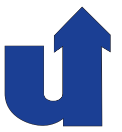
Client/Server-Programmierung

WS 2019/2020

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

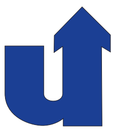
Stand: 17. Januar 2020



Client/Server-Programmierung

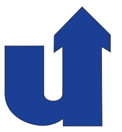
WS 2019/2020

7 Web Services



Inhalt

- Einführung
- Web-Service-Standards
  - XML, SOAP, WSDL, UDDI
- Web Services mit Axis2
- Sicherheit von Web Services

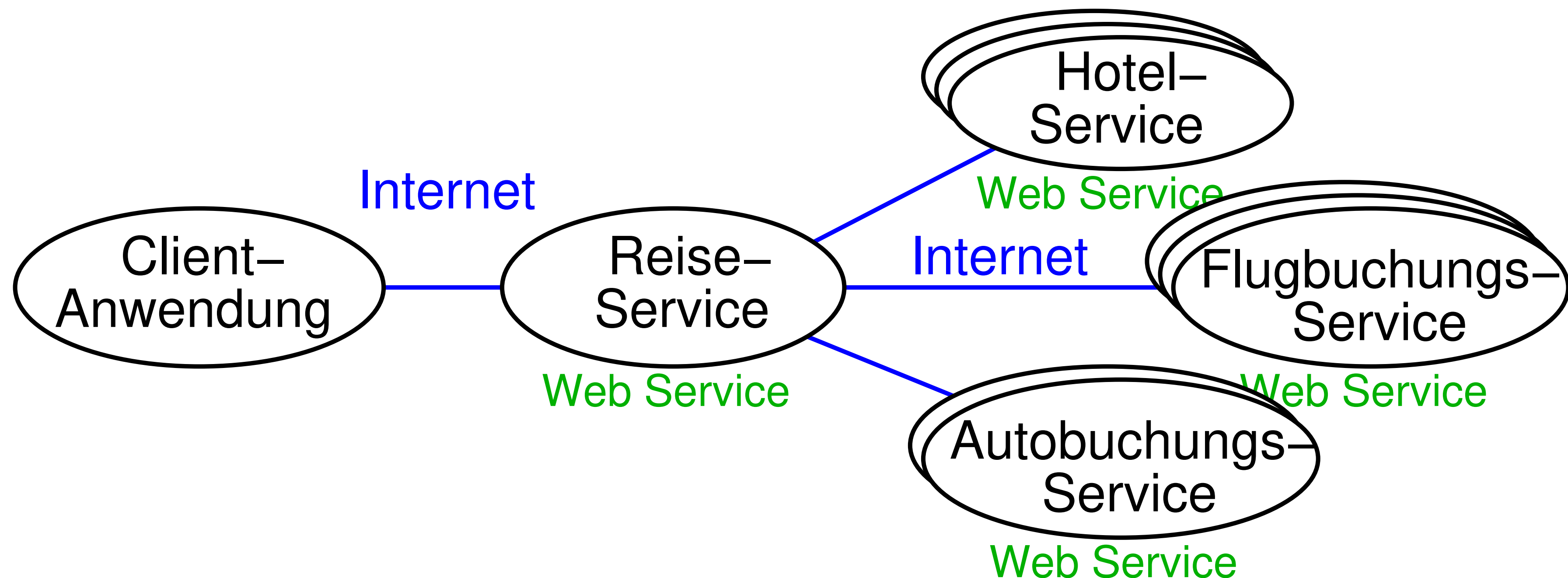


Literatur

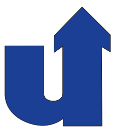
- Hammerschall, Kap. 7
- Langner: *Web Services mit Java*
- Hein, Zeller: *Java Web Services*
- A. Eberhart, S. Fischer: *Web Services*, Hanser, 2003.
- M. P. Papagoglou: *Web Servces: Principles and Technology*, Pearson, 2008.
- T. Frotscher, M. Teufel, D. Wang: *Java Web Services mit Apache Axis2*, Entwickler.press, 2007
- Zu XML auch: Horstmann / Cornell, Kap. 12

7.1 Einführung

- Im Web heute: Mensch-Maschine-Kommunikation
  - statische und dynamische Web-Seiten (Servlets, JSP, ...)
- Ziel von Web-Services: Maschine-Maschine-Kommunikation
  - Vision: Web-Services bieten komplexe Dienste an, suchen dazu eigenständig nach Teildiensten im Netz

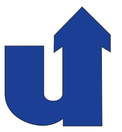


7.1 Einführung ...

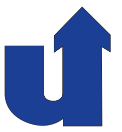


- Ziel von Web Services:
  - plattform- und sprachunabhängige Nutzung von Diensten auf entfernten Rechnern
- Modell ist dienst-orientiert, nicht objekt-orientiert
  - Web Services definieren kein verteiltes Objektmodell
  - Dienste sind als zustandslos angenommen
    - aber: Erweiterungen für Sitzungsbehandlung möglich
- Web Services kommunizieren durch den Austausch von XML-Dokumenten
- I.a. synchrone Kommunikation mit Anfrage/Antwort-Muster
  - aber auch allgemeinere Kommunikationsmuster möglich
    - Anfragen ohne Antwort, Ereignismeldungen, ...

7.1 Einführung ...

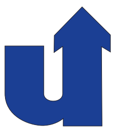


- Web Services sind zunächst nur über zwei Standards definiert:
  - das Kommunikationsprotokoll (SOAP)
  - die Schnittstellenbeschreibung (WSDL)
- Diese reichen aus, um Interoperabilität zu gewährleisten
  - vgl. IIOP und OMG-IDL bei CORBA
- Web Service Frameworks (z.B. Axis2) bieten darüberhinaus (proprietäre) Werkzeuge und APIs für z.B.
  - die Erzeugung von Client- und Server-Stubs
  - die Erzeugung / Manipulation von SOAP-Nachrichten
  - das Deployment von Web Services



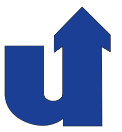
7.2 Web-Service-Standards

- Grundelemente zur Realisierung von Web Services:
  - SOAP (ursprünglich *Simple Object Access Protocol*)
    - zum Nachrichtenaustausch
    - legt i.w. Codierung von Datenstrukturen fest
  - WSDL (*Web Service Description Language*)
    - Dienst- und Schnittstellenbeschreibung
    - enthält u.a. auch „Ort“ des Dienstes (Host / Port)
  - UDDI (*Universal Description and Discovery Interface*)
    - zur Registrierung und zum Auffinden von Diensteanbietern und Diensten
- Alle drei Elemente basieren auf XML



7.2.1 XML (*Extensible Markup Language*)

- Deskriptive Sprache zur Beschreibung komplexer Datenstrukturen in einem Dokument
  - vorwiegend zum Datenaustausch zwischen Systemen
  - d.h. XML liefert ein vereinheitlichtes Datenfomat
- Technisch:
  - XML-Dokument ist ein Textdokument mit ***Tags***, die Nutzinformation umschließen (**Elemente**)
  - Elemente können verschachtelt werden
    - hierarchische Dokumenten-Struktur (mit Verweisen)
  - Syntax ähnlich wie bei HTML, jedoch strikter
    - aber: bei XML legen Tags die Bedeutung der Information fest, bei HTML die Formatierung



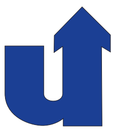
Aufbau eines XML-Dokuments

- Ein XML-Dokument besteht aus zwei Teilen:
  - **Kopf** (*Header*): XML Version, Zeichensatz, ggf. Angabe des Dokumententyps (DTD bzw. XML-Schema, s. Folie 349ff)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE addressbook SYSTEM ¨addressbook.dtd¨>
```

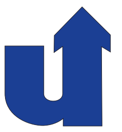
  - **Rumpf** (*Body*): verschachtelte Folge von Elementen

```
<addressbook>
  <mail-address category="professor">
    <name>Roland Wismüller</name>
    <email>roland.wismueller@uni-siegen.de</email>
  </mail-address>
</addressbook>
```



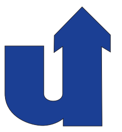
Wohlgeformte XML-Dokumente

- Ein XML-Dokument ist wohlgeformt, wenn es die syntaktischen Regeln von XML einhält, u.a.:
  - Struktur eines Elements: `<name> ... </name>`
    - Ende-Tag muß immer vorhanden sein
  - Abkürzung für leere Elemente: `<leeresElement/>`
  - korrekte Verschachtelung der Elemente in Baumstruktur
    - **nicht**: `<name>...<email> </name>...</email>`
  - XML-Dokument muß genau ein Wurzelelement besitzen
  - öffnende (und leere) Tags können auch Attribute besitzen
    - z.B. `<person name="Heinz" age="9">... </person>`
  - Kommentare: `<!-- ein Kommentar -->`



Gültige XML-Dokumente

- Nicht jedes wohlgeformte XML-Dokument beschreibt eine gültige Datenstruktur
  - der Datentyp muß ebenfalls berücksichtigt werden
  - z.B. im Beispiel von Folie 345: ein Adreßbucheintrag muß aus Name und Email-Adresse bestehen
- Dazu: XML erlaubt die Definition eines Dokumententyps bzw. Schemas
  - legt Namen, Verschachtelung, Abfolge, und möglichen Inhalt der Elemente fest
  - DTD (*Document Type Definition*): alte Methode
  - XML Schema: aktuelle Methode

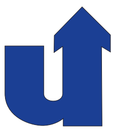


Namensräume in XML

- Um *Tag*-Namen aus verschiedenen DTDs bzw. Schematas unterscheiden zu können, führt XML Namensräume ein
- Namensraum wird durch URI (meist: URL) identifiziert
  - URI garantiert weltweite Eindeutigkeit
- Beispiel:

```
<bsp:myElem xmlns:bsp="http://www.uni-siegen.de/bsp">
  <bsp:mySubElem>Hallo</bsp:mySubElem>
</bsp:myElem>
```

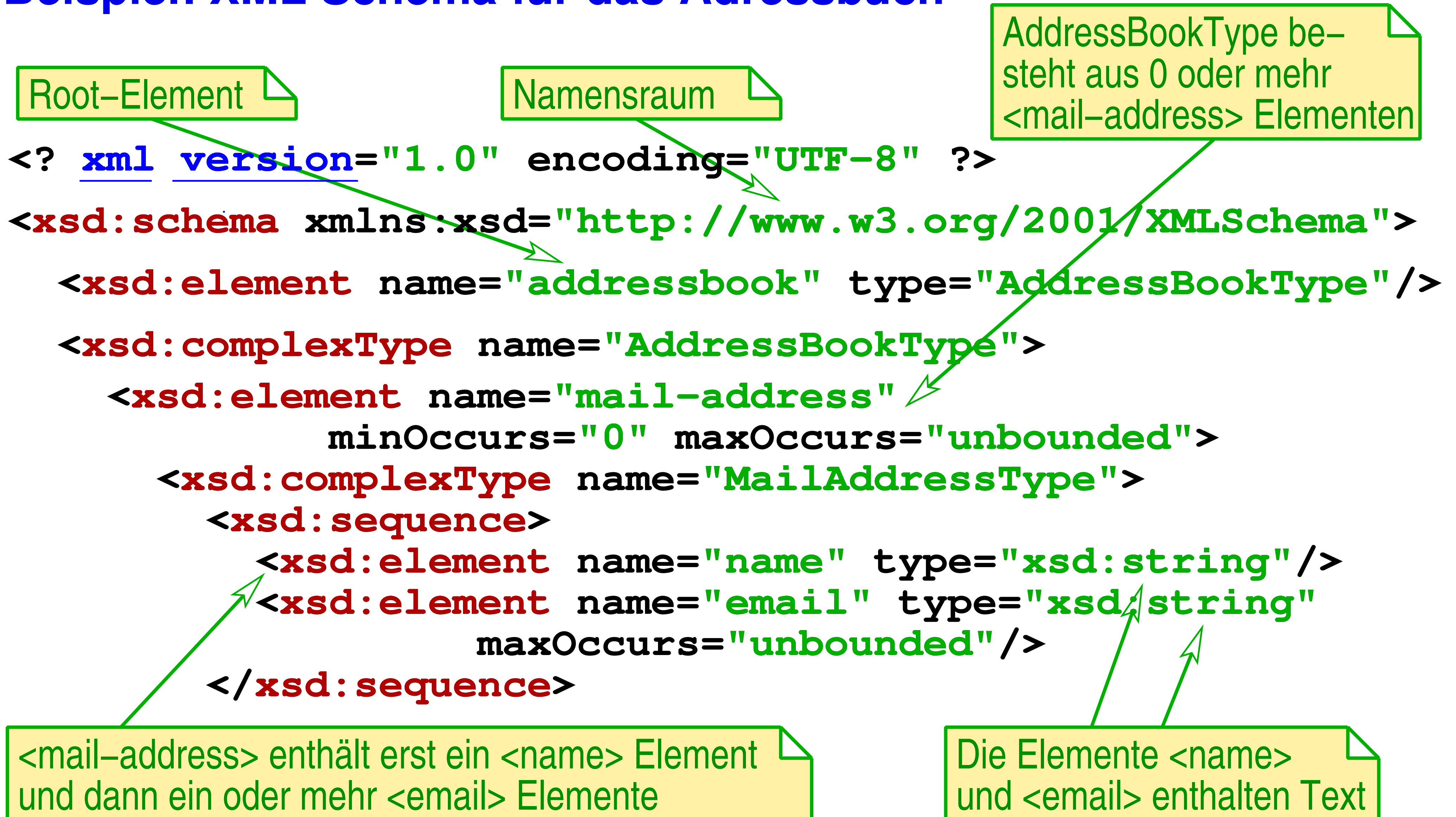
  - der String `bsp` ist ein Alias für den Namensraum
    - wird allen Elementen des Namensraums vorangestellt
- Ein Dokument kann auch mehrere Namensräume nutzen

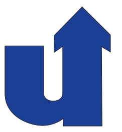


XML Schema

- Ein XML Schema ist ein XML-Dokument, das den Aufbau von XML-Dokumenten beschreibt
- Prinzip: zulässige Struktur und Inhalt eines XML-Dokuments wird als Datentyp beschrieben
- Bausteine:
  - vordefinierte, einfache Datentypen
    - ggf. mit Einschränkung von Wertebereich bzw. Syntax
  - selbst definierte, einfache Datentypen (z.B. Aufzählung)
  - komplexe (zusammengesetzte) Datentypen
    - Sequenzen (feste Folge von Unterelementen)
    - Alternativtypen (alternativ wählbare Unterelemente)
  - Kardinalitäts-Angabe (wie oft muß/darf Element auftreten?)

Beispiel: XML Schema für das Adressbuch



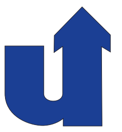


Beispiel: XML Schema für das Adressbuch ...

<mail–address> kann ein Attribut vom Typ CategoryType haben

```
        <xsd:attribute name="category" type="CategoryType"
                       use="optional"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:complexType>

  <xsd:simpleType name="CategoryType" base="xsd:string">
    <xsd:enumeration value="professor"/>
    <xsd:enumeration value="assistant"/>
  </xsd:simpleType>
</xsd:schema>
```

CategoryType ist ein String, der nur die Werte "professor" oder "assistant" haben kann



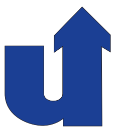
Beispiele für XML-Dokumente (nur Rumpf)

Gültig:

```
<addressbook>
  <mail-address>
    <name>Hugo</name>
    <email>hg@foo.de</email>
  </mail-address>
  <mail-address
   category="assistant">
    <name>Fritz</name>
    <email>fr@home</email>
    <email>Hallo</email>
  </mail-address>
</addressbook>
```

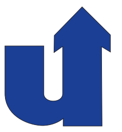
Ungültig:

```
<addressbook>
  <mail-address>
    <email>hg@foo.de</email>
    <name>Hugo</name>
  </mail-address>
  <mail-address
   category="assistant">
    <name>Fritz</name>
  </mail-address>
  <name>Hans</name>
</addressbook>
```



XML Parser

- Für XML existieren generische Parser, die zusätzlich auch die Gültigkeit des Dokuments prüfen können
  - Vorteil von XML gegenüber anderen Beschreibungssprachen
- Es gibt zwei Parsertypen:
  - SAX-Parser erzeugen für jedes auftretende XML-Primitiv ein Ereignis
    - Anwendung muß zugehörige Handler definieren
  - DOM-Parser setzen Dokument komplett in Datenstruktur um
    - DOM: *Document Object Model*
    - vorteilhaft, wenn Dokument manipuliert werden soll
    - nachteilig bei großen Dokumenten (Speicherbedarf)

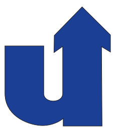


7.2.2 SOAP

- XML-basiertes Protokoll zum Austausch strukturierter Daten
- Unabhängig vom darunterliegenden Transportprotokoll
  - Bindungen für HTTP, SMTP, FTP, etc. möglich
- SOAP ist ein allgemeines Nachrichtenprotokoll
  - SOAP 1.2 definiert mehrere Nachrichtenaustausch-Muster, u.a.:
    - dialogorientierter (freier) Nachrichtenaustausch
    - Anfrage/Antwort-Muster (für RPC)
- SOAP definiert i.W. ein Nachrichtenformat
- SOAP ist erweiterbar
  - z.B. für Sicherheit (Verschlüsselung etc.)
- Aktuelle Version: 1.2 (April 2007)

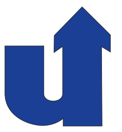
Das SOAP-Nachrichtenformat

```
<?xml version="1.0" ?>
<env:Envelope
 xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
      Header Block  (anwendungsspezifisch)
      Header Block  (anwendungsspezifisch)
  </env:Header>
  <env:Body>
      Nachrichten–Rumpf  (anwendungsspezifisch)
  </env:Body>
</env:Envelope>
```



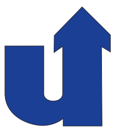
Das SOAP-Nachrichtenformat ...

- SOAP legt i.W. nur Struktur der Nachrichten fest
  - Inhalt von *Header* und *Body* sind anwendungsspezifisch
    - d.h. Schema läßt beliebige Elemente zu
  - *Header* ist optional, enthält Metadaten
    - z.B. Transaktionsnummern, Authentifizierungsdaten
  - *Body* muß vorhanden sein, enthält Anwendungsdaten
- Elemente des *Envelope* liegen in eigenem Namensraum
  - typisch: eigene Namensräume für *Header*-Blöcke und *Body*
- Hinzufügen weiterer Elemente im *Envelope* möglich



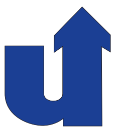
Der SOAP-*Header*

- Enthält Anweisungen, die festlegen, wie ein SOAP-Knoten die Nachricht verarbeiten soll
  - SOAP-Knoten: Sender, Empfänger und Zwischenknoten (z.B. Weiterleitung, Signierung, ...)
- Beispiele:
  - Sicherheits-*Header*
    - Zertifikat und Signatur für einige *Body*-Elemente
  - anwendungsspezifische Metadaten
    - z.B. Kundendaten des Clients
- Die Elemente des *Headers* werden von SOAP nicht festgelegt, es gibt aber vorgeschriebene Attribute der *Header*-Elemente



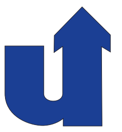
Der SOAP-*Header* ...

- Attribute von SOAP *Header*-Elementen:
  - `role`: für wen ist der Header gedacht?
    - Kodierung der Rollen durch URI (wg. Eindeutigkeit)
    - einige vordefinierte URIs, u.a. für beliebige Empfangs-knoten und endgültigen Empfänger
  - `mustUnderstand`: der Knoten, für den der *Header*-Block gedacht ist, muß ihn verstehen oder eine Fehlernachricht zurückgeben
  - `relay`: soll der *Header*-Block weitergereicht werden, falls er nicht verarbeitet wurde?
    - verarbeitete *Header*-Blöcke werden nie weitergereicht (können aber erneut eingefügt werden)



Der SOAP *Body*

- Aufbau ist abhängig von Kommunikationsstil und Codierungsstil
- Kommunikationsstil (*communication style* / *binding style*):
  - *RPC*: SOAP-Nachrichten speziell für RPCs
    - Prozeduraufruf mit Parametern bzw. Rückgabewerte
    - Aufbau des Body ist i.w. durch SOAP festgelegt
  - *document*: SOAP-Nachrichten für allgemeine Dokumente
    - für nachrichtenorientierte Kommunkation, wird aber auch für RPCs verwendet
    - Body kann beliebige XML-Daten enthalten
    - Aufbau des Body ist nur durch XML-Schema im WSDL-Dokument festgelegt

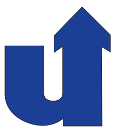


Der SOAP *Body* ...

- Codierungsstil (*encoding style* / *use*):
  - *encoded*:
    - SOAP-spezifische Typen
    - Werte in der SOAP-Nachricht haben explizite Typangabe
  - *literal*:
    - Datentypen werden durch XML-Schema beschrieben
    - bei *RPC*-Stil nur Parameter, bei *document* gesamter *Body*
  - Definition eigener Datentypen mit XML-Schema möglich
    - z.B. auch zusammengesetzte Datenstrukturen
- Relevante Kombinationen: *RPC/encoded* und *document/literal*
  - *encoded* nicht konform zu WS-I (*Web Services-Interoperability*)

Aufbau des SOAP *Body* beim *RPC* Stil

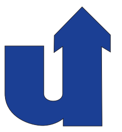
- *Body* enthält immer nur genau ein Element
- Bei Anfragenachricht:
  - Name = Name der aufzurufenden Operation
  - Kindelemente: Parameter der Operation (mit Namen)
- Bei Antwortnachricht:
  - Name ist frei wählbar (meist: Operationsname + `Response`)
  - Kindelemente:
    - Rückgabewert (meist `result`), falls Ergebnistyp $\neq$ `void`
    - Ausgabeparameter der Operation (mit Namen)
  - zum Melden von Fehlern: spezielles `<fault>`-Element
    - Unterelemente für Fehlercode und -beschreibung



Beispiel einer SOAP-Nachricht (RPC/encoded)

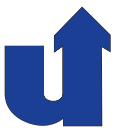
- Request-Nachricht für einen Aufruf der Java-Methode
  `String reserviere(String flugNr, int sitze, Date datum);`

```
<?xml version='1.0' encoding='UTF-8'?>
<env:Envelope
 xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <ns1:reserviere
     env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding
     xmlns:ns1="http://www.web-air.de/axis/Buchung.jws">
      <flugNr xsi:type="xsd:string">WA417</flugNr>
      <sitze xsi:type="xsd:int">3</sitze>
      <datum xsi:type="xsd:dateTime">2003-07-11</datum>
    </ns1:reserviere>
  </env:Body>
</env:Envelope>
```



Aufbau des SOAP *Body* beim *document* Stil

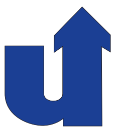
- Nach WS-I sollte *Body* nur ein Element enthalten
  - ggf. „Einwickeln“ der Argumente in ein neues Element (*wrapped* Konvention)
- Problem bei RPCs: Name der Operation geht aus SOAP-Nachricht nicht zwangsläufig hervor
- Mögliche Abhilfen:
  - bei *document/wrapped*: Elementname = Operationsname
  - Spezifikation über das Transportprotokoll
    - z.B. im HTTP-Header: `SOAPAction` (SOAP 1.1) bzw. `action` Attribut von `Content-Type` (SOAP 1.2)
  - *WS-Addressing*: Spezifikation im SOAP Header



Beispiel einer SOAP-Nachricht (document/literal/wrapped)

- Request-Nachricht für einen Aufruf der Java-Methode
  `String reserviere(String flugNr, int size, Date datum);`

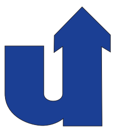
```
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope
 xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
 <soapenv:Body>
  <reserviere xmlns="http://ws.apache.org/axis2">
   <flugNr>WA417</flugNr>
   <sitze>3</sitze>
   <datum>2003-07-11</datum>
  </reserviere>
 </soapenv:Body>
</soapenv:Envelope>
```



Beispiel einer SOAP-Nachricht (document/literal/wrapped) ...

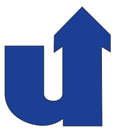
- Antwortnachricht

```
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope
 xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
 <soapenv:Body>
  <reserviereResponse xmlns="http://ws.apache.org/axis2">
   <return>GHFTR89</return>
  </reserviereResponse>
 </soapenv:Body>
</soapenv:Envelope>
```



Die SOAP-HTTP-Bindung

- SOAP-Standard definiert zwei Methoden:
  - Verwendung von HTTP POST:
    - SOAP-Anfrage-Nachricht wird im Rumpf der HTTP-Anfrage an den Server gesendet
    - Antwort als SOAP-Nachricht in der HTTP-Antwort
  - Verwendung von HTTP GET:
    - Operation und Parameter werden in URL der HTTP-Anfrage codiert (ohne SOAP)
    - Antwort als SOAP-Nachricht in der HTTP-Antwort
- Empfehlung des Standards:
  - GET-Methode (nur) verwenden bei Aufrufen ohne Seiteneffekte und ohne SOAP-Header



7.2.3 WSDL

- XML-basierte Sprache zur Beschreibung von Diensten
  - also u.a. Schnittstellen-Beschreibungssprache
  - zusätzlich aber auch: Ort (URL) des Dienstes und zu verwendendes Protokoll
- Eigenschaften von WSDL
  - unabhängig von Programmiermodell, Implementierungssprachen und Transportprotokoll
    - WSDL ist auch unabhängig von SOAP
  - Unterstützung von XML Schema zur Definition von Typen
  - WSDL ist selbst über XML Schema definiert
- Aktuelle Version: 2.0 (Juni 2007), verwendet häufig noch 1.1

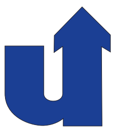
Aufbau eines WSDL-Dokuments

```
<?xml version="1.0" ?>
<wsdl:definitions
 targetNamespace="http://www.web-air.de/Buchung/"
 ...
 xmlns:wsdl="http://www.w3.org/2003/06/wsdl">
```

Wurzelelement

- Spezifikation von Datentypen
- Beschreibung der Nachrichtenformate (WSDL 1.1)
- Beschreibung der Schnittstellen
- Beschreibung der Bindungen
- Beschreibung des Dienstes

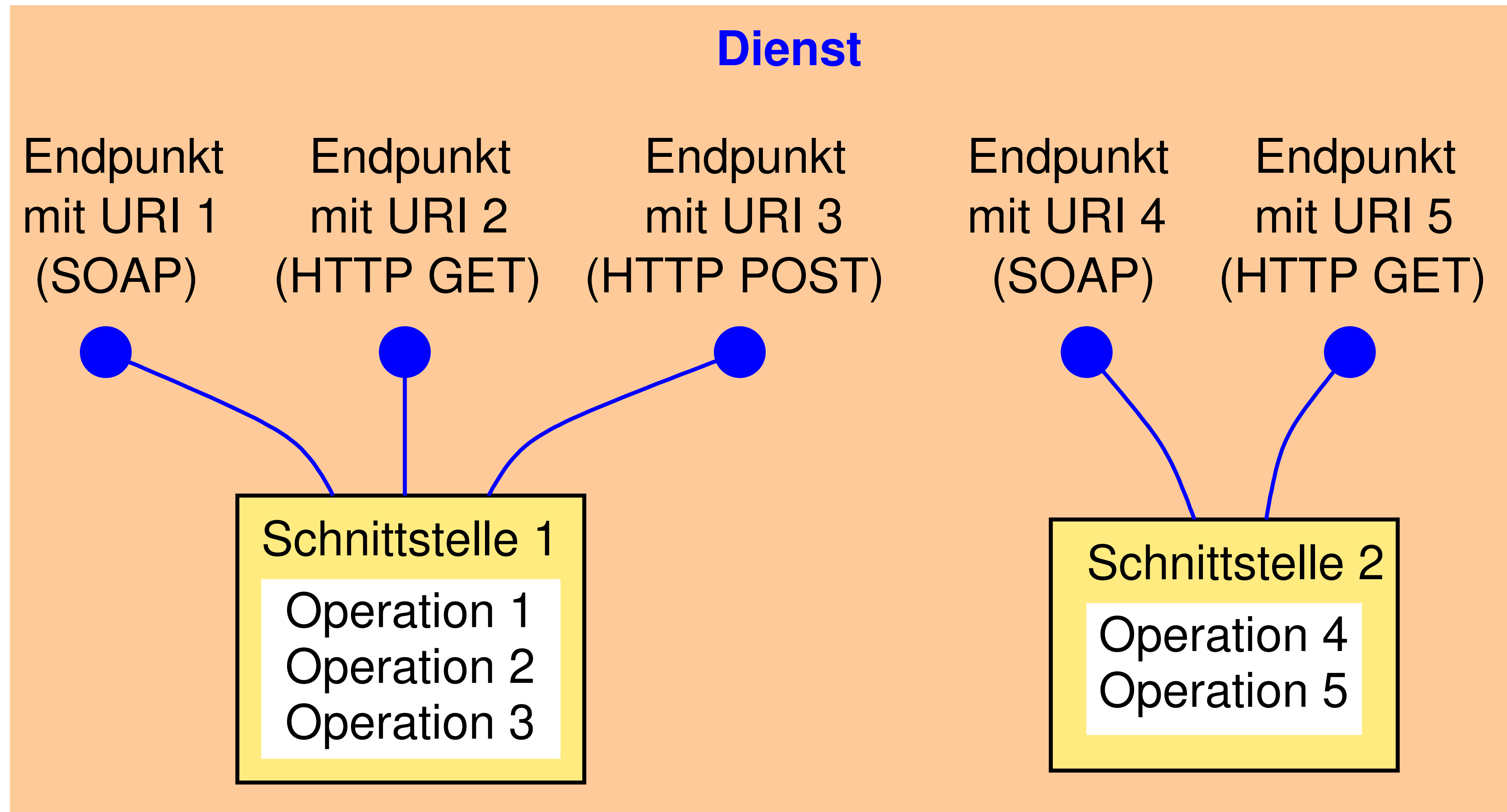
```
</wsdl:definitions>
```



Zur Struktur eines WSDL-Dokuments

- Ein WSDL-Dokument beschreibt **Dienst**e
- Ein Dienst wird über ein oder mehrere **Endpunkt**e erreicht
- Ein Endpunkt beschreibt den Ort (URI) des Dienstes und ist mit genau einer **Bindung** assoziiert
- Eine Bindung bindet die Operationen einer **Schnittstelle** an ein bestimmtes Protokoll
- Eine Schnittstelle beschreibt mehrere **Operationen**
- Für jede Operation werden die **Nachrichtenformate** für Ein- und Ausgabeparameter festgelegt (WSDL 1.1)
- Ein Nachrichtenformat besteht aus einer Sequenz von Feldern mit gegebenen **Datentypen**

Beispiel zur allgemeinen Struktur eines Dienstes



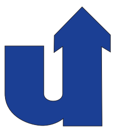
Client/Server-Programmierung

WS 2019/2020

10.01.2020

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

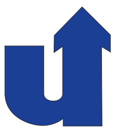
Stand: 17. Januar 2020



Spezifikation von Datentypen

- Element <types>
- Erlaubt Definition eigener komplexer Datentypen
  - Nutzung von XML Schema
- Beispiel: Datentyp für Parameter einer Flugreservierung

```
<wsdl:types>
 <xs:schema targetNamespace="http://www.web-air.de/Buchung/">
  <xs:element name="reservierung">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="flugNr" type="xs:string"/>
     <xs:element name="sitze" type="xs:int"/>
     <xs:element name="date" type="xs:date"/>
    ...
</wsdl:types>
```



Beschreibung der Nachrichtenformate (WSDL 1.1)

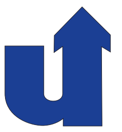
- Element <message>
- Für jede vom Dienst verwendete Nachricht (Anfrage, Antwort, ...) wird das Format definiert:
  - eine Nachricht besteht aus beliebig vielen Teilen mit festgelegtem Datentyp
    - Standard XML-Datentyp oder in <types> definiert
  - üblich: nur ein Teil mit einem `complexType` als Typ
- Beispiel (ohne Namensraumangabe für WSDL-Elemente):

```
<message name="reserviereRequest">
  <part name="parameters" type="ns:reservierung"/>
</message>
<message name="reserviereResponse">
  <part name="parameters" type="ns:reserviereResponse"/>
</message>
```

Beschreibung der Schnittstellen

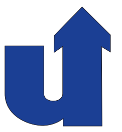
- Element <portType> (WSDL 2.0: <interface>)
- Beschreibt beliebig viele Operationen mit
  - ihren Eingabe-, Ausgabe- und ggf. Fehlernachrichtentypen
  - ihrem Interaktionsmuster
    - z.B. nur Eingabenachricht, *Request/Response*, ...
    - definiert durch Reihenfolge der Nachrichtenspezifikationen
- Beispiel (ohne Namensraumangabe für WSDL-Elemente):

```
<portType name="BuchungPortType">
  <operation name="reserviere">
    <input message="ns:reserviereRequest" />
    <output message="ns:reserviereResponse" />
  </operation>
</portType>
```



Beschreibung der Bindungen

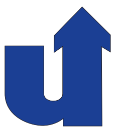
- Element <binding>
- Zu jeder Schnittstelle kann es eine oder mehrere Bindungen geben
- Die Bindung legt fest
  - das Transportprotokoll (z.B. SOAP, HTTP PUT, HTTP GET)
  - den Kommunikationsstil (`rpc` oder `document`)
  - für jede Operation:
    - URI für den Aufruf der Operation
      - wird bei HTTP in das `SOAPAction`-Element bzw. `action`-Attribut im HTTP-Header übernommen
      - wichtig bei `document`, vgl. Folie 363
    - Codierung der Ein- und Ausgabe (`encoded` oder `literal`)



Beschreibung der Bindungen ...

- Beispiel (ohne Namensraumangabe für WSDL-Elemente):

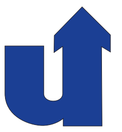
```
<binding name="BuchungSoapBinding" type="ns:BuchungPortType">
  <soap:binding transport=
   "http://schemas.xmlsoap.org/soap/http" style="document" />
  <operation name="reserviere">
    <soap:operation soapAction="urn:reserviere" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
```



Beschreibung des Dienstes

- Beschreibung besteht aus Liste von Endpunkten
  - Element <port> (WSDL 2.0: <endpoint>)
  - jeweils mit Ortsangabe (URI) und zugehöriger Bindung
- Beispiel (ohne Namensraumangabe für WSDL-Elemente):

```
<service name="Buchung">
  <port name="BuchungHttpSoapEndpoint"
   binding="s0:BuchungSoapBinding">
    <soap:address
     location="http://www.web-air.de/Buchung/Buchung.asmx" />
  </port>
</service>
```



Vollständiges Beispiel eines WSDL-Dokuments (WSDL 1.1)

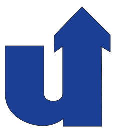
- Dienst mit zwei Operation (als Java-Interface beschrieben):

```
public interface Test {
    public String getIt();
    public void putIt(String it);
}
```

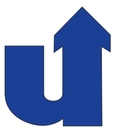
- WSDL-Dokument für den entsprechenden Web-Service:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:axis2="http://ws.apache.org/axis2"
 xmlns:ns="http://ws.apache.org/axis2/xsd"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
 xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
 targetNamespace="http://ws.apache.org/axis2">
```

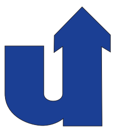
7.2.3 WSDL ...



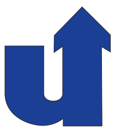
```
<wsdl:types>
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  attributeFormDefault="qualified" elementFormDefault="qualified"
  targetNamespace="http://ws.apache.org/axis2/xsd">
  <xs:element name="getItResponse">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="return" nillable="true" type="xs:string"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="putIt">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="param0" nillable="true" type="xs:string"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
 </xs:schema>
</wsdl:types>
```



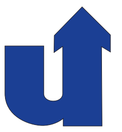
```
<wsdl:message name="getItMessage" />
<wsdl:message name="getItResponseMessage">
 <wsdl:part name="part1" element="ns:getItResponse" />
</wsdl:message>
<wsdl:message name="putItMessage">
 <wsdl:part name="part1" element="ns:putIt" />
</wsdl:message>
<wsdl:portType name="TestPortType">
 <wsdl:operation name="getIt">
  <wsdl:input xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsd
   wsaw:Action="urn:getIt" message="axis2:getItMessage"/>
  <wsdl:output xmlns:wsaw="http://www.w3.org/2006/05/addressing/ws
   message="axis2:getItResponseMessage" wsaw:Action="urn:getIt"/>
 </wsdl:operation>
 <wsdl:operation name="putIt">
  <wsdl:input xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsd
   wsaw:Action="urn:putIt" message="axis2:putItMessage"/>
 </wsdl:operation>
</wsdl:portType>
```



```
<wsdl:binding name="TestSOAP11Binding" type="axis2:TestPortType">
 <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
  style="document" />
 <wsdl:operation name="getIt">
  <soap:operation soapAction="urn:getIt" style="document" />
  <wsdl:input>
   <soap:body use="literal" />
  </wsdl:input>
  <wsdl:output>
   <soap:body use="literal" />
  </wsdl:output>
 </wsdl:operation>
 <wsdl:operation name="putIt">
  <soap:operation soapAction="urn:putIt" style="document" />
  <wsdl:input>
   <soap:body use="literal" />
  </wsdl:input>
 </wsdl:operation>
</wsdl:binding>
```

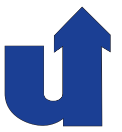


```
<wsdl:binding name="TestSOAP12Binding" type="axis2:TestPortType">
 <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"
  style="document" />
  ...
 </wsdl:operation>
</wsdl:binding>
<wsdl:service name="Test">
 <wsdl:port name="TestSOAP11port"
  binding="axis2:TestSOAP11Binding">
  <soap:address
   location="http://localhost:8080/axis2/services/Test" />
 </wsdl:port>
 <wsdl:port name="TestSOAP12port"
  binding="axis2:TestSOAP12Binding">
  <soap12:address
   location="http://localhost:8080/axis2/services/Test" />
 </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```



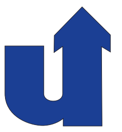
7.2.4 Nutzung von SOAP und WSDL

- Programmierer von Web Services hat i.a. nicht direkt mit SOAP und WSDL zu tun
- Web Service Frameworks enthalten Werkzeuge, die
  - *Stubs* und *Skeletons* zum Erzeugen und Interpretieren von SOAP-Nachrichten generieren
  - WSDL-Beschreibungen aus z.B. Java Interfaces erzeugen (und umgekehrt)
- Aber: Grundkenntnisse nützlich für
  - Verständnis von Web Services
  - Fehlersuche / Nutzung der Frameworks
  - spezielle Funktionen wie z.B. Sicherheit
    - benötigen Manipulation der SOAP-Nachrichten



Werkzeuge und Java-APIs für SOAP und WSDL

- Java-APIs
  - JAXM: Erstellung, Verarbeitung und Versand von SOAP-Nachrichten
    - `javax.xml.soap.*`, `javax.xml.messaging.*`
  - JAX-RPC: API und Werkzeuge für RPCs über SOAP
    - `javax.rpc.*`
    - u.a. auch Erstellung von WSDL und *Stubs*/*Skeletons*
- Frameworks für Web-Services
  - in der Vorlesung: EJB 3 (☞ **7.3**) und Axis2 (☞ **7.4**)
  - weitere z.B. IBM Web-Sphere, Microsoft Visual Studio .NET, BEA WebLogic, Borland JBuilder, ...
    - integrierte Programmentwicklungs-Umgebungen

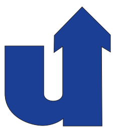


7.3 Web-Services und EJB

- Eine *stateless Session Bean* kann durch einfache Annotation zu einem Web-Service gemacht werden:

```
@Stateless
@WebService
public class HelloImpl implements HelloRemote {
  @WebMethod
  public String sayHello() {
    return "Hallo? Jemand da?";
  }
}
```

- WSDL-Datei bei OpenEJB dann unter folgender URL verfügbar:
  - http://<*host*>:<*httpejbd-port*>/HelloImpl?WSDL



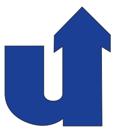
7.4 Web Services mit Axis2

- Axis2: Framework für Entwicklung und Deployment Java-basierter Web Services
- Bestandteile von Axis2
  - Web Service-Container
  - Java API
  - Werkzeuge für WSDL-Unterstützung und Deployment
- Funktionsumfang von Axis2
  - Unterstützung von SOAP 1.1/1.2, WSDL 1.1/2.0
  - Flexibles Deployment (*Hot Deployment / Hot Update*)
  - WSDL-Unterstützung (Übersetzung WSDL $\leftrightarrow$ Java)
  - Fehlerbehandlung (Java Exception $\longrightarrow$ `<fault>`-Nachricht)
  - Erweiterbarkeit durch Plug-In Module

7.4.1 Technisches zu Axis2

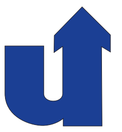
- Der Web-Service-Container von Axis2 ist als Servlet realisiert
  - meist in Verbindung mit dem Tomcat-Server
- Im Folgenden wird Kombination Axis2/Tomcat vorausgesetzt
  - entspricht Installation im Labor H-A 4111
- Verzeichnisstruktur von Axis2 (wichtig für das Deployment)

```
$CATALINA_BASE       Wurzelverzeichnis von tomcat
├── webapps
¦   ├── ROOT
¦   ├── axis2
    ¦   └── WEB-INF
    ¦       └── services   Zielverzeichnis für Deployment
```



Start des Axis2 Web-Service-Containers

- Wenn Axis2 korrekt installiert ist, muß nur der Tomcat-Server gestartet werden:
  - `$CATALINA_HOME/bin/catalina.sh run`
  - Wenn *Hot Update* in Axis2 nicht aktiviert ist, muß Tomcat bei einer Änderung deployter Web Services neu gestartet werden!
    - alternativ: *Reload* von Axis2 im Tomcat-Manager
- Funktionstest:
  - Aufruf der Axis2-Startseite im Browser
    - URL: `http://localhost:8080/axis2/`
    - enthält u.a. Links zu Validierungsseite und Liste deployter Web Services



7.4.2 Beispiel: Hello-World mit Axis2

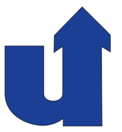
➥ Schritte zur Erstellung der WSDL-Beschreibung

1. Erstellung der Java-Schnittstelle `HelloWorld.java`
2. Übersetzen der Schnittstelle:
   ```
   javac HelloWorld.java
   ```
3. Erzeugung einer WSDL-Datei aus dieser Schnittstelle:
   ```
   java2wsdl.sh
     -l http://localhost:8080/axis2/services/HelloWorld
     -cn HelloWorld
   ```

➥ Schritte zur Erstellung des Web Services

1. Erzeugung von Implementierungsrahmen, Hilfsklassen, Service-Deskriptor und *Build*-Skript aus der WSDL-Datei:
   ```
   wsdl2java.sh -d adb -uw -ss -p hello -sd
     -uri HelloWorld.wsdl
   ```

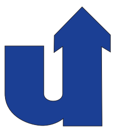
7.4.2 Beispiel: Hello-World mit Axis2 ...



- Schritte zur Erstellung des Web Services ...
  2. Editieren von `src/hello/HelloWorldSkeleton.java`
     - Implementieren der Operationen (Methoden)
  3. ggf. Anpassen des Service-Deskriptors (☞ **S. 401**)
  4. Übersetzen und Packen des Services:
     ```
     ant jar.server
     ```
  5. Kopieren des Archivs in das Axis2-Verzeichnis:
     ```
     cp build/lib/HelloWorld.aar
       $CATALINA_BASE/webapps/axis2/WEB-INF/services/
     ```

- Erster Test kann über folgende URL erfolgen:
  - `http://localhost:8080/axis2/services/HelloWorld/sayHello?args0=Roland`

7.4.2 Beispiel: Hello-World mit Axis2 ...



➥ Schritte zur Erstellung des Clients

1. Erstellung einer Java-Schnittstelle für den aufzurufenden Web Service:

```
wsdl2java.sh -uri
  http://localhost:8080/axis2/services/HelloWorld?wsdl
  -d adb -uw -p hello
```

2. Erstellung des Client-Programms

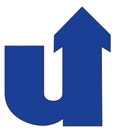
```
src/hello/HelloClient.java
```

3. Übersetzung und Packen des Client-Programms:

```
ant jar.client
```

4. Start des Client-Programms:

```
axis2.sh -cp build/lib/HelloWorld-test-client.jar
  HelloClient
```

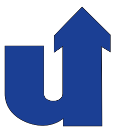


- Die Schnittstelle `HelloWorld.java`:

```
public interface HelloWorld {
    public String sayHello (String name);
}
```

- Der von Hand ergänzte Implementierungsrahmen `src/hello/HelloWorldSkeleton.java`:

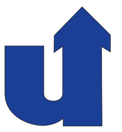
```
package hello;
public class HelloWorldSkeleton {
    int cnt = 0;
    public java.lang.String sayHello(java.lang.String args0)
    {
        return "Hello to " + args0 + " (" + ++cnt + ")";
    }
}
```



- Das Client-Programm `HelloClient.java`:

```
import hello.*;

public class HelloClient
{
    public static void main(String[] args) throws Exception
    {
        HelloWorldStub stub;

        // Referenz auf Stub holen
        stub = new HelloWorldStub();

        // Methoden des Dienstes aufrufen
        System.out.println(stub.sayHello("Roland"));
        System.out.println(stub.sayHello("Frank"));
    }
}
```

- Vollständiger Beispielcode: siehe WWW



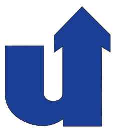
7.4.3 Details zu den Axis2-Werkzeugen

axis2.sh

- ruft `java` mit nötigem `CLASSPATH` und Java Properties auf
- nützlich zum Start von Clients

java2wsdl.sh

- erzeugt WSDL-Dokument aus Java-Schnittstelle
- wichtige Optionen:
  - `-l`: gewünschte URL des Web Services (Endpunkt)
  - `-cn`: (qualifizierter) Name der Klasse
  - `-of`: Name der Ausgabedatei

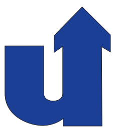


java2wsdl.sh: Beispiel

```
public interface HelloWorld {
     public String sayHello (String name);
}
```

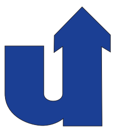
```
java2wsdl.sh -cn HelloWorld
   -l http://localhost:8080/axis2/services/Hello
```

```
...
  <wsdl:message name="sayHelloRequest">
    <wsdl:part name="parameters" element="ns:sayHello"/>
  </wsdl:message>
...
  <wsdl:portType name="HelloWorldPortType">
    <wsdl:operation name="sayHello">
...
  <wsdl:binding name="HelloWorldSoap11Binding"
                type="ns:HelloWorldPortType">
...
  <wsdl:service name="HelloWorld">
    <wsdl:port name="HelloWorldHttpSoap11Endpoint"
               binding="ns:HelloWorldSoap11Binding">
      <wsdlsoap:address
       location="http://localhost:8080/axis2/services/Hello"/>
...
```



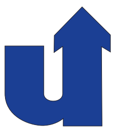
wsdl2java.sh

- erzeugt aus einem WSDL-Dokument u.a.:
  - Rahmen für die Java-Implementierungsklasse:
    `src/`*Paket*`/`*Service*`Skeleton.java`
  - Service-Deskriptor `resources/services.xml`
  - Build-Skript `build.xml` für `ant`
    - Target `jar.server` für Server-Archiv
    - Target `jar.client` für Client-Archiv
  - Client-Stub für den Web-Service:
    `src/`*Paket*`/`*Service*`Stub.java`
    - Default-Konstruktor: Endpunkt aus WSDL-Dokument
    - weiterer Konstruktor erlaubt Angabe der Endpunkt-URL als String



wsdl2java.sh ...

- wichtige Optionen:
  - `-d`: Festlegung des *Data Binding*
    - wie erfolgt die Umwandlung von XML-Dokumenten in Java-Objekte? (`adb`, `jibx`, `xmlbeans`, `jaxbri`)
  - `-uw`: Einschalten des *Unwrapping*
    - das (einzige) XML-Element des SOAP-Body wird in einzelne Parameter zerlegt
  - `-ss`: Code-Erzeugung für Serverseite
  - `-sd`: erzeuge Service-Deskriptor
  - `-p`: Festlegung des Paketnamens
  - `-uri`: Ort des WSDL-Dokuments
  - `-sn`: Auswahl eines Services aus dem WSDL-Doument
  - `-pn`: Auswahl eines Ports aus dem WSDL-Doument



Konstruktoren der Klasse *Service*`Stub`

- *Service*`Stub()`
  - erzeugt Stub, der sich mit dem Endpunkt aus der WSDL-Datei verbindet
  - Festlegung ggf. über Option `-pn` von `wsdl2java.sh`
- *Service*`Stub(String targetEndpoint)`
  - erzeugt Stub, der sich mit dem (als URL) gegebenen Endpunkt verbindet

(*Service* ist der Name der Services aus der WSDL-Datei)

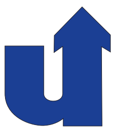


Abbildung von Datentypen von XML auf Java durch Axis2

- XML-Schema aus WSDL muß in Java-Klassen umgesetzt werden
  - prinzipiell: je eine Klasse für Anfrage- und Ergebnisnachricht
    - mit get- und set-Methoden für einzelne Komponenten
  - zusätzlich möglich: *unwrapping*
    - entfernt Wrapper-Klasse von *document/literal/wrapped*
    - Argumente können einzeln an Operation übergeben werden
- wsdl2java.sh unterstützt mehrere Möglichkeiten (*data bindings*):
  - `adb`: beste Integration mit Axis2
  - `xmlbeans`: vollständigste Unterstützung von XML-Schema
  - `jibx`: kann mit existierenden Java-Klassen arbeiten, beste Unterstützung für *unwrapping*

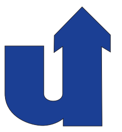
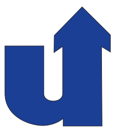


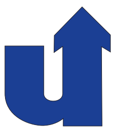
Abbildung von Datentypen von XML auf Java durch Axis2 ...

- Falls WSDL-Datei durch `java2wsdl.sh` erzeugt wird: `wsdl2java.sh` liefert auch mit *unwrapping* evtl. nicht mehr die ursprüngliche Schnittstelle
- Umsetzung ist i.a. problemlos für
  - einfache Datentypen (`int`, `double`, `boolean`, ...)
  - Arrays und Strings
- *Remote*-Referenzen auf Objekte sind mit Axis2 (und SOAP) nicht möglich!
  - im Bedarfsfall muß ggf. mit selbst implementierten Objekt-IDs gearbeitet werden
    - Server muß Abbildung von Objekt-ID auf Objekt pflegen
  - besser: *WS-Resource* Standard verwenden



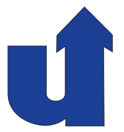
Service-Deskriptor

```
<serviceGroup>
  <service name="HelloWorld">
    <messageReceivers>
      <messageReceiver mep="http://www.w3.org/ns/wsdl/in-out"
       class="hello.HelloWorldMessageReceiverInOut"/>
    </messageReceivers>
    <parameter name="ServiceClass">hello.HelloWorldSkeleton</parame
    <parameter name="useOriginalwsdl">true</parameter>
    <parameter name="modifyUserWSDLPortAddress">true</parameter>
    <operation name="sayHello"
     mep="http://www.w3.org/ns/wsdl/in-out"
     namespace="http://ws.apache.org/axis2">
      <actionMapping>urn:sayHello</actionMapping>
      <outputActionMapping>urn:sayHelloResponse</outputActionMappin
    </operation>
  </service>
</serviceGroup>
```



Wichtige Elemente im Service-Deskriptor

- Parameter `modifyUserWSDLPortAddress`
  - falls `true`: Axis2-Container ersetzt angegebene Port-Adresse durch seine eigene
    - Probleme z.B. mit NAT und Web-Proxies
  - im Labor H-A 4111 auf `false` setzen
- Attribut `scope` des `service`-Tags
  - für Sitzungsverwaltung
  - mögliche Werte: `request`, `soapsession`, `transportsession`, `application`
  - siehe Abschnitt 7.4.4



Ablauf der Kommunikation

Client

POST /axis2/services/HelloWorld HTTP/1.1
Content−Type: application/soap+xml;
charset=UTF−8; action="urn:sayHello"
...
<?xml version="1.0" encoding="UTF−8"?>
<soapenv:Envelope ...>...<soapenv:Body>
<sayHello ...><args0>Roland</args0></sayHello>
...

HTTP/1.1 200 OK
Server: Apache−Coyote/1.1
Content−Type:application/soap+xml;action=
"urn:sayHelloResponse";charset=UTF−8
...
<?xml version="1.0" encoding="UTF−8"?>
<soapenv:Envelope ...>...<soapenv:Body>
<sayHelloResponse ...><return>Hello to
Roland</return></sayHelloResponse>
...

Server

Tomcat Web− Server

doPost()

Axis2 Servlet

sayHello()

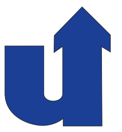
HelloWorld

<?xml version
<soapenv:Env
<sayHello ...>
...

<?xml version
<soapenv:Env
<sayHelloRes
...

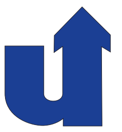
"Roland"

"Hello to Roland"



7.4.4 Sitzungs-Management mit Axis2

- Sitzungs-Management ermöglicht, in einem Web-Service einen client-spezifischen Zustand zu verwalten
- Manuell programmiertes Sitzungs-Management:
  - es gibt genau eine Instanz der Implementierungs-Klasse
  - jede Operation erhält Sitzungs-ID als zusätzlichen Parameter
    - Abbildung der Sitzungs-ID auf Sitzungsdaten z.B. über Hash-Tabelle
  - zusätzliche Methode zum Erzeugen einer Sitzungs-ID
- Sitzungs-Management mit Hilfe von Axis:
  - eine Instanz der Implementierungs-Klasse pro Sitzung
  - automatische Verfolgung der Sitzungen über generierte Sitzungs-IDs in HTTP-*Cookies* oder im SOAP-*Header*



Realisierung des Sitzungs-Managements

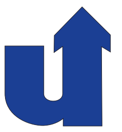
- Spezifikation des *Session Scopes* im *Deployment*-Deskriptor des Dienstes, z.B.:

```
<serviceGroup>
 <service name="HelloWorld" scope="soapsession">
 ...
```

  - Mögliche Werte:
    - `request`: keine Sitzungen, Dienst ist zustandslos
    - `transportsession`: verwende Sitzung der Transport-Ebene
    - `soapsession`: Sitzung durch SOAP realisiert
    - `application`: genau eine Instanz für alle Nachrichten

- Einschalten des Sitzungs-Managements im Client:

```
HelloWorldStub stub = new HelloWorldStub();
stub._getServiceClient().getOptions().setManageSession(true);
```



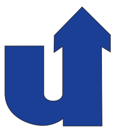
Beispiel zum Sitzungs-Management

- Implementierung des Dienstes:

```
public class HelloWorldSkeleton {
   int cnt = 1;   // Zählervariable als (sitzungslokaler) Zustand
   public String sayHello(String to) {
      return "Hello to " + to + " (" + cnt++ + ")";
   }
   // Lebenszyklus—Methoden: werden von Axis2 (über Reflection)
   // aufgerufen, wenn Session erzeugt bzw. gelöscht wird
   public void init(ServiceContext context) { ... }
   public void destroy(ServiceContext context) { ... }
}
```

- *Deployment*-Deskriptor:

```
<serviceGroup>
 <service name="HelloWorld" scope="transportsession">
 ...
```

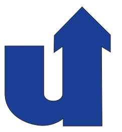


Beispiel zum Sitzungs-Management ...

- Implementierung des Clients:

```
public class HelloClient {
  public static void main(String[] args) throws Exception {
    HelloWorldStub stub = new HelloWorldStub();
    // Sitzungs-Verfolgung einschalten
    stub._getServiceClient().getOptions()
      .setManageSession(true);
    System.out.println(stub.sayHello("Roland"));
    System.out.println(stub.sayHello("Adrian"));
    System.out.println(stub.sayHello("Andreas"));
  }
}
```

- Vollständiges Beispiel im WWW

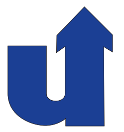


Sitzungs-Management mit HTTP-*Cookies*

- Eintrag im *Deployment*-Deskriptor:

```
<service name="HelloWorld" scope="transportsession">
```

- Axis-Server setzt bei jeder Antwort über HTTP ein *Cookie*, das der Client beim nächsten Aufruf zurückschickt
- Problem: Interoperabilität
  - HTTP als Transportprotokoll vorausgesetzt
  - Nicht-Java-Client kann evtl. *Cookie* nicht zurücksenden
    - kein direkter Zugriff auf HTTP-Anfrage
- Vorteil: Sitzungen über Dienstgruppen hinweg möglich



Ablauf der Kommunikation im Beispiel

Client

①

POST /axis2/services/HelloWorld HTTP/1.1
. . .
<?xml . . .

HTTP/1.1 200 OK
. . .
Set–Cookie: JSESSIONID=66CFAB...; Path=/axis2
. . .
<?xml . . .

②

POST /axis2/services/HelloWorld HTTP/1.1
. . .
Cookie: JSESSIONID=66CFAB...; Path=/axis2
<?xml . . .

HTTP/1.1 200 OK
. . .
<?xml . . .

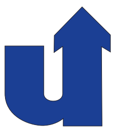
Server

Tomcat Web–Server

doPost()

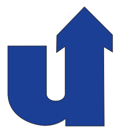
Axis2 Servlet

HelloWorld
HelloWorld
HelloWorld

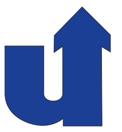


Sitzungs-Management über SOAP-*Header*

- Eintrag im *Deployment*-Deskriptor:
```
<service name="HelloWorld" scope="soapsession">
```
- Zusätzlich muß *WS-Addressing* Modul im Server und im Client aktiviert sein (Standard bei Axis2)
  - Handler modifizieren bzw. analysieren den SOAP-*Header*
  - Server fügt bei jeder Antwort *ServiceGroupId* ein
  - Client sendet diese bei Folgeanfragen zurück
- Vorteile:
  - unabhängig vom Transportprotokoll
  - Auslesen und Einfügen des SOAP-*Headers* notfalls per Hand im Client implementierbar



Ablauf der Kommunikation im Beispiel



7.4.5 Axis2 Module

- Bearbeitung von SOAP-Nachrichten in Axis2 ist über Module erweiterbar
- Modul enthält einen oder mehrere Handler
  - Handler inspizieren bzw. manipulieren SOAP-Nachrichten
  - Anwendungen z.B.
    - Sicherheit (*WS-Security*, Rampart-Modul)
      - Authentifizierung, Verschlüsselung
    - Adressierung (*WS-Addressing*)
    - Zuverlässige Kommunikation (*WS-ReliableMessaging*)
    - Debugging / Monitoring / Logging / Accounting ...
- Vorteil: Trennung von Verwaltungs-Aufgaben und Anwendungs-logik

Client/Server-Programmierung

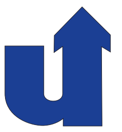
WS 2019/2020

17.01.2020

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

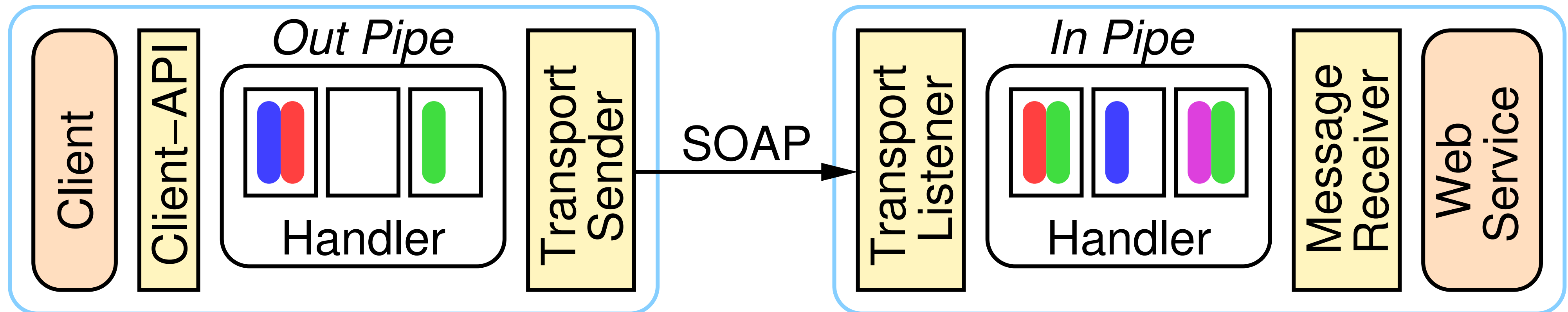
Stand: 17. Januar 2020

7.4.5 Axis2 Module ...

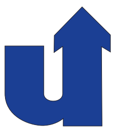


- Module können zur Laufzeit eingebunden (*engaged*) werden
  - global
  - für Dienstgruppen und einzelne Dienste
  - für einzelne Operationen
- Definition über Axis2 Konfigurationsdatei, *Deployment*-Deskriptor oder Axis2 *Web Admin*
  - `$CATALINA_BASE/webapps/axis2/WEB-INF/conf/axis2.xml`
  - `resources/services.xml`
  - `http://localhost:8080/axis2/axis2-admin/`
- Beim *Engagement* eines Moduls werden dessen Handler in definierte Phasen der Nachrichtenverarbeitung eingebunden
  - Spezifikation über Modul-Deskriptor `META_INF/module.xml`
- Phasen sind in Axis2 Konfigurationsdatei festgelegt

Nachrichtenverarbeitung in Axis2

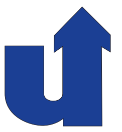


- Vier Datenflüsse: *InFlow*, *OutFlow*, *InFaultFlow*, *OutFaultFlow*
- Jeder Datenfluss wird in mehreren Phasen bearbeitet
  - z.B. *Transport*, *Pre-Dispatch*, *Dispatch*, *Message Processing* für *InFlow*
- Jede Phase besteht aus beliebig vielen Handlern
- Phasen legen Ausführungsreihenfolge der Handler fest



Beispiel: Hello World mit Logging

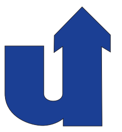
- Vollständiger Code: siehe WWW
- Dienst-Implementierung bleibt unverändert
- Aber: Einführung eines Moduls mit Handler für Anfrage- und Antwort-Nachrichten
  - `LOGHANDLER/logger/LogModule.java`
    - i.w. leere Implementierung der Axis2-Schnittstelle `Module`
  - `LOGHANDLER/logger/LogHandler.java`
    - gibt SOAP-Nachrichten auf Konsole aus
    - Parameter im *Deployment*-Deskriptor gibt Präfix an, das jeder Ausgabe vorangestellt wird



Beispiel: Hello World mit Logging ...

- Code des Handlers `LOGHANDLER/logger/LogHandler.java`:

```
public class LogHandler extends AbstractHandler {
  public InvocationResponse invoke(MessageContext context)
                                            throws AxisFault {
    // Parameter aus Deployment−Deskriptor holen
    String pre = "";
    Parameter p = getParameter("prefix");
    if (p != null)
      pre = (String)p.getValue();

    // SOAP−Nachricht ausgeben
    System.out.println(pre + context.getEnvelope());

    return InvocationResponse.CONTINUE;
  }
}
```



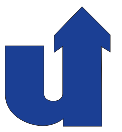
Beispiel: Hello World mit Logging ...

- *Deployment*-Deskriptor des Moduls (`LOGHANDLER/META-INF/module.xml`):

```
<module name="logger" class="logger.LogModule">
 <InFlow>
  <handler name="InFlowLogHandler" class="logger.LogHandler">
   <order phase="loggingPhase" />
   <parameter name="prefix">IN: </parameter>
  </handler>
 </InFlow>
 ...
```

- Voraussetzung: `loggingPhase` ist in Axis2 Konfiguration definiert (`$CATALINA_BASE/webapps/axis2/WEB-INF/conf/axis2.xml`):

```
<phaseOrder type="InFlow">
 <phase name="Transport"> ... </phase>
 ...
 <phase name="loggingPhase"/>
 ...
```



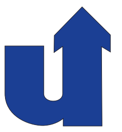
Beispiel: Hello World mit Logging ...

- *Deployment* des Moduls: Packen des Deployment-Deskriptors und der Klassen in ein Archiv `logger.mar` und Kopie nach `$CATALINA_BASE/webapps/axis2/WEB-INF/modules`
- *Engagement* des Moduls im *Deployment*-Deskriptor des Services (`SERVER/resources/services.xml`):
  - für den kompletten Dienst:

```
...
<service name="HelloWorld">
 <module ref="logger"/>
 ...
```

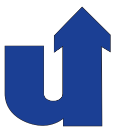
  - für eine spezifische Operation:

```
...
<operation name="sayHello" mep="http://www.w3..." ...>
 <module ref="logger"/>
 ...
```



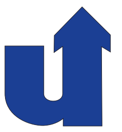
Anmerkungen zum Beispiel

- Handler erbt von `org.apache.axis2.handlers.AbstractHandler` und überschreibt `invoke(MessageContext context)`
  - Methode `getParameter()` von `AbstractHandler` erlaubt Zugriff auf die im *Deployment*-Deskriptor gesetzten Parameter
- Methoden von `org.apache.axis2.context.MessageContext`:
  - `getEnvelope()`: liefert Objekt-Repräsentation der SOAP Nachricht
  - `getProperty(String name)`: liefert Wert einer gegebenen Property aus `MessageContext`
  - `getPropertyNames()`: liefert Iterator für alle Property-Namen
  - etliche weitere Verwaltungsmethoden
    - i.w. undokumentiert ...



7.5 Web Services und Sicherheit

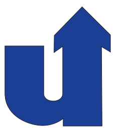
- Mögliche Maßnahmen zur Absicherung von Web Services:
  - Nutzung von HTTP über TLS/SSL
    - Verschlüsselung ($\Rightarrow$ Vertraulichkeit, Integrität)
    - Authentifizierung des Servers über Zertifikat
    - Authentifizierung des Clients (durch den Web-Server!)
      - über Zertifikat möglich
      - meist aber nur über Name und Paßwort
  - Nutzung der *WS Security* Paradigmen
    - Verschlüsselung von Teilen einer SOAP-Nachricht
    - digitale Signatur über Teile einer SOAP-Nachricht



7.5.1 Nutzung von Tomcat6 mit TLS/SSL

- Zunächst: Erstellung eines Server-Zertifikats
  - einfachste Möglichkeit: Verwendung des Java `keytools`:
    - `keytool -genkey -alias tomcat -keyalg RSA`
      - *Key* Paßwort und *Keystore* Paßwort müssen gleich sein!
    - erzeugt selbstsigniertes Zertifikat (führt zu Warnung im Web Browser)
  - i.d.R. muß das Zertifikat von einer Zertifizierungsstelle signiert werden
    - dazu: Verwendung der OpenSSL Werkzeuge und Exportieren des Zertifikats im PKCS12-Format
    - siehe Dokumentation von Tomcat und OpenSSL

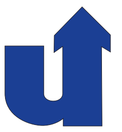
7.5.1 Nutzung von Tomcat6 mit TLS/SSL ...



- Anpassen der Datei `$CATALINA_BASE/conf/server.xml`:
  - Einfügen eines `<Connector>`-Elements für den SSL Port:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
           maxThreads="150" scheme="https" secure="true"
           clientAuth="false" sslProtocol="TLS"
           keystoreFile="/home/wismueller/.keystore"
           keystorePass="passwd1" />
```

    - Vorlage bereits als Kommentar vorhanden
  - Evtl. Löschen des `<Connector>`-Elements für den nicht-SSL Port
  - Web Services nun über `https://`*host*`:8443/axis2/...` erreichbar



7.5.2 Passwort-Authentifizierung mit Tomcat6 und Axis2

- Zugriff auf Web-Service in Axis2 einschränken (`$CATALINA_BASE/webapps/axis2/WEB-INF/web.xml`):

```
<security-constraint>
 <web-resource-collection>
  <web-resource-name>Axis services</web-resource-name>
  <url-pattern>/services/HelloWorld</url-pattern>
 </web-resource-collection>
 <auth-constraint>
  <role-name>manager</role-name>
 </auth-constraint>
</security-constraint>
<login-config>
 <auth-method>BASIC</auth-method>
 <realm-name>Axis Services</realm-name>
</login-config>
<security-role>
  <role-name>manager</role-name>
</security-role>
```

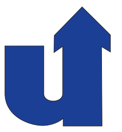
7.5.2 Passwort-Authentifizierung mit Tomcat6 und Axis ...

- Rolle und Benutzer in Tomcat konfigurieren (`$CATALINA_BASE/conf/tomcat-users.xml`):

```
<tomcat-users>
  <role rolename="manager"/>
  <user username="admin" password="s3cret" roles="manager"/>
</tomcat-users>
```

- Benutzer und Paßwort im Client übergeben:

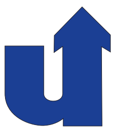
```
HelloWorldStub stub = new HelloWorldStub();

HttpTransportProperties.Authenticator auth
    = new HttpTransportProperties.Authenticator();
auth.setUsername("admin");
auth.setPassword("s3cret");
auth.setPreemptiveAuthentication(true);
stub._getServiceClient().getOptions()
    .setProperty(HTTPConstants.AUTHENTICATE, auth);
```



7.5.3 Nutzung der *WS Security* Paradigmen

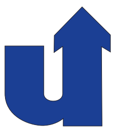
- Probleme bei Verwendung von HTTP/SSL:
  - keine Verbindlichkeit (keine digitale Signatur)
  - Verschlüsselung der gesamten SOAP-Nachricht oft unnötig
  - keine Ende-zu-Ende-Sicherheit (Verschlüsselung) bei SOAP-Nachrichten über Zwischenknoten möglch
- Lösung: Verschlüsselung / Signierung von Teilen der SOAP-Nachricht mittels *XML Encryption* bzw. *XML Signature*
- *WS Security* ist ein Standard, der u.a. festlegt, wie
  - kryptographische Information (z.B. Schlüssel, Zertifikate, digitale Signaturen)
  - Authentifizierungsinformation (z.B. Benutzername/Paßwort)

  im SOAP-*Header* abgelegt werden



Verschlüsselung von SOAP-Nachrichten

- Basis: *XML Encryption*, erlaubt die Verschlüsselung verschiedener Teile eines XML-Dokuments:
  - das komplette XML-Dokument,
  - einzelne Elemente mit ihren Unterelementen
  - oder nur der Inhalt einzelner Elemente
- Verschlüsselte Teile werden durch `<EncryptedData>`-Elemente ersetzt
  - mögliche Unterelemente u.a. für Name des Verschlüsselungsalgorithmus, Schlüsselinformation, Chiffretext
- SOAP-*Header* enthält Referenzen auf alle verschlüsselten Teile des *Body*



Beispiel für eine verschlüsselte SOAP-Nachricht

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
               wss-wssecurity-secext-1.0.xsd"
   xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
   xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
  <env:Header>
    <wsse:Security>
      <xenc:ReferenceList>
        <xenc:DataReference URI="#bodyID"/>
      </xenc:ReferenceList>
    </wsse:Security>
  </env:Header>
  <env:Body>
    <xenc:EncryptedData Id="bodyID">
      <ds:KeyInfo>
        <ds:KeyName>CN=Ali Baba, C=DE</ds:KeyName>
      </ds:KeyInfo>
      <xenc:EncryptionMethod Algorithm="..."/>
      <xenc:CipherData>
        <xenc:CipherValue>39kDeFAl...</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </env:Body>
</env:Envelope>
```

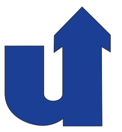
Security–Header–Block zeigt an, daß Inhalt des SOAP–Bodies verschlüsselt ist

Verschlüsselter Element–Inhalt

Schlüsselname

Verschlüsselungsalgorithmus

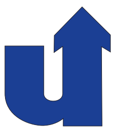
Chiffretext



Signierung von SOAP-Nachrichten

- Basis: *XML Signature*, erlaubt das (gemeinsame) Signieren verschiedener Teile eines XML-Dokuments
- Signatur-Element wird in SOAP-*Header* eingebettet und verweist auf signierte Elemente im *Body*
  - *Body* bleibt unverändert, d.h. SOAP-Knoten können digitale Signatur auch ignorieren
- Signatur-Element enthält u.a.:
  - Referenz auf die signierte Information
  - Angabe der verwendeten Algorithmen
  - Zertifikat, einschließlich des öffentlichen Schlüssels
  - die digitale Signatur selbst

7.5.3 Nutzung der *WS Security* Paradigmen ...



Beispiel für eine signierte SOAP-Nachricht

X509–Zertifikat, binär, Base64–codiert

```
<env:Envelope ...>
  <env:Header>
    <wsse:Security>
      <wsse:BinarySecurityToken ValueType="...#X509v3"
       EncodingType="...#Base64Binary" wsu:Id="X509Token">
         MIIEZzCCA9CgAwIBA...
      </wsse:BinarySecurityToken>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm="..."/>
          <ds:SignatureMethod Algorithm= ".../xmldsig#rsa-sha1"/>
          <ds:Reference URI="#myBody">
            <ds:Transforms>
              <ds:Transform Algorithm="..."/>
            </ds:Transforms>
```

Referenz auf signiertes Element

Beispiel für eine signierte SOAP-Nachricht ...

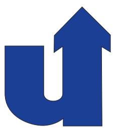
```
        <ds:DigestMethod Algorithm=".../xmldsig#sha1"/>
        <ds:DigestValue>EULddytSo1...</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>BL8jdfTPOV...</ds:SignatureValue>
    <ds:KeyInfo>
      <wsse:SecurityTokenReference>
        <wsse:Reference URI="#X509Token"/>
      </wsse:SecurityTokenReference>
    </ds:KeyInfo>
  </ds:Signature>
 </wsse:Security>
</env:Header>
<env:Body wsu:Id="myBody">
  <tru:StockSymbol xmlns:...">IBM</tru:StockSymbol>
</env:Body>
</env:Envelope>
```

Hashwert für dieses Element

Signatur für dieses Element

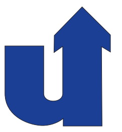
Angaben zum Signatur–schlüssel (Verweis auf obiges Zertifikat)

7.5.3 Nutzung der *WS Security* Paradigmen ...



Praktische Nutzung von *WS Security* mit Axis2

- Erweiterungs-Modul Rampart
- Definiert Axis2 *Handler*, die ein- und ausgehende SOAP-Nachrichten bearbeiten
  - Ver- bzw. Entschlüsselung von Elementen
  - Erzeugung bzw. Prüfung von Signaturen
- Rampart nutzt dazu die Bibliotheken der Apache-Projekte
  - WSS4J (*WS Security*)
  - XML Security (*XML Encryption* und *XML Signature*)



7.6 Zusammenfassung

- Grundlage von Web Services: RPC-Mechanismus
  - XML-basiertes Kommunikationsprotokoll: SOAP
  - WSDL als IDL, definiert zusätzlich Adresse des Dienstes
  - kein verteiltes Objektmodell
  - unabhängig von Plattform, Implementierungssprache und Transportprotokoll
- Axis2 als verbreitetes Web Service Framework
  - automatische Erzeugung von WSDL-Dokumenten und Client-*Stubs*
  - Axis2 Handler erlauben direkten Zugriff auf SOAP-Nachrichten
    - z.B. für Logging, Accounting, Sitzungsverwaltung, Verschlüsselung und Signierung, ...