung für DecidR uninteressant.[1]

message-oriented Web Services: Diese Ausprägung wird auch „Big Web Services“ genannt. Während RPC-basierte Web Services Prozedur-Aufrufe und RESTful Web Services durch MIME identifizierte Objekte über HTTP als Kommunikationseinheit verwenden, benutzen message-orientierte Web Services abstrakte

[1] Diese Darstellung ist stark vereinfacht. Deshalb sind einige der Möglichkeiten der RESTful Web Services unterschlagen worden. Da message-oriented Web Services für uns wesentlich interessanter sind, ist diese grobe Darstellung jedoch ausreichend. Für Interessierte sei auf [5], Abschnitt 3 verwiesen.

Nachrichten. Im Gegensatz zu RPC-basierten Web Services können so verschiedene binär inkompatible Programmiersprachen verwendet werden. Im Gegensatz zu RESTful Web Services sind message-oriented Web Services unabhängig vom Transportprotokoll. Dadurch können sie flexibel den Gegebenheiten angepasst werden und es ist möglich, „Qualtity of Service“ (QoS) auch dann zu gewährleisten, wenn die Kommunikationspartner mehrere Netzwerke mit verschiedenartiger Infrastruktur trennen. Diese Unterschiede der Infrastruktur können sowohl in der verwendeten Hardware als auch im verwendeten Protokollstack liegen.
Da nur eine von der Implementierung unabhängige Schnittstelle veröffentlicht wird, ist eine lose Kopplung gegeben. Diese Schnittstelle ist meist in WSDL beschrieben.

Um die Interoperabilität von Web Services zu verbessern, publiziert das WS-I[6] sogenannte Profile. Profile bestehen aus definierten Spezifikationen (z.B. SOAP, WSDL) und den zugelassenen Versionen (z.B. SOAP 1.2, WSDL 2.0). Hinzu kommen noch zusätzliche Einschränkungen (z.B. „der SOAP-Body darf nur genau einen Unterknoten haben“), die Unklarheiten der Spezifikationen ausräumen. Außerdem publiziert das WS-I Anwendungsfälle und Testwerkzeuge um das Deployen profilkonformer Web Services zu erleichtern.
Da diese Profile die Zahl der einsetzbaren Standards stark einschränken, erleichtert die Beschränkung auf ein Profil (z.B. das „WS-I Basic Profile“) die Implementierung und das Deployment erheblich. Existierende Frameworks und IDEs kennen und unterstützen diese Profile und können aufgrund der Einschränkungen detailliertere Annahmen machen, was den Einsatz von Deployment Deskriptoren weitgehend überflüssig macht und den benötigten Abstraktionsgrad der Laufzeitumgebung erheblich verringert.

Es gibt des weiteren Standards, die die Fähigkeiten von Web Services erweitern. Diese Standards haben meist einen Namen nach dem Schema „WS-`x`“ (z.B. WS-Security, WS-Transaction). Sie bauen auf den Basis-Standards (SOAP, WSDL, etc.) und aufeinander auf um zusätzliche Features wie End-to-End-Verschlüsselung oder Transaktionssicherheit zu implementieren. Dieser modulare Aufbau minimiert den Aufwand, der nötig ist, um einen Web Service mit zusätzlichen Features auszurüsten oder einen Standard gegen einen anderen, besser geeigneten auszutauschen.

3.1 SOAP

SOAP ist ein Nachrichtenprotokoll, das den Austausch strukturierter Daten zwischen vernetzten Web Services erleichtert. Das wird dadurch erreicht, dass das Nachrichtenformat standardisiert ist und auf XML aufbaut. SOAP selbst definiert keine Methoden zum Übertragen von Daten; stattdessen stützt es sich auf andere Protokolle der Anwendungsschicht (z.B. HTTP, SMTP, FTP) für die konkrete Datenübermittlung. Bei der Wahl des Transportprotokolls ist Vorsicht geboten; einige Protokolle werden standardmäßig von Firewalls gefiltert und sollten deshalb gemieden werden. SOAP stellt die Basis der Kommunikation zwischen Web Services dar.

Die Wahl von XML als Nachrichtenformat hat sowohl Vor- als auch Nachteile. Positiv sind die leichtere Lesbarkeit für Menschen, die bessere Interoperabilität und die vereinfachte Fehlersuche. Außerdem sind die Nachrichten validierbar, wodurch Fehler durch ein nicht unterstütztes Datenformat ausgeschlossen werden können und bei der Implementierung fehlerhafte Datentypen nicht berücksichtigt werden müssen. Dank der Verwendung von XML ist SOAP leicht erweiterbar. Allerdings ist XML etwas unhandlich und verlangsamt die Verarbeitungsgeschwindigkeit durch seine Verbosität. Dieses Problem kann jedoch mit Binary XML und optimierten XML-Parsern minimiert werden.

Eine minimale SOAP-Nachricht (siehe Abbildung 1) besteht aus einem `SOAP-Envelope`, der einen `SOAP-Header` und einen `SOAP-Body` enthält. Während der `SOAP-Body` die tatsächliche Nutzlast der Nachricht enthält, stehen im `SOAP-Header` Metadaten, die für die Datenverarbeitung und -übertragung wichtige Parameter enthalten. Zusätzlich unterstützt SOAP sogenannte „Features“, die Einfluss auf die Datenverarbeitung und -übertragung nehmen können. Da diese jedoch unter das Thema „Erweiterbarkeit von SOAP“ fallen, sprengen sie den Rahmen dieser Arbeit.

```
<?xml version="1.0"?>
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
    <s:Header>
    </s:Header>
    <s:Body>
    </s:Body>
</s:Envelope>
```

Abbildung 1. Minimale SOAP-Nachricht

3.2 WSDL

An dieser Stelle sei auf den Foliensatz „Workflow, Objects and Web Services“[11] der Vorlesung „Workflow Management“[7] verwiesen. Dort befindet sich wichtige Terminologie, die an dieser Stelle vorausgesetzt wird.

WSDL (Web Services Description Language) ist eine XML-basierte Sprache zur Beschreibung der öffentlichen Schnittstellen von Web Services und ihrer Zugriffsmethoden. Ein WSDL-Dokument ist typischerweise mit XML-Schema validierbar. Die wichtigsten Elemente eines WSDL-Dokuments sind:

types Definiert die Datentypen, die in den Messages verwendet werden (siehe Abbildung 2). Diese Definition ist abstrakt und unabhängig von der Implementierungssprache des Web Service.

messages Beschreibt die Nachrichten, die ein Web Service akzeptiert und versendet (siehe Abbildung 6). Entspricht den Parametern und dem Rückgabewert in Java.

portType Beschreibt die verfügbaren Operationen und ihre Messages (siehe Abbildung 3). Eine WSDL-Operation entspricht einer Funktion in klassischen Programmiersprachen. Web Services mit dem gleichen `portType` implementieren per Definition die gleiche Funktion.

binding Definiert das Nachrichtenformat und ein Mapping auf ein Protokoll, das zur Datenübertragung verwendet wird (siehe Abbildung 5). Standard-Bindings sind für SOAP über HTTP definiert. Prinzipiell können aber zusätzliche Bindings selbst definiert werden oder durch die Laufzeitumgebung bereitgestellt werden.

service Definiert einen Zugangspunkt für jedes Binding (siehe Abbildung 4). Diese Zugangspunkte werden auch „Endpoints“ genannt. Dieses Element wird meist zur Laufzeit generiert und spezifiziert einen Zugriffspunkt für den Web Service.

Zusätzlich gibt es noch das `documentation`-Element, das – ähnlich wie Javadoc für Java – Dokumentation für WSDL-Elemente enthält. Mit dem `import`-Element aus XML-Schema lassen sich Teile eines WSDL-Dokuments auslagern und wiederverwenden. Dafür eignen sich vor allem WSDL-Types und WSDL-Messages.

```
<wsdl:types>
    <xs:schema xmlns:ax22="http://util.java/xsd"
        attributeFormDefault="qualified" elementFormDefault="qualified"
        targetNamespace="http://decidr.de/seminarWS">
        <xs:import namespace="http://util.java/xsd" />
        <xs:element name="setField">
            <xs:complexType>
                <xs:sequence>
                    <xs:element minOccurs="0" name="sessionID" nillable="true"
                        type="xs:string" />
                    <xs:element minOccurs="0" name="fieldName" nillable="true"
                        type="xs:string" />
                    <xs:element minOccurs="0" name="value" nillable="true"
                        type="xs:string" />
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="setFields">
            <xs:complexType>
                <xs:sequence>
                    <xs:element minOccurs="0" name="sessionID" nillable="true"
                        type="xs:string" />
                    <xs:element minOccurs="0" name="fieldMap" nillable="true"
                        type="ax22:Map" />
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:schema>
</wsdl:types>
```

Abbildung 2. WSDL-Types

```
<wsdl:portType name="TenantServicePortType">
    <wsdl:operation name="getTenantFields">
        <wsdl:input message="ns:getTenantFieldsRequest"
            wsaw:Action="urn:getTenantFields" />
        <wsdl:output message="ns:getTenantFieldsResponse"
            wsaw:Action="urn:getTenantFieldsResponse" />
        <wsdl:fault message="ns:Exception" name="Exception"
            wsaw:Action="urn:getTenantFieldsException" />
    </wsdl:operation>
    <wsdl:operation name="login">
        <wsdl:input message="ns:loginRequest" wsaw:Action="urn:login" />
        <wsdl:output message="ns:loginResponse" wsaw:Action="urn:loginResponse" />
        <wsdl:fault message="ns:Exception" name="Exception"
            wsaw:Action="urn:loginException" />
    </wsdl:operation>
    <wsdl:operation name="getField">
        <wsdl:input message="ns:getFieldRequest" wsaw:Action="urn:getField" />
        <wsdl:output message="ns:getFieldResponse" wsaw:Action="urn:getFieldResponse" />
        <wsdl:fault message="ns:Exception" name="Exception"
            wsaw:Action="urn:getFieldException" />
    </wsdl:operation>
    <wsdl:operation name="addTenant">
        <wsdl:input message="ns:addTenantRequest" wsaw:Action="urn:addTenant" />
    </wsdl:operation>
    <wsdl:operation name="setField">
        <wsdl:input message="ns:setFieldRequest" wsaw:Action="urn:setField" />
    </wsdl:operation>
</wsdl:portType>
```

Abbildung 3. WSDL-portType-Element

```
<wsdl:service name="TenantService">
    <wsdl:port name="TenantServiceHttpSoap11Endpoint"
        binding="ns:TenantServiceSoap11Binding">
        <soap:address
            location="http://localhost/TenantService.HttpSoap11Endpoint/" />
    </wsdl:port>
    <wsdl:port name="TenantServiceHttpSoap12Endpoint"
        binding="ns:TenantServiceSoap12Binding">
        <soap12:address
            location="http://localhost/TenantService.HttpSoap12Endpoint/" />
    </wsdl:port>
    <wsdl:port name="TenantServiceHttpEndpoint"
        binding="ns:TenantServiceHttpBinding">
        <http:address
            location="http://localhost/TenantService.HttpEndpoint/" />
    </wsdl:port>
</wsdl:service>
```

Abbildung 4. WSDL-Service-Element

```
<wsdl:binding name="TenantServiceSoap12Binding" type="ns:TenantServicePortType">
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"
        style="document" />
    <wsdl:operation name="logout">
        <soap12:operation soapAction="urn:logout" style="document" />
        <wsdl:input>
            <soap12:body use="literal" />
        </wsdl:input>
    </wsdl:operation>
    <wsdl:operation name="getTenantFields">
        <soap12:operation soapAction="urn:getTenantFields"
            style="document" />
        <wsdl:input>
            <soap12:body use="literal" />
        </wsdl:input>
        <wsdl:output>
            <soap12:body use="literal" />
        </wsdl:output>
        <wsdl:fault name="Exception">
            <soap12:fault use="literal" name="Exception" />
        </wsdl:fault>
    </wsdl:operation>
</wsdl:binding>
```

Abbildung 5. WSDL-Binding-Element

```
<wsdl:message name="loginResponse">
    <wsdl:part name="parameters" element="ns:loginResponse" />
</wsdl:message>
<wsdl:message name="getFieldRequest">
    <wsdl:part name="parameters" element="ns:getField" />
</wsdl:message>
<wsdl:message name="getFieldResponse">
    <wsdl:part name="parameters" element="ns:getFieldResponse" />
</wsdl:message>
<wsdl:message name="addTenantRequest">
    <wsdl:part name="parameters" element="ns:addTenant" />
</wsdl:message>
<wsdl:message name="setFieldRequest">
    <wsdl:part name="parameters" element="ns:setField" />
</wsdl:message>
```

Abbildung 6. WSDL-Message-Elemente

3.3 UDDI

UDDI[12] (Universal Description Discovery and Integration) ist ein Dienst zum Auffinden von Web Services in heterogenen Netzwerken. Er hält die WSDL-Beschreibungen und einige Metadaten registrierter Web Services bereit. Wegen verschiedener Probleme dieses Standards (die den Rahmen dieser Arbeit sprengen) hat er mittlerweile kaum noch Relevanz in der Industrie.

4 Java und Web Services

Um Web Services in Java zu implementieren braucht man vor allem drei Standards: JAX-WS 2.0 (JSR-224), WS-Metadata 2.0 (JSR-181) und JAXB 2.0 (JSR-222). Diese Standards sind in der Java Enterprise Edition 5 enthalten. Somit muss man in dieser Umgebung keine zusätzlichen Bibliotheken einbinden. Eine solche JEE-Umgebung stellt Methoden zum veröffentlichen von Web Services bereit. Deshalb enthält ein Web Service typischerweise Anwendungslogik, aber keinen Code, der direkt mit dem Netzwerk kommuniziert. Das Übersetzen von Java-Objekten in XML-Konstrukte (unmarshalling) und vice versa (marshalling) übernimmt die Laufzeitumgebung, sofern man dies nicht explizit selbst übernimmt.

4.1 JAX-WS 2.0 (JSR-224)

Bei JAX-WS[13] (Java API for XML-Based Web Services) handelt es sich um den Nachfolger von JAX-RPC (JSR-101). JAX-RPC war für RPC-orientierte Web Services konzipiert, JAX-WS für message-oriented Web Services. JAX-WS enthält alles, was man zum schreiben eines einfachen Web Service braucht – aber keine Methoden um arbiträre Java-Klassen auf XML-Schema zu mappen. Dafür ist JAXB vorgesehen. Außerdem kann man mit JAX-WS kaum Metadaten fürs Deployment in die Java-Klassen einbringen. Hier ist die Verwendung von WS-Metadata vorgesehen.

JAX-WS definiert unter anderem Standard WSDL 1.1 $\Leftrightarrow$ Java Mappings, Standard SOAP- und HTTP-Bindings, ein Standard Handler-Framework und vor allem die Client-, Server und Core-APIs für JAX-WS-konforme Web Service-Implementierungen. Allerdings sind diese Standard-Mappings und Standard-Bindings sehr minimalistisch gehalten um Redundanz und Überschneidungen mit JAXB und WS-Metadata zu vermeiden.

@WebServiceProvider
Diese Annotation schließt die Verwendung von `@WebService` aus und kann nur auf Klassen angewendet werden, die `javax.xml.ws.Provider` implementieren. Sie erlaubt den direkten Zugriff auf die empfangene XML-Nachricht. Dadurch entfällt der für das Marshalling und Unmarshalling nötige Rechenaufwand. Allerdings muss man bei dieser Methode direkt auf dem XML-Baum arbeiten, weshalb sie sich nicht für einfache Web Services eignet. Somit ist diese Annotation nur von eingeschränkter Bedeutung für DecidR.
Anwendbar auf:

– Klassen, die `javax.xml.ws.Provider` implementieren

Parameter	Zweck	Standard
serviceName	Wird als `name`-Attribut im `wsdl:service`-Element des generierten WSDL-Dokuments genutzt.	„“
portName	Wird als `name`-Attribut im `wsdl:port`-Element des generierten WSDL-Dokuments genutzt.	„“
targetNamespace	Wird im generierten WSDL-Dokument als `targetNamespace` verwendet.	„“
wsdlLocation	Eine URL, die auf eine vordefinierte WSDL-Datei zeigt. Die URL kann relativ oder absolut sein. Falls Inkonsistenzen zwischen der Implementierung und dem WSDL-Dokument bestehen, wird zur Laufzeit darauf hingewiesen.	„“

```
@WebServiceProvider
public class Examples implements Provider<SAXSource> {

    @Override
    public SAXSource invoke(SAXSource request) {
        // TODO Auto-generated method stub
        return null;
    }

}
```

Abbildung 7. Beispiel: @WebServiceProvider

4.2 WS-Metadata 2.0 (JSR-181)

WS-Metadata[15] wurde entwickelt, um die Entwicklung von Web Services zu vereinfachen. Deshalb macht WS-Metadata bewusst Annahmen, die die Implementierung einiger weniger, selten verwendeten Arten von Web Services verhindern.[2]

WS-Metadata definiert drei Ansätze für die Entwicklung von Web Services: „Start with Java“, „Start with WSDL“ und „Start with WSDL and Java“. Der letzte Ansatz ist optional und muss von der Laufzeitumgebung nicht unterstützt werden.

Start with Java Bei diesem Ansatz wird zuerst eine Java-Klasse geschrieben, die später als Web Service veröffentlicht wird. Wenn die Klasse fertig ist, wird aus ihr mittels WS-Metadata-Annotationen ein WSDL-Dokument generiert und veröffentlicht. Das generierte WSDL-Dokument folgt standardmäßig dem in JAX-WS definierten Java-zu-XML-Mapping.

Es ist auch möglich, den Java-Quelltext aus einem vorher erstellten XML-Schema generieren zu lassen.

Start with WSDL Hier wird zuerst aus einem vorher erstellten WSDL-Dokument ein Service Endpoint Interface generiert, das dann implementiert wird. Bei der Generierung des SEIs können zusätzlich Vorlagen für den Web Service generiert werden, die der Programmierer dann mit der Anwendungslogik füllt.

Start with WSDL and Java Hier sind sowohl der Java-Quellcode als auch das WSDL-Dokument vorgegeben. Der Programmierer muss lediglich den Java-Quelltext mit den nötigen Annotationen versehen. Leider ist es nicht immer möglich, existierendem Quellcode auf diese Weise ein WSDL-Dokument zuzuordnen. In diesem Fall muss zuerst mit einem der anderen Ansätze eine Wrapper-Klasse erstellt werden, die den bestehenden Quellcode mit dem WSDL-Dokument verknüpft.

Es folgt eine Liste der wichtigsten Annotationen mit ihren wichtigsten Parametern und deren Standard-Werten. Die Beschreibungen sind gekürzt aus der JSR-181-Spezifikation[15] übernommen, weshalb teilweise weniger relevante Informationen fehlen. Genauere Details sind in der JSR-181-Spezifikation[15] zu finden.

[2] Es ist beispielsweise möglich, einen JAX-WS-konformen Web Service aus den Methoden mehrerer Java-Klassen aufzubauen. WS-Metadata nimmt jedoch an, dass jede Klasse genau einen Web Service implementiert und umgekehrt.

@WebService
Markiert Klassen als Web Services oder Java Interfaces als Service Endpoint Interfaces.
Anwendbar auf:

- Klassen
- Interfaces

Parameter	**Zweck**	**Standard**
name	Wird als `name`-Attribut im `wsdl:portType`-Element des generierten WSDL-Dokuments genutzt.	Name der Klasse/des Interface
serviceName	Wird als `name`-Attribut im `wsdl:service`-Element des generierten WSDL-Dokuments genutzt. Darf nicht in SEIs verwendet werden.	Name der Klasse + „Service“
portName	Wird als `name`-Attribut im `wsdl:port`-Element des generierten WSDL-Dokuments genutzt. Darf nicht in SEIs verwendet werden.	`@WebService.name` + „Port“
targetNamespace	Wird im generierten WSDL-Dokument als `targetNamespace` verwendet. Für Details, bei welchen Elementen dieses Attribut gesetzt wird, siehe JSR-181[15]	Implementierungsabhängig, siehe JAX-WS[13], Abschnitt 3.2
wsdlLocation	Eine URL, die auf eine vordefinierte WSDL-Datei zeigt. Die URL kann relativ oder absolut sein. Falls Inkonsistenzen zwischen der Implementierung und dem WSDL-Dokument bestehen, wird zur Laufzeit darauf hingewiesen.	nichts
endpointInterface	Gibt den Namen des SEIs an, das implementiert werden soll. Das zugehörige Java-Interface muss nicht mit `implements` als implementiert markiert werden. Es bietet sich jedoch an, da der Compiler so verifizieren kann, dass das SEI korrekt implementiert wurde. Darf nicht in SEIs verwendet werden.	nichts Wird bei Bedarf implementierungsabhängig generiert, wenn es die Zielumgebung erfordert.

```
@WebService
public class Examples implements ServiceEndpointInterface {

}
```

Abbildung 8. Beispiel: @WebService

@WebMethod
Bestimmt ob und unter welchem Namen eine Methode veröffentlicht wird.[3]
Anwendbar auf:

– Methoden

Parameter	**Zweck**	**Standard**
operationName	Wird als `name`-Attribut des `wsdl:operation`-Elements genutzt, das der annotierten Methode entspricht.	Name der annotierten Methode.
exclude	Dieser Parameter gibt an, dass die annotierte Methode *nicht* veröffentlicht wird. Wenn dieser Parameter auf `true` gesetzt wird, darf `operationName` nicht gesetzt werden.	`false`

```
@WebMethod
public void exampleMethod(){

}
```

Abbildung 9. Beispiel: @WebMethod

@Oneway
Eine mit `@Oneway` annotierte `@WebMethod` hat keine Output-Message. Das bedeutet, dass der Return-Typ `void` sein muss. Wenn eine mit `@Oneway` annotierte Methode einen Rückgabewert, definierte Exceptions oder `OUT`- bzw. `INOUT`-Parameter hat, muss spätestens zur Laufzeit ein Fehler gemeldet werden.
Anwendbar auf:

– Methoden

[3] In Axis2 1.4.1 muss `operationName` dem Namen der annotierten Methode entsprechen, da sonst beim Aufruf eine `MedthodNotFoundException` geworfen wird.

```
@WebMethod
@Oneway
public void exampleMethod(){

}
```

Abbildung 10. Beispiel: @Oneway

@WebParam
Bestimmt die Eigenschaften, mit denen ein Parameter veröffentlicht wird. Anwendbar auf:

– Parameter

Parameter	**Zweck**	**Standard**
name	Name des Parameters. Muss unter bestimmten Umständen angegeben werden.	Normalerweise `arg`*N*, wobei *N* ein Integer ≥ 0 ist.
partName	Der name des `wsdl:part`s, das den Parameter repräsentiert. Wird nur unter bestimmten Umständen benutzt.	`@WebParam.name`
targetNamespace	Der XML-Namespace des Parameters. Wird nur unter bestimmten Umständen benutzt.	Entweder der leere Namespace oder der Standard-targetNamespace.
mode	Gibt an, ob der Parameter als Eingabe, Ausgabe oder beides dient.	INOUT, wenn der Parameter von `javax.xml.ws.Holder<T>` abgeleitet ist, sonst IN.
header	Wenn der Parameter im Nachrichtenkopf abgelegt ist, `true`, andernfalls `false`.	`false`

```
@WebMethod
public void exampleMethod(@WebParam int i){

}
```

Abbildung 11. Beispiel: @WebParam

@WebResult
Bestimmt die Eigenschaften, mit denen der Return-Wert einer Methode veröffentlicht wird.
Anwendbar auf:

- Methoden

Parameter	Zweck	Standard
name	Name der Ausgabe. Muss unter bestimmten Umständen angegeben werden.	Normalerweise „return“.
partName	Der name des `wsdl:parts`, das die Ausgabe repräsentiert. Wird nur in bestimmten Umständen benutzt.	`@WebResult.name`
targetNamespace	Der XML-Namespace der Ausgabe. Wird nur in bestimmten Umständen benutzt.	Entweder der leere Namespace oder der Standard-targetNamespace.
header	Wenn die Ausgabe im Nachrichtenkopf abgelegt ist, `true`, andernfalls `false`.	`false`

```
@WebMethod
@WebResult
public void exampleMethod(){

}
```

Abbildung 12. Beispiel: @WebResult

@HandlerChain
Erlaubt es, eine Folge von Handlern für Web Services anzugeben.
Anwendbar auf:

- Klassen
- Methoden
- Felder

Parameter	Zweck	Standard
File	Referenziert eine Datei, die eine HandlerChain definiert.	nichts

```
@WebService
@HandlerChain(file="handlerChain.xml")
public class Examples implements ServiceEndpointInterface {

}
```

Abbildung 13. Beispiel: @HandlerChain

@SOAPBinding
Spezifiziert, dass der Web Service auf das SOAP-Protokoll gemappt wird. Wird als `@SOAPBinding.Style` „DOCUMENT“ verwendet, dürfen Klassen und Methoden annotiert werden, andernfalls nur Klassen. Methoden, die nicht annotiert sind, verwenden die für die Klasse definierten Werte.

Die Kommunikation zwischen Web Services wird maßgeblich von dieser Annotation geprägt. Deshalb sollte man sich detailliert Gedanken machen, welche Werte man hier verwendet. Für DecidR dürfte die Standardkombination (DOCUMENT, LITERAL, WRAPPED) ausreichen. Für eine tiefergehende Diskussion zu diesem Thema siehe [16].
Anwendbar auf:

- Klassen
- Methoden

Parameter	**Zweck**	**Standard**
Style	Der Kodierungsstil für Nachrichten von und zum Web Service. Entweder „DOCUMENT“ oder „RPC“.	DOCUMENT
Use	Der Formattierungsstil für Nachrichten von und zum Web Service. Entweder „LITERAL“ oder „ENCODED“.	LITERAL
parameterStyle	Gibt an, ob der SOAP-Body nur die Parameter enthält („BARE“), oder ob sie in ein Element eingepackt werden, das nach der Methode benannt ist („WRAPPED“).	WRAPPED

```
@WebService
@SOAPBinding
public class Examples implements ServiceEndpointInterface {

}
```

Abbildung 14. Beispiel: @SOAPBinding

4.3 JAXB 2.0 (JSR-222)

JAXB[17] definiert Methoden zum Binding von Java-Klassen auf XML-Schemas und umgekehrt. Es erlaubt die Nutzung von Klassen ohne sinnvolles Standard-Binding durch einen Web Service. Das Unmarshalling erzeugt XML-Konstrukte, die validiert werden können (z.B. mit JAXP 1.3).

Auf der Java-Seite arbeitet JAXB – ähnlich wie WS-Metadata – mit Annotationen. Für die wichtigsten Annotationen sei auf [18], Folie 68 verwiesen.

Da DecidR voraussichtlich vor allem Datentypen verwenden wird, die bereits ein brauchbares Standard-Binding aufweisen, wird JAXB in diesem Projekt eine untergeordnete Rolle spielen.

5 Zusammenfassung

Web Services sind ein großes, an Bedeutung gewinnendes Thema, das im Rahmen dieser Arbeit nur oberflächlich behandelt werden kann. Sie werden von großen Unternehmen wie IBM, Microsoft und Google eingesetzt, was sie sehr zukunftssicher macht. Zudem eignen sie sich hervorragend um eine SOA zu implementieren.

Die für das DecidR-Team wichtigste Spezifikation ist WS-Metadata 2.0 (JSR-181)[15]. Sie definiert und erläutert Annotationen, die jeder in DecidR verwendete Web Service benötigt. Da diese Spezifikation die seltene Eigenschaft hat, leicht lesbar zu sein, empfiehlt es sich, einen Blick auf sie zu werfen.

Glossar

D

deployment descriptor Ein Konstrukt, das der Laufzeitumgebung eines Web Service für das Deployment benötigte Parameter liefert.

H

Handler Modifizieren Nachrichten vor und nach der Bearbeitung durch Web Services.

M

Marshalling Das erstellen eines Java-Objektes aus einem XML-Konstrukt.

S

Service Endpoint Interface Ein Java-Interface, das Definitionen zum abstrakten WSDL-Interface enthält.

U

Unmarshalling Das erstellen eines XML-Konstrukts aus einem Java-Objekt.

Literatur

1. Mark D. Hansen: SOA: Using Java Web Services. illustrated edn. Prentice Hall International (Juli 2007)
2. wikipedia.org: Service-oriented architecture. http://en.wikipedia.org/wiki/Service-oriented_architecture (25.12.2008)
3. wikipedia.org: Web service. http://en.wikipedia.org/wiki/Web_service (26.12.2008)
4. W3C: Web Services Glossary. http://www.w3.org/TR/ws-gloss/#webservice (11.02.2004)
5. Cesare Pautasso, Olaf Zimmermann, Frank Leymann: RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. http://doi.acm.org/10.1145/1367497.1367606 (April 2008)
6. Web Services Interoperability Organization. http://www.ws-i.org/ (31.12.2008)
7. Leymann, F.: Workflow Management. http://www.iaas.uni-stuttgart.de/lehre/vorlesung/aktuell/vorlesungen/workflow/ (WS08-09)
8. wikipedia.org: SOAP (protocol). http://en.wikipedia.org/wiki/SOAP_(protocol) (26.12.2008)
9. wikipedia.org: Web Services Description Language. http://en.wikipedia.org/wiki/Web_Services_Description_Language (11.12.2008)
10. Refsnes Data: WSDL Tutorial. http://www.w3schools.com/wsdl/default.asp (01.02.2009)
11. Leymann, F.: Workflow, Objects & Web Services. http://www.iaas.uni-stuttgart.de/lehre/vorlesung/aktuell/vorlesungen/workflow/materialien/Wfm07Workflow,Objects&WebServices.pdf (WS08-09)
12. wikipedia.org: Universal Description Discovery and Integration. http://en.wikipedia.org/wiki/Universal_Description_Discovery_and_Integration (05.01.2009)
13. Jitendra Kotamraju, Sun Microsystems, Inc.: JSR 224: JavaTM API for XML-Based Web Services (JAX-WS) 2.0. http://jcp.org/en/jsr/detail?id=224 (08.05.2007)
14. Progress Software Corporation: Implementing a Provider Object. http://fusesource.com/docs/framework/2.1/jaxws/JAXWSProviderImplement.html (21.01.2009)
15. Alan Mullendore, BEA Systems: JSR 181: Web Services Metadata for the JavaTM Platform. http://jcp.org/en/jsr/detail?id=181 (27.06.2006)
16. Russell Butek: Which style of WSDL should I use? http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/ (21.01.2009)
17. Kohsuke Kawaguchi, Sun Microsystems, Inc.: JSR 222: JavaTM Architecture for XML Binding (JAXB) 2.0. http://jcp.org/en/jsr/detail?id=222 (17.07.2008)
18. Tammo van Lessen, Ralph Mietzner: Web Services Crashkurs (10.12.2008)