Angewandte Softwareentwicklung
Web Services

WS 2014/2015

Markus Berg
Hochschule Wismar
Fakultät für Ingenieurwissenschaften
Bereich Elektrotechnik und Informatik

markus.berg@hs-wismar.de
http://mmberg.net

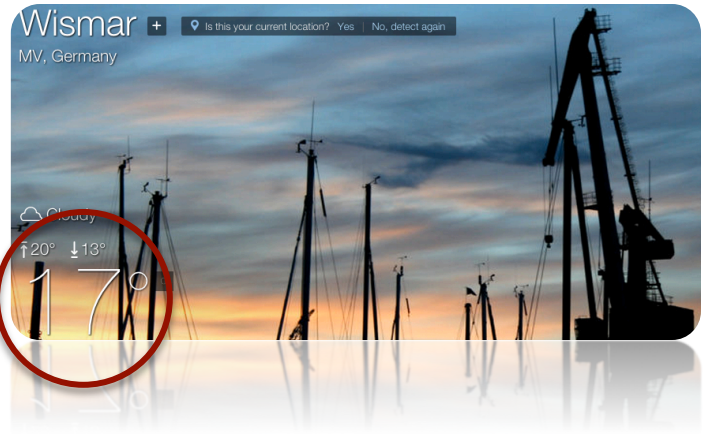
Teil I

SOAP Web Services

Motivation

- Mensch-Maschine vs. Maschine-Maschine-Kommunikation
  - Webseite: Maschine-Mensch
    - Für den Menschen leicht lesbar
    - Allerdings für Maschine schwer interpretierbar
      - Temperatur?

```
<span class="num">17</span>
```



  - Webservice: Maschine-Maschine
    - Statt den Quellcode einer Webseite zu parsen, einen Dienst aufrufen, der die Temperatur maschinenlesbar und im Sinne einer Information (also mit einer Bedeutung versehen) zurückgibt
    - Vgl. `getTemperature(„Wismar“)`

```
<div id="lead-bd" class="bd">
  <div class="cond wc-cloudy wc-icon-32">Cloudy</div>
  <div class="temp">
    <span class="c"><span class="num">17</span><span class="deg">&deg;</span></span>
    <span class="hi-lo">
      <span class="hi f w-up-arrow">68&deg;</span>
      <span class="hi c w-up-arrow">20&deg;</span>
      <span class="lo f w-down-arrow">56&deg;</span>
      <span class="lo c w-down-arrow">13&deg;</span>
    </span>
    <span class="unit">
      <span class="f">F</span>
      <span class="c">C</span>
    </span>
  </div>
```

https://weather.yahoo.com

Was ist ein Web Service?

„A Web service is a software system designed to support interoperable machine-to-machine interaction over a network.“ [W3C]

„XML Web Services bilden die grundlegenden Bausteine für die Verwirklichung des verteilten Computing im Internet.“ [Microsoft]

„Einer der Hauptvorteile der XML Web Services-Architektur besteht darin, dass Programme, die in unterschiedlichen Programmiersprachen auf verschiedenen Plattformen erstellt wurden, mit Hilfe standardisierter Verfahren miteinander kommunizieren können. “ [Microsoft]

Was ist ein Web Service? (II)

- Zur Bereitstellung von Diensten / Daten
- Dienen der Maschine-Maschine-Kommunikation (in heterogenen Systemen)
- Kommunizieren über ein Netzwerk
- Sind selbstbeschreibend
- Kommunikation über definierte Schnittstellen
  - Verbergen Details der Implementierung
- Konsument und Anbieter sind möglichst lose gekoppelt
  - D.h. deren Abhängigkeit soll gering sein
  - Werden unabhängig voneinander entwickelt (unter Beibehaltung der Schnittstelle)

Web Services (XML)

- Verschiedene Möglichkeiten der Realisierung
  - u.a. auf XML-Basis
  - D.h. Kommunikation über XML-basierte Nachrichten über eine XML-basierte Schnittstelle

> *„It has an **interface described in a machine-processable** format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an **XML serialization** in conjunction with other Web-related standards.“* [W3C]

> *„Using Web services, you can exchange loosely coupled data as XML messages between heterogeneous systems.“* [Microsoft]

XML-Web Services

- Um Daten in heterogenen Systemen (plattform-, programmiersprachen-, betriebssystemunabhängig) austauschen zu können, nutzen wir XML-Technologien
- Die Schnittstelle eines Dienstes wird über WSDL beschrieben
- Die Daten werden per SOAP beschrieben und ausgetauscht
  - Der Transport geschieht meist über HTTP (aber auch andere Protokolle wie SMTP sind möglich)
- Das Format von WSDL und SOAP selbst sowie die Definition der Nutzdaten wird über XSD beschrieben
- Oft auch „SOAP Web Services“ genannt

WSDL: Übersicht

- *Web Services Description Language*
- XML-basierte Beschreibung des Dienstes, d.h. seiner Schnittstelle
  - Operationen (Methoden) und deren Parameter
  - Protokoll und Datenformat
- Im Speziellen:
  - Types
    - Definiert die verwendeten Datentypen
    - Kann auch in externes Schema ausgelagert werden
  - Messages
    - Definiert die auszutauschenden Nachrichten
  - PortType
    - Definiert die angebotenen Operationen und deren dazugehörigen Nachrichten
  - Binding
    - Spezifiziert das Protokoll
  - Service
    - Fasst verschiedene Ports / Endpunkte zusammen und gibt eine URL an

WSDL: PortType

- Grob vergleichbar mit einer Java-Klasse (beherbergt Methoden)
- Bsp.: Ein Port namens „Weather“ hat mehrere Operationen
  - getTemperature
  - getHumidity
- PortType legt fest welche Operationen angeboten werden und welche Messages zu einer Operation gehören
  - Hierdurch wird implizit Operationstyp (one-way, request-response) festgelegt
- Bsp.: Die Operation getTemperature hat eine Input- und eine Output-Message
  - Vergleichbar mit Argumenten und Rückgabewert einer Java-Methode

```
<portType name="Weather">
  <operation name="getTemperature">
    <input message="TemperatureRequestMessage"/>
    <output message="TemperatureResponseMessage"/>
  </operation>
</portType >
```

Two-Way bzw. Request-Response Operation

WSDL: PortType (MEP)

- Message Exchange Patterns
  - One-way

```
<portType name="Weather">
  <operation name="logData">
    <input message="LogDataMessage"/>
</operation>
</portType >
```

  - Request-Response (synchron, two-way)
    - Optionale Fehlernachricht

```
<portType name="Weather">
  <operation name="getTemperature">
    <input message="TemperatureRequestMessage"/>
    <output message="TemperatureResponseMessage"/>
    <fault name="WeatherFault" message="TemperatureFaultMessage"/>
  </operation>
</portType >
```

WSDL: Message

- Eine Nachricht kann aus mehreren „Parts“ bestehen
- Oft wird jedoch nur ein Part genutzt
- Jeder Part verweist auf ein XML-Element, das die Nachricht näher beschreibt
  - Die Input-Message der Operation getTemperature verweist auf ein XML-Element namens TemperatureRequest

```
<message name="TemperatureRequestMessage">
  <part name="part1" type="w:TemperatureRequest"/>
</message>
```

WSDL: Types

- Definieren die in der Message definierten Datentypen (XML Schema)
- z.B. TemperatureRequest

```
<xsd:element name="TemperatureRequest">
  <xsd:ComplexType>
    <xsd:sequence>
      <xsd:element name="city" type="xsd:string"/>
    </xsd:sequence>
  </xsd:ComplexType>
</xsd:element>
```

- Stattdessen kann auch externe Schemadatei eingebunden werden

WSDL: Binding

- Spezifiziert wie der Port aufgerufen wird
- Binding-Type referenziert den PortType
  - Beliebiger Name verwendbar
  - mehrere Bindings können den gleichen PortType nutzen
- Soap-Binding
  - Style: Document
    - Document: Beliebiges XML-konformes Dokument
    - RPC (RPC-konformes XML)
  - SOAP Body: Literal
    - Literal: Nachricht muss XSD befolgen (validierbar)
    - Encoded: wohlgeformtes XML, jedoch kein Schema nötig (nicht validierbar)
  - Transport: SOAP over HTTP
    - vs. SMTP

WSDL: Service

- Definiert die Ports/Endpunkte unter Angabe des Bindings und gibt die URL an wo der Dienst angeboten wird
  - Ein Port dient dazu einen Dienst aufzurufen
  - Jeder Port ist mit einem Binding (z.B. HTTP) verbunden
  - Jeder Port ist mit einer Adresse verbunden
  - Der Service kann mehrere Ports besitzen (z.B. einen Port, der über HTTP, und einen, der über SMTP gebunden ist)

WSDL-Beispiel

```
▼<!--
    Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2-hudson-740-.
  -->
▼<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
  1.0.xsd" xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://currency.ase.mmberg.net/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://currency.ase.mmberg.net/" name="ConverterService">
  ▼<types>
    ▼<xsd:schema>
       <xsd:import namespace="http://currency.ase.mmberg.net/"
       schemaLocation="http://localhost:8080/CurrencyConverter/ConverterService?xsd=1"/>
     </xsd:schema>
   </types>
  ▼<message name="convert">
     <part name="parameters" element="tns:convert"/>
   </message>
  ▼<message name="convertResponse">
     <part name="parameters" element="tns:convertResponse"/>
   </message>
  ▼<portType name="ConverterService">
    ▼<operation name="convert">
       <input wsam:Action="http://currency.ase.mmberg.net/ConverterService/convertRequest"
       message="tns:convert"/>
       <output wsam:Action="http://currency.ase.mmberg.net/ConverterService/convertResponse"
       message="tns:convertResponse"/>
     </operation>
   </portType>
  ▼<binding name="ConverterServicePortBinding" type="tns:ConverterService">
     <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    ▼<operation name="convert">
       <soap:operation soapAction=""/>
      ▼<input>
         <soap:body use="literal"/>
       </input>
      ▼<output>
         <soap:body use="literal"/>
       </output>
     </operation>
   </binding>
  ▼<service name="ConverterService">
    ▼<port name="ConverterServicePort" binding="tns:ConverterServicePortBinding">
       <soap:address location="http://localhost:8080/CurrencyConverter/ConverterService"/>
     </port>
   </service>
</definitions>
```

Types: Verwendete Datentypen (XSD)

↑

Messages: Nachrichten (quasi Parameter und Rückgabewerte); nutzen die Typen

↑

PortType: Definition der Operationen (d.h. Methoden) unter Angabe der verwendeten Nachrichten

↑

Binding: Definiert die Art des Aufrufs (z.B. HTTP) für einen PortType

↑

Service & Port : Definiert die Ports (Endpunkte) über eine URL und ein bestimmtes Binding und fasst diese als Service zusammen

WSDL: Ziel bzw. Nutzen

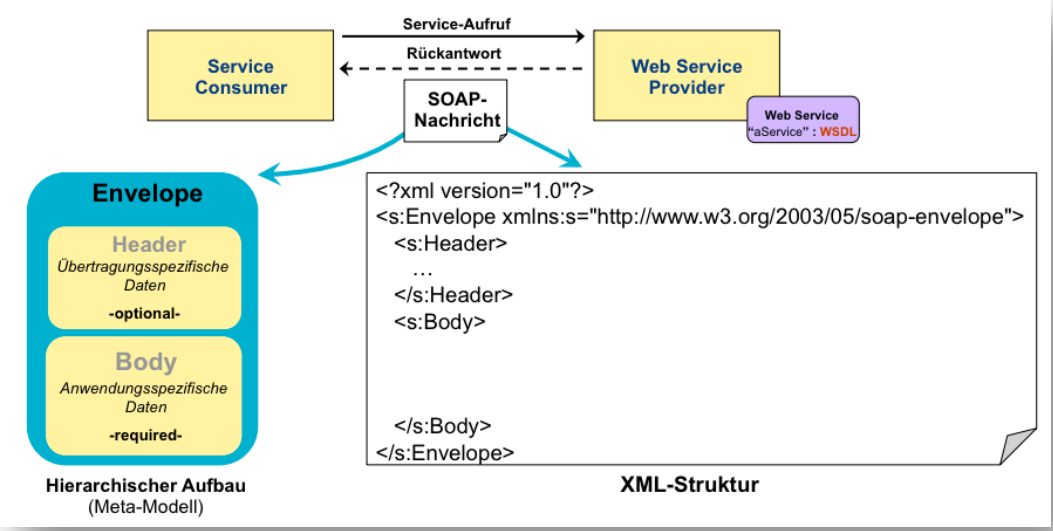
- Automatisches Generieren von Clients
  - Unabhängig von Programmiersprache
  - Wird ermöglicht durch Beschreibung der Schnittstelle
    - WSDL-first (vgl. Schema-first)

SOAP

- Ehemaliger Name: Simple Object Access Protocol
  - Heute ist SOAP kein Akronym mehr
- Kommunikationsprotokoll bzw. Nachrichtenformat für XML-Webservices
  - Daher oft auch SOAP Webservices genannt
  - Ebenfalls XML-basiert
- Definiert das Format (den Aufbau) der XML-Nachrichten die Webservices austauschen
- SOAP-Nachrichten werden meist über HTTP übertragen
  - Somit meist nicht durch Firewalls geblockt

SOAP: Aufbau

- Normales XML
- Namespace: `xmlns:soap="http://www.w3.org/2001/12/soap-envelope"`
- SOAP-Envelope:
  - Root-Element einer jeden SOAP-Nachricht
  - umschließt die folgenden Elemente:
    - Header (optional)
    - Body
      - Fault (optional)



http://www.sascha-alda.de/lehre/soa/vorlesung/Kapitel%204%20-%20Web%20Services%20I.pdf

SOAP: Header

- Allgemeine Informationen (die nicht nur für den Endpunkt sondern evtl. auch für Transportknoten gedacht sind)
- Folgende Attribute sind möglich:
  - mustUnderstand
    - Gibt an, ob ein Headerelement optional oder verpflichtend ist, d.h. ob es vom Empfänger verstanden und verarbeitet werden muss
    - Nimmt die Werte 0 oder 1 an
    - Oftmals für SecurityHeader verwendet, z.B. um sicherzustellen, dass der Empfänger nur über SSL arbeitet
  - Actor
    - Gibt Empfänger für verschiedene Teile des Headers an (falls die Nachricht über mehrere Knoten/Intermediäre vermittelt wird)
  - encodingStyle
    - Gibt optional ein Encoding für jedes beliebige Element an

SOAP: Body

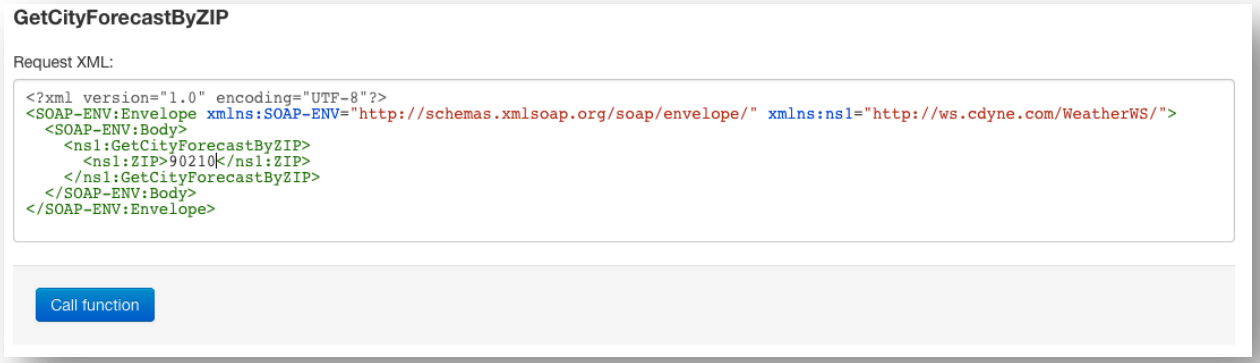
- Beinhaltet die eigentliche Nachricht
- Inhalt des Bodys ist anwendungsspezifisch
  - Muss wohlgeformtes XML sein
  - Inhalt nicht durch SOAP-Protokoll festgelegt
  - Stellt ein Request oder eine Response dar
    - Reponse kann auch ein *Fault* sein

SOAP: Fault

- Optionales Element zum Übermitteln von Fehlermeldungen
- Darf nur einmal vorkommen pro SOAP-Nachricht
- Bestandteile:
  - Faultcode
  - Faultstring
  - Faultactor (optional)
  - Detail (optional)

Testen: Online

- Wenn‘s mal schnell gehen muss…
  - Online: http://wsdlbrowser.com/
  - Eingeben einer WSDL
    - z.B. http://wsf.cdyne.com/WeatherWS/Weather.asmx?WSDL
    - Tool liest aus der WSDL die angebotenen Methoden aus
    - Bei Auswahl einer Methode wird (ebenfalls aufgrund der WSDL) die SOAP-Nachricht erzeugt
      - Die Attributwerte müssen nun ausgefüllt werden
    - Anfrage abschicken



Testen: Online (Result)

- Antwort ebenfalls als SOAP-Envelope
- Body ist anwendungsspezifisch und entspricht einem in der WSDL definierten Type
- Kein Header- und kein Fault-Element
- Ergebnis kann ausgelesen und in eigener Anwendung weiterverwendet werden

  → Maschine-Maschine-Kommunikation

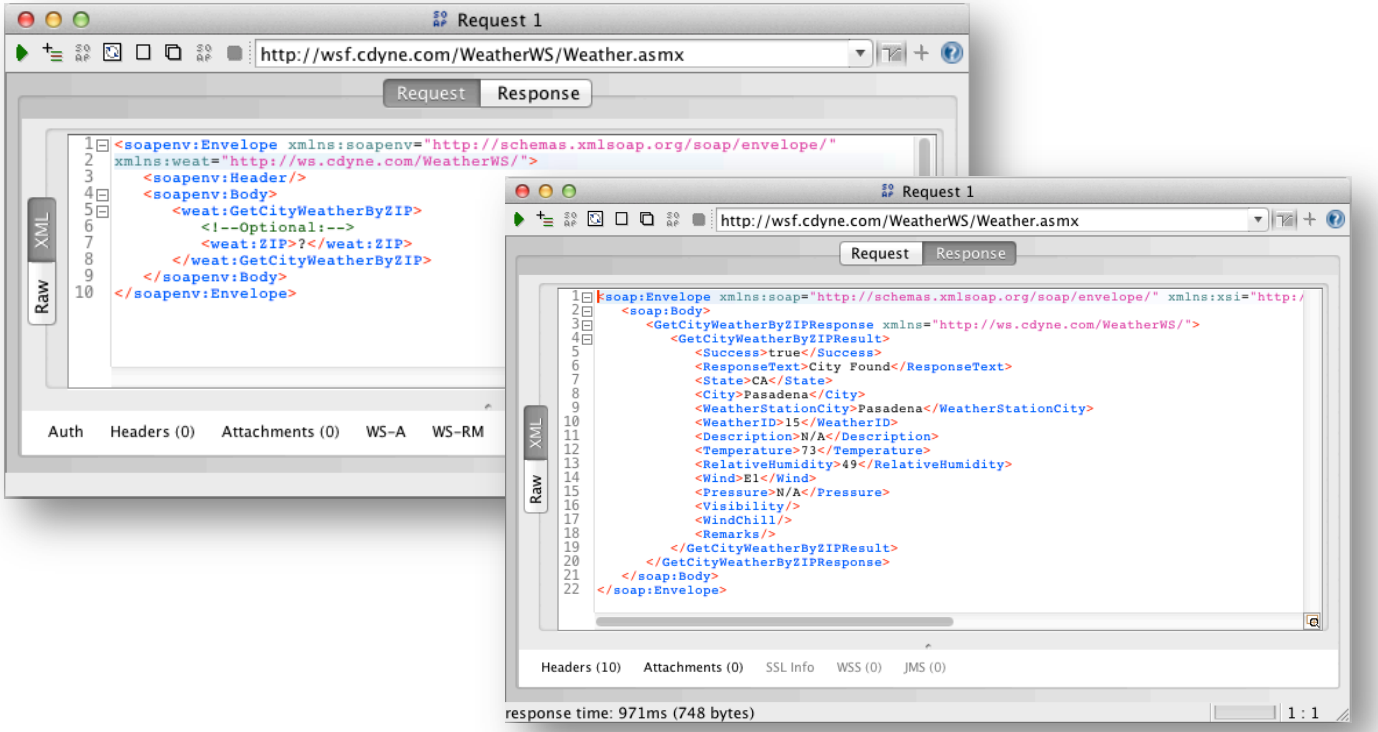
```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <soap:Body>
        <GetCityWeatherByZIPResponse
            xmlns="http://ws.cdyne.com/WeatherWS/">
            <GetCityWeatherByZIPResult>
                <Success>true</Success>
                <ResponseText>City Found</ResponseText>
                <State>CA</State>
                <City>Beverly Hills</City>
                <WeatherStationCity>Van Nuys</WeatherStationCity>
                <WeatherID>4</WeatherID>
                <Description>Sunny</Description>
                <Temperature>68</Temperature>
                <RelativeHumidity>54</RelativeHumidity>
                <Wind>CALM</Wind>
                <Pressure>29.89R</Pressure>
                <Visibility />
                <WindChill />
                <Remarks />
            </GetCityWeatherByZIPResult>
        </GetCityWeatherByZIPResponse>
    </soap:Body>
</soap:Envelope>
```

Testen mit Soap UI

- Für professionelles Testen: Soap UI
  - Kostenlos in der Standardversion
  - Geht weit über das einfache Aufrufen von Webservices hinaus
  - Echte Tests möglich
    - Testsuiten mit mehreren Testschritten und *Assertions*, die prüfen ob ein erwarteter Rückgabewert tatsächlich zurückgegeben wird
    - soll an dieser Stelle nicht behandelt werden
    - Wir nutzen Soap UI vorerst nur als Client für unsere Services und zum Analysieren der SOAP Nachrichten

Testen mit Soap UI

- Neues SOAP-Projekt anlegen & WSDL eintragen
- Soap UI erstellt Requests für jede Operation
- „?“ ersetzen durch echten Wert, z.B. 91101 für Pasadena, CA



Exkurs: HTTP

- Hyper Text Transfer Protokoll
- Grundlage der Kommunikation zwischen Webserver und Client
- Request-Arten
  - POST
  - GET
  - …
- Response-Statuscodes
  - 200 – ok
  - 201 - created
  - 404 – not found
  - 500 – server error
  - …

Exkurs: HTTP

- Jede Nachricht besteht aus
  - Header
  - Body
- Beispiel: Was passiert beim Aufruf einer Website?
  - Ausprobieren? Kommandozeilentool „curl“ (Linux/Mac)
  - Request Header

```
GET / HTTP/1.1
User-Agent: curl/7.21.4 (universal-apple-darwin11.0) libcurl/7.21.4
OpenSSL/0.9.8y zlib/1.2.5
Host: www.google.de
Accept: */*
```

  - Response Header

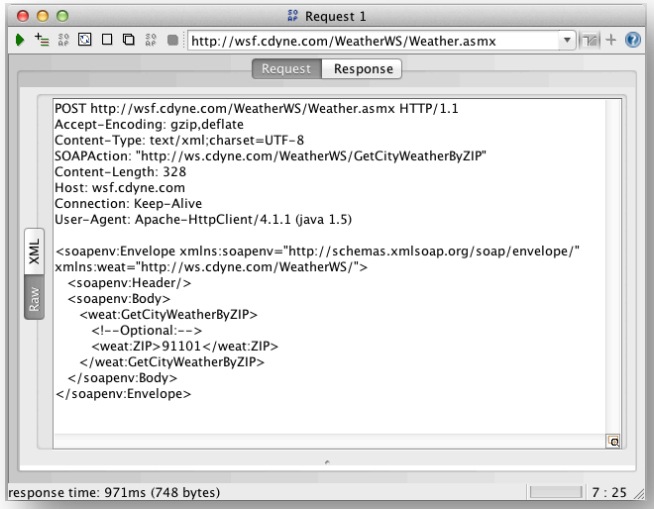
```
HTTP/1.1 200 OK
Date: Wed, 22 Oct 2014 20:29:29 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
```

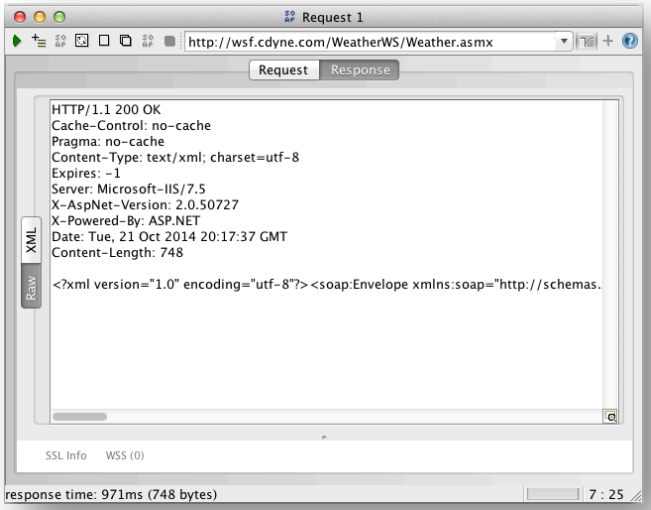
Exkurs: HTTP

- Eigentliche HTML-Seite wird im Body übertragen
- Inhalt im Body (je nach Anwendung) beliebig
- POST vs. GET
  - Verschiedene Arten der Übermittlung von Parametern
  - GET: über die URL (sichtbar, verlinkbar)
    - In der Regel, um anzugeben, was ich vom Server haben will, z.B. Suchbegriff
  - POST: als Formular (nicht in URL sichtbar, nicht verlinkbar, größere Datenmengen)
    - Um dem Server Daten zur Weiterverarbeitung zu übermitteln, z.B. Anmeldeformular, Dateiupload

SOAP Request / Response

- „SOAP over HTTP“: Der Blick unter die Haube
  - Soap UI Raw-Ansicht zeigt was gesendet und empfangen wird
  - HTTP-POST mit WS-Endpunkt als Ziel
  - Zusätzlich Variable im Header, die auf die Operation verweist (SOAPAction)
  - SOAP-Request-Message wird im Body gesendet
  - SOAP-Response-Message ebenfalls in HTTP-Response verpackt





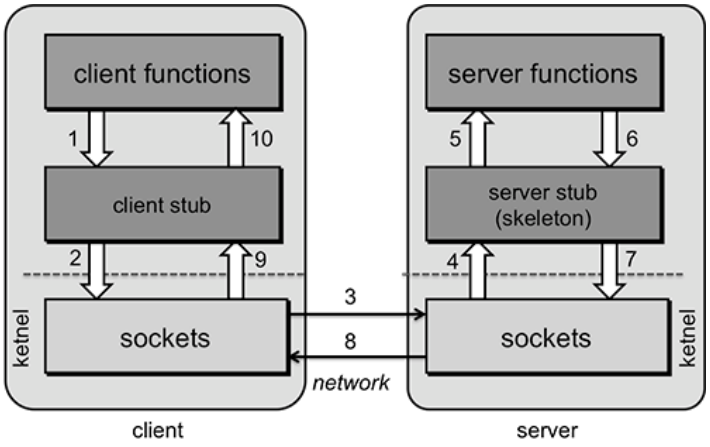
Client/Server Kommunikation

- Herkömmlicher Weg: Sockets
  - Lese-/Schreiboperationen zwischen Server und Client
  - Einfach nur ein String (ähnl. einem Lesevorgang auf der Konsole)

- Stattdessen: Remote Procedure Calls
  - Client sendet nicht nur Nutzdaten, sondern Methodenname, Attributnamen und deren Werte für eine auf dem Server existierende Methode

Kapselung der Kommunikation

- Client-Programmierung als ob es sich um einen lokalen Methodenaufruf handeln würde
  - Wir müssen nicht selbst aktiv die entsprechenden Daten senden, sondern arbeiten mit einem Proxy, der das für uns erledigt
  - Grundlage für Webservices
- Erstellen von Proxy-Klassen (Stub), die lokal die entfernten Methoden vorhalten (jedoch ohne Implementierung)
- Bei Aufruf dieser lokalen Methoden wird die Anfrage (z.B. nach SOAP) übersetzt und als Nutzlast (im Falle von RPC) Methodenname und Parameterliste übergeben
- Diese werden vom Server ausgewertet und an die entsprechenden Methoden weitergreicht



Kapselung der Kommunikation (II)

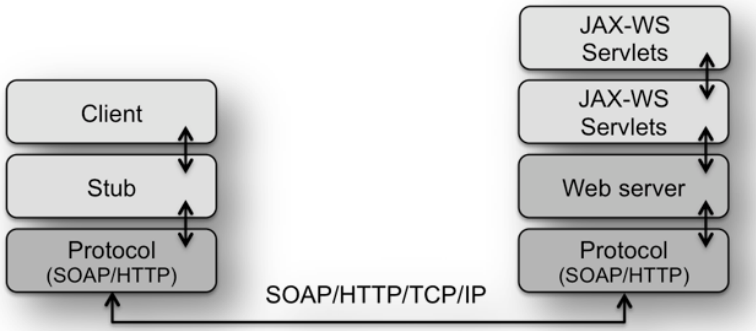
- Schnittstelle zu Funktionen des entfernten Rechners ist für den Programmierer die lokale Proxy-/Stub-Klasse
- Die Kommunikation selbst übernimmt die entsprechende Implementierung der WS API
  - Dies beinhaltet auch das Erzeugen der SOAP-Nachricht
  - Der Programmierer merkt nicht, dass er einen entfernten Dienst aufruft
- Entscheidend ist folglich die korrekte Erstellung der Proxy-Klasse, die die richtigen Methodennamen und Parameter beinhalten muss
  - Dies geschieht auf Grundlage einer abstrakten Beschreibung der Schnittstelle
  - Bei XML-Webservices: WSDL

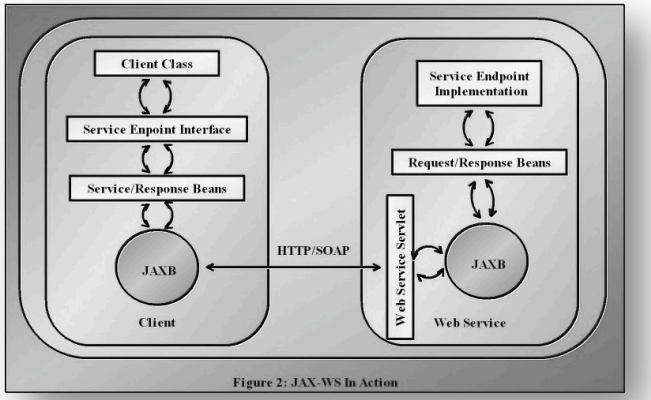
Erstellen von Clients

- Bis jetzt Testtools zum Aufrufen von Webservices genutzt
  - Diese erzeugen automatisch Clients auf Grundlage der WSDL (und einer API, s.u.)
- Erstellen von Clients für die eigene Anwendung
  - Verschiedene APIs und Implementierungen, die SOAP/WSDL entsprechend der W3C Standards WS-* umsetzen
  - Java APIs:
    - Jax-WS
    - Apache Axis 1 & 2
    - Apache CXF
  - Technologieunabhängig
    - Auch in .NET etc. möglich
  - Vereinfachen das Programmieren von Client/Server-Anwendungen, indem sie die Kommunikation kapseln
  - Kann eine lokale Desktop- oder auch eine Webanwendung sein

API: JAX WS

- API zu XML-basierten Webservices auf SOAP-Basis
- Referenzimplementierung: Metro
  - Bestandteil vom Glassfish Application Server
    - Jedoch auch auf anderen Application Servern lauffähig
  - Nutzt u.a. JAXB (zur Serialisierung der Parameter)





http://assets.devx.com/articlefigs/14978.jpg

Beispiel-Service: Währungsrechner

- Auf unserem Server gibt es einen Währungsrechner unter dem Endpunkt
  - http://luna.et.hs-wismar.de:8080/CurrencyConverter/ConverterService
  - Währungen: EUR, GBP, NAD, USD
- Zunächst sehen wir uns die WSDL an
  - Erreichbar indem wir „?wsdl“ an die URL anhängen

WSDL des Währungsservice

- Folgende Methode wird angeboten: convert (Two-Way/Request-Response)
  - Operation verweist auf Messages

```
<operation name="convert">
  <input wsam:Action="http://currency.ase.mmberg.net/ConverterService/convertRequest" message="tns:convert"/>
  <output wsam:Action="http://currency.ase.mmberg.net/ConverterService/convertResponse" message="tns:convertResponse"/>
</operation>
```

- Messages verweisen auf Elemente, die innerhalb der *Types* definiert werden (hier: Verweis auf externe XSD)

```
<message name="convert">
  <part name="parameters" element="tns:convert"/>
</message>
<message name="convertResponse">
  <part name="parameters" element="tns:convertResponse"/>
</message>
```

```
▼<types>
  ▼<xsd:schema>
     <xsd:import namespace="http://currency.ase.mmberg.net/"
     schemaLocation="http://luna.et.hs-
     wismar.de:8080/CurrencyConverter/ConverterService?xsd=1"/>
   </xsd:schema>
 </types>
```

- Das Binding ist HTTP

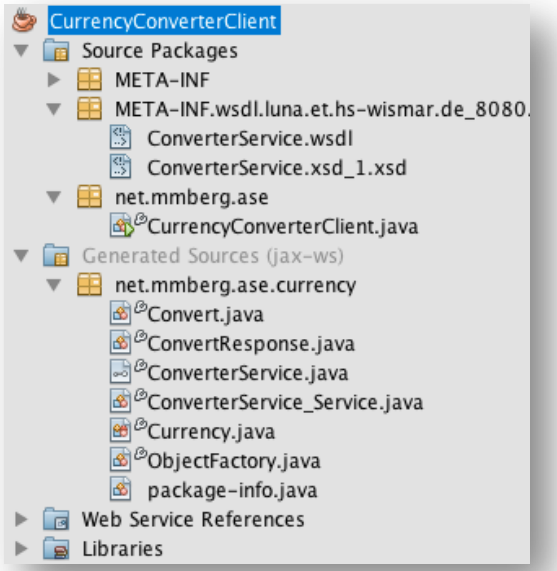
```
<binding name="ConverterServicePortBinding" type="tns:ConverterService">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
```

Beispiel: Client-Erzeugung

- Wir wollen nun einen Client bauen
  - Diese Schnittstellenbeschreibung (WSDL) ist Grundlage für den Stub
  - Netbeans nimmt uns die Erzeugung ab
  - Der Einfachheit halber: Konsolenapplikation
- Neues Projekt (Java Project)
- New > WebService Client
  - WSDL eingeben
  - Proxyklassen werden erzeugt (bilden die Funktionen des entfernten Dienstes lokal ab)

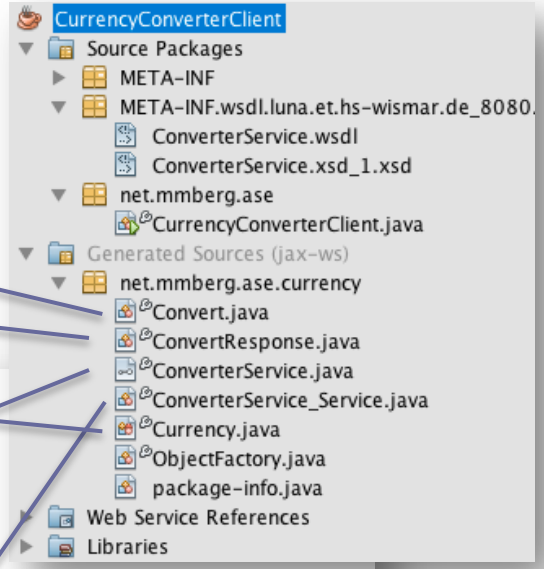
Beispiel: Proxy

- Was wurde erzeugt?
  - Lokale Kopie der WSDL und XSD
  - Proxies und Typen
    - Service
      - ConverterService_Service
    - Port
      - ConverterService
    - Messages
      - Convert
      - ConvertResponse
    - Datentypen
      - Currency
    - ObjectFactory für Serialisierung (JAX B)
      - Erzeugung der Messages als XML



```
▼<xs:schema xmlns:tns="http://currency.ase.mmberg.net/" xmlns:xs="http:/
 version="1.0" targetNamespace="http://currency.ase.mmberg.net/">
  <xs:element name="convert" type="tns:convert"/>
  <xs:element name="convertResponse" type="tns:convertResponse"/>
 ▼<xs:complexType name="convert">
  ▼<xs:sequence>
     <xs:element name="from" type="tns:currency"/>
     <xs:element name="to" type="tns:currency"/>
     <xs:element name="value" type="xs:float"/>
   </xs:sequence>
  </xs:complexType>
 ▼<xs:complexType name="convertResponse">
  ▼<xs:sequence>
     <xs:element name="return" type="xs:float"/>
   </xs:sequence>
  </xs:complexType>
 ▼<xs:simpleType name="currency">
  ▼<xs:restriction base="xs:string">
     <xs:enumeration value="EUR"/>
     <xs:enumeration value="USD"/>
     <xs:enumeration value="NAD"/>
     <xs:enumeration value="GBP"/>
   </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

XSD




```
▼<message name="convert">
   <part name="parameters" element="tns:convert"/>
 </message>
▼<message name="convertResponse">
   <part name="parameters" element="tns:convertResponse"/>
 </message>
▼<portType name="ConverterService">
  ▼<operation name="convert">
     <input wsam:Action="http://currency.ase.mmberg.net/ConverterService/convertRequest"
     message="tns:convert"/>
     <output wsam:Action="http://currency.ase.mmberg.net/ConverterService/convertResponse"
     message="tns:convertResponse"/>
   </operation>
 </portType>
▼<binding name="ConverterServicePortBinding" type="tns:ConverterService">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
  ▼<operation name="convert">
     <soap:operation soapAction=""/>
    ▼<input>
       <soap:body use="literal"/>
     </input>
    ▼<output>
       <soap:body use="literal"/>
     </output>
   </operation>
 </binding>
▼<service name="ConverterService">
  ▼<port name="ConverterServicePort" binding="tns:ConverterServicePortBinding">
     <soap:address location="http://luna.et.hs-
     wismar.de:8080/CurrencyConverter/ConverterService"/>
   </port>
 </service>
```

Auszug aus der WSDL

JAVA

```
public float convert(
        @WebParam(name = "from", targetNamespace = "")
        Currency from,
        @WebParam(name = "to", targetNamespace = "")
        Currency to,
        @WebParam(name = "value", targetNamespace = "")
        float value);
```

```
▼<xs:schema xmlns:tns="http://currency.ase.mmberg.net/" xmlns:xs="http:/
 version="1.0" targetNamespace="http://currency.ase.mmberg.net/">
   <xs:element name="convert" type="tns:convert"/>
   <xs:element name="convertResponse" type="tns:convertResponse"/>
  ▼<xs:complexType name="convert">
    ▼<xs:sequence>
       <xs:element name="from" type="tns:currency"/>
       <xs:element name="to" type="tns:currency"/>
       <xs:element name="value" type="xs:float"/>
     </xs:sequence>
   </xs:complexType>
  ▼<xs:complexType name="convertResponse">
    ▼<xs:sequence>
       <xs:element name="return" type="xs:float"/>
     </xs:sequence>
   </xs:complexType>
  ▼<xs:simpleType name="currency">
    ▼<xs:restriction base="xs:string">
       <xs:enumeration value="EUR"/>
       <xs:enumeration value="USD"/>
       <xs:enumeration value="NAD"/>
       <xs:enumeration value="GBP"/>
     </xs:restriction>
   </xs:simpleType>
 </xs:schema>
```

```
public class Convert {
    @XmlElement(required = true)
    protected Currency from;
    @XmlElement(required = true)
    protected Currency to;
    protected float value;
}
```

```
public enum Currency {EUR,USD,NAD,GBP;}
```

```
public JAXBElement<Convert> createConvert(Convert value) {
        return new JAXBElement<Convert>(_Convert_QNAME, Convert.class, null, value);
    }
```

XML

SOAP

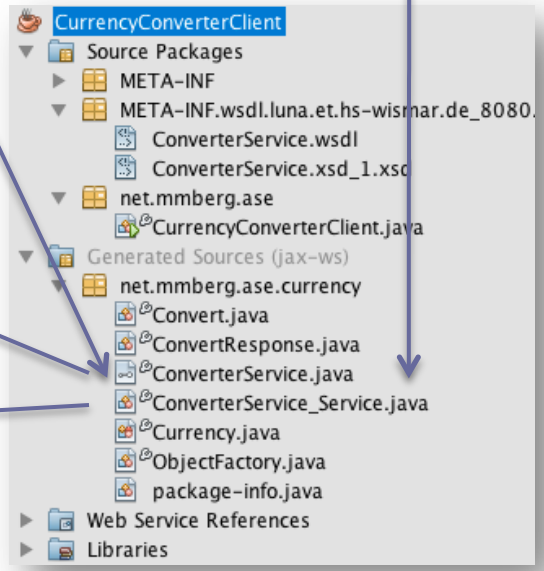
Serialisierung

Beispiel: Aufrufen des Dienstes

- Instantiieren des Proxys
- Abrufen des Ports
- Aufrufen der Operation

```
public static void main(String[] args) {
    ConverterService_Service service = new ConverterService_Service();
    ConverterService port = service.getConverterServicePort();
    float result = port.convert(Currency.EUR, Currency.USD, 12f);
    System.out.println(result);
}
```

```
▼<portType name="ConverterService">
 ▼<operation name="convert">
    <input wsam:Action="http://currency.ase.mmberg.net/ConverterService/convertRequest"
    message="tns:convert"/>
    <output wsam:Action="http://currency.ase.mmberg.net/ConverterService/convertResponse"
    message="tns:convertResponse"/>
  </operation>
 </portType>
▼<binding name="ConverterServicePortBinding" type="tns:ConverterService">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
 ▼<operation name="convert">
    <soap:operation soapAction=""/>
   ▼<input>
      <soap:body use="literal"/>
    </input>
   ▼<output>
      <soap:body use="literal"/>
    </output>
  </operation>
 </binding>
▼<service name="ConverterService">
 ▼<port name="ConverterServicePort" binding="tns:ConverterServicePortBinding">
    <soap:address location="http://luna.et.hs-
    wismar.de:8080/CurrencyConverter/ConverterService"/>
  </port>
 </service>
```



Auszug aus der WSDL

Beispiel: Aufrufen des Dienstes

- Aus Programmierersicht:

```
public static void main(String[] args) {
    ConverterService_Service service = new ConverterService_Service();
    ConverterService port = service.getConverterServicePort();
    float result = port.convert(Currency.EUR, Currency.USD, 12f);
    System.out.println(result);
}
```

```
wsimport-client-generate:
Compiling 1 source file to /Users/markus/
compile:
run:
15.20364
```

Erstellen von Services

- Verschiedene Möglichkeiten (ähnlich wie bei der Serialisierung)
  - Code-first
    - Der meist verbreitetste Weg
  - Contract-first (WSDL)
    - Seltener, aber durchaus möglich

Währungsrechner-Service

- Code-first Ansatz
- Netbeans: Neue *Web Application* (CurrencyConverter)
  - Tomcat / Java EE 6
  - ContextPath: CurrencyConverter
  - Rechte Maustaste: Neuen Web Service anlegen (ConverterService)
    - Metro benutzen? JA
  - Neue Methode `convert()`
    - Annotieren mit @WebMethod

```
@WebMethod
public float convert(String from, String to, float value){
}
```

    - Implementieren (Bsp. siehe SVN) und deployen (rechte Maustaste auf Projekt: Deploy)

Was passiert beim Deployment?

- Kompilieren und packen als Webarchiv (war)
- Server starten
- Webarchiv in Deployment-Ordner des Servers kopieren
- Auto Deployment (Server erkennt, dass ein neues Archiv im überwachten Ordner ist, entpackt die Anwendung und stellt sie bereit)
  - WSDL und XSD werden erstellt
  - Service zeigt auf die URL, wo der Dienst bereit gestellt wurde

```
INFO: Initializing ProtocolHandler ["http-bio-8080"]
Okt 25, 2014 3:43:22 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 154414 ms
INFO: Deploying configuration descriptor /Applications/apache-tomcat-7.0.52/conf/Catalina/
localhost/CurrencyConverter.xml
Okt 25, 2014 3:43:43 PM com.sun.xml.ws.transport.http.servlet.WSServletContextListener
contextInitialized
INFO: Metro monitoring rootname successfully set to:
com.sun.metro:pp=/,type=WSEndpoint,name=/CurrencyConverter-ConverterService-
ConverterServicePort
```

Exkurs: Web Server

- Web Server liefert Webseiten aus (HTML)
  - Kommuniziert per HTTP
  - Prinizipiell statischer Content
  - Dynamische Seiten durch Skriptsprachen und entsprechende Module zum Ausführen von PHP, Perl, ASP, …
    - Webserver gibt Anfrage an entsprechenden Interpreter weiter
    - Diese Skriptsprachen werden auf dem Server interpretiert und geben HTML-Code aus, der dann vom Server an den Client (Browser) übertragen wird
    - Skripte direkt in den HTML-Code integriert
  - z.B. Apache
  - Typischerweise Port 80

Exkurs: Application Server

- Application Server führt Webanwendungen aus
  - Einem Webserver sehr ähnlich, da er einen Webserver beinhaltet bzw. erweitert
  - Nicht auf HTTP beschränkt
  - Logik in Programm ausgelagert statt direkt mit dem HTML-Code verwoben
    - Der Begriff (Java) Application Server wird meist genutzt, sobald wir kompilierten Code ausführen bzw. „echtes“ Java (z.B. in Java Servlets) statt Skripte benutzen
  - Oft im Zusammenhang mit Java EE verwendet, aber existiert auch für .NET
  - Unterschied zw. Webserver mit Servlet Container und App Server noch schwieriger
    - Für uns ist ein Java Application Server ein Server der Java-Code ausführen kann
    - z.B.
      - Apache Tomcat (unterstützt nur einen Teil von Java EE, Webserver mit Servlet Container)
      - Oracle Glassfish (Java EE inkl. EJB)
      - Oracle Weblogic (Java EE inkl. EJB)
      - JBoss (Java EE inkl. EJB)
  - Liefert nicht unbedingt HTML aus, sondern z.B. auch SOAP Responses
    - Muss demnach nicht unbedingt ein grafisches Frontend haben

Deployment

- Endpunkt: http://localhost:8080/CurrencyConverter/ConverterService

Web Services

Endpoint		Information	
Service Name:	{http://currency.ase.mmberg.net/}ConverterService	Address:	http://localhost:8080/CurrencyConverter/ConverterService
Port Name:	{http://currency.ase.mmberg.net/}ConverterServicePort	WSDL:	http://localhost:8080/CurrencyConverter/ConverterService?wsdl
		Implementation class:	net.mmberg.ase.currency.ConverterService

- WSDL und XSD wurden automatisch erstellt

Automatisch erstellte WSDL

```
▼<!--
    Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2-hudson-740-.
  -->
▼<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
  1.0.xsd" xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://currency.ase.mmberg.net/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://currency.ase.mmberg.net/" name="ConverterService">
  ▼<types>
    ▼<xsd:schema>
       <xsd:import namespace="http://currency.ase.mmberg.net/"
       schemaLocation="http://localhost:8080/CurrencyConverter/ConverterService?xsd=1"/>
     </xsd:schema>
   </types>
  ▼<message name="convert">
     <part name="parameters" element="tns:convert"/>
   </message>
  ▼<message name="convertResponse">
     <part name="parameters" element="tns:convertResponse"/>
   </message>
  ▼<portType name="ConverterService">
    ▼<operation name="convert">
       <input wsam:Action="http://currency.ase.mmberg.net/ConverterService/convertRequest"
       message="tns:convert"/>
       <output wsam:Action="http://currency.ase.mmberg.net/ConverterService/convertResponse"
       message="tns:convertResponse"/>
     </operation>
   </portType>
  ▼<binding name="ConverterServicePortBinding" type="tns:ConverterService">
     <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    ▼<operation name="convert">
       <soap:operation soapAction=""/>
      ▼<input>
         <soap:body use="literal"/>
       </input>
      ▼<output>
         <soap:body use="literal"/>
       </output>
     </operation>
   </binding>
  ▼<service name="ConverterService">
    ▼<port name="ConverterServicePort" binding="tns:ConverterServicePortBinding">
       <soap:address location="http://localhost:8080/CurrencyConverter/ConverterService"/>
     </port>
   </service>
 </definitions>
```

- Kennen wir bereits aus dem WSDL-Beispiel…

Erstellte XSD

```
▼<xs:schema xmlns:tns="http://currency.ase.mmberg.net/" xmlns:xs="htt
  <xs:element name="convert" type="tns:convert"/>
  <xs:element name="convertResponse" type="tns:convertResponse"/>
 ▼<xs:complexType name="convert">
  ▼<xs:sequence>
     <xs:element name="from" type="xs:string" minOccurs="0"/>
     <xs:element name="to" type="xs:string" minOccurs="0"/>
     <xs:element name="value" type="xs:float"/>
   </xs:sequence>
  </xs:complexType>
 ▼<xs:complexType name="convertResponse">
  ▼<xs:sequence>
     <xs:element name="return" type="xs:float"/>
   </xs:sequence>
  </xs:complexType>
</xs:schema>
```

- Funktional, aber Elementnamen heißen „argN“
- umbenennen über Annotation `@WebParam(name=„NAME“)`

Testen des Dienstes

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:cur="http://currency.ase.mmberg.net/">
   <soapenv:Header/>
   <soapenv:Body>
      <cur:convert>
         <from>EUR</from>
         <to>GBP</to>
         <value>6</value>
      </cur:convert>
   </soapenv:Body>
</soapenv:Envelope>
```

```xml
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:convertResponse xmlns:ns2="http://currency.ase.mmberg.net/">
         <return>4.7250023</return>
      </ns2:convertResponse>
   </S:Body>
</S:Envelope>
```

Schema überarbeiten

- Client weiß nicht welche Währungen angeboten werden
- Für Währungen statt String einen neuen Enum-Datentyp anlegen

```
public enum CURRENCY {EUR,USD,NAD,GBP};
```

- Parameter sind teilweise optional
  - Nur primitive Java-Typen sind Pflichtfelder
  - Daher: Annotation

    `@XmlElement(required = true)`
- Resultiert in neuem Schema mit neuem Datentyp

```
▼<xs:schema xmlns:tns="http://currency.ase.mmberg.net/" xmlns:xs="http:
  <xs:element name="convert" type="tns:convert"/>
  <xs:element name="convertResponse" type="tns:convertResponse"/>
 ▼<xs:complexType name="convert">
  ▼<xs:sequence>
     <xs:element name="from" type="tns:currency"/>
     <xs:element name="to" type="tns:currency"/>
     <xs:element name="value" type="xs:float"/>
   </xs:sequence>
  </xs:complexType>
 ▼<xs:complexType name="convertResponse">
  ▼<xs:sequence>
     <xs:element name="return" type="xs:float"/>
   </xs:sequence>
  </xs:complexType>
 ▼<xs:simpleType name="currency">
  ▼<xs:restriction base="xs:string">
     <xs:enumeration value="EUR"/>
     <xs:enumeration value="USD"/>
     <xs:enumeration value="NAD"/>
     <xs:enumeration value="GBP"/>
   </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Quellen und weiterführende Literatur

- http://www.w3.org/TR/ws-gloss/
- https://www.gi.de/service/informatiklexikon/detailansicht/article/web-services.html
- http://msdn.microsoft.com/de-de/library/yzbxwf53.aspx
- http://msdn.microsoft.com/de-de/library/ms996507.aspx
- http://www.w3schools.com/webservices/
- https://blogs.oracle.com/gopalan/en/entry/wsdl_soap_binding_style_to
- http://www.sascha-alda.de/lehre/soa/vorlesung/Kapitel%204%20-%20Web%20Services%20I.pdf
- https://www.cs.rutgers.edu/~pxk/417/notes/08-rpc.html

Quiz

>> *http://twbk.de/ASE14*

Wozu dienen Webservices?

A	Als Nachfolger von Webseiten
B	Zur Kommunikation zwischen Mensch und Web
C	Zur Kommunikation zwischen Mensch und Maschine
D	Zur Kommunikation zwischen Maschinen über das Web

>> *http://twbk.de/ASE14*

Wozu dient eine WSDL Datei?

A	Zur Definition der Nachrichten, die zwischen WS ausgetauscht werden
B	Um WS über HTTP aufrufen zu können
C	Zur Spezifikation der WS-Schnittstelle
D	Zur Definition eines Schemas, das der WS nutzt

>> *http://twbk.de/ASE14*

Was ist SOAP?

A	Ein Protokoll zum Serialisieren von XML-Daten
B	Eine Policy für Service Oriented Architectures
C	Ein Protokoll zur Definition des Datenaustausches zwischen WS

>> *http://twbk.de/ASE14*

Was wird im PortType angegeben?

A	Die existierenden Operationen und deren Messages
B	Die URL des Service
C	Das XSD zur Definition der auszutauschenden Daten

>> *http://twbk.de/ASE14*

Was ist eine Proxyklasse ?

A	Kapselt die Funktionalität des entfernten Dienstes als lokale Klasse
B	Implementiert ein Proxy-Interface des WS
C	Implementiert den Socket zum Zugriff auf den WS

>> *http://twbk.de/ASE14*

Reicht ein Webserver zur Ausführung von Java EE aus?

A	**Ja**
B	**Nein**

>> *http://twbk.de/ASE14*

Was ist Jax WS?

A	**Eine Spezifikation zur Definition von SOAP-WS in Java**
B	**Eine Implementierung zum Erstellen von SOAP-WS in Java**
C	**Ein Tool zum Testen von WS in Java**

>> *http://twbk.de/ASE14*

Was ist Serialisierung?

A	**Die Überführung einer hierarchischen in eine flache Datenstruktur**
B	**Das Persistieren eines Objektzustandes**
C	**Das Umwandeln eines Multithreading-Algorithmus in eine sequentielle Abarbeitung**

Teil II

Nachfolgend ein **kurzer Überblick** zu **weiterführenden Themen:**

- Faults
- WS Security
- UDDI
- Asynchrone Services
- Web Service Interaktion

Faults

- WSDL kennt Fault-Message:

```
<portType name="ConverterService">
  <operation name="convert">
    <input wsam:Action="http://currency.ase.mmberg.net/ConverterService/convertRequest" message="tns:convert"/>
    <output wsam:Action="http://currency.ase.mmberg.net/ConverterService/convertResponse"
       message="tns:convertResponse"/>
    <fault message="tns:ConverterFault" name="ConverterFault" wsam:Action="http://
       currency.ase.mmberg.net/ConverterService/convert/Fault/ConverterFault"/>
  </operation>
</portType>
```

- SOAP kennt keine Java-Exceptions
- Daher müssen unsere Exceptions entsprechend angepasst werden, damit folgendes Beispiel funktioniert:

```
@WebMethod
    public float convert(String from, String to, float value)
          throws ConverterFault{

        if (value<0){
            ConverterFaultBean f = new ConverterFaultBean();
            f.setFaultCode("err01");
            f.setFaultString("Wert zu klein");
            throw new ConverterFault(f);
        }

        return (value/eurToXMap.get(from))*eurToXMap.get(to);
    }
```

Faults: Exception

```
@WebFault(name="ConverterFault")
public class ConverterFault extends Exception {

   private ConverterFaultBean faultBean;

   public ConverterFault(){
      super();
   }

   public ConverterFault(ConverterFaultBean f){
      super(f.getFaultString());
      this.faultBean=f;
   }

   public ConverterFault(String message, ConverterFaultBean faultInfo, Throwable cause) {
      super(message, cause);
      faultBean = faultInfo;
   }

   public ConverterFaultBean getFaultInfo(){
      return faultBean;
   }
}
```

- Entsprechend Jax WS Definition
  - Von Exception erben
  - Annotieren mit @WebFault
  - Konstruktoren
  - FaultBean erstellen
  - getFaultInfo() liefert das FaultBean zurück
- Exception referenziert ConverterFaultBean (by convention)
  - WebFault name+“Bean“
  - Bean ist Grundlage für Anzeige des „Detail“ im SOAP Fault

Jax WS: http://download.oracle.com/otn-pub/jcp/jaxws-2.2-mrel3-evalu-oth-JSpec/jaxws-2_2-mrel3-spec.pdf?AuthParam=1414343586_27e7f7dc256c5cf22124021c241f229e

Faults: FaultBean

```
public class ConverterFaultBean {
    /** * Fault Code */
    private String faultCode;

    /** * Fault String */
    private String faultString;

    public ConverterFaultBean(){

    }

    /** * @return the faultCode */
    public String getFaultCode() { return faultCode; }

    /** * @param faultCode the faultCode to set */
    public void setFaultCode(String faultCode) { this.faultCode = faultCode; }

    /** * @return the faultString */
    public String getFaultString() { return faultString; }

    /** * @param faultString the faultString to set */
    public void setFaultString(String faultString) { this.faultString =
faultString; }
}
```

Fault: Beispiel

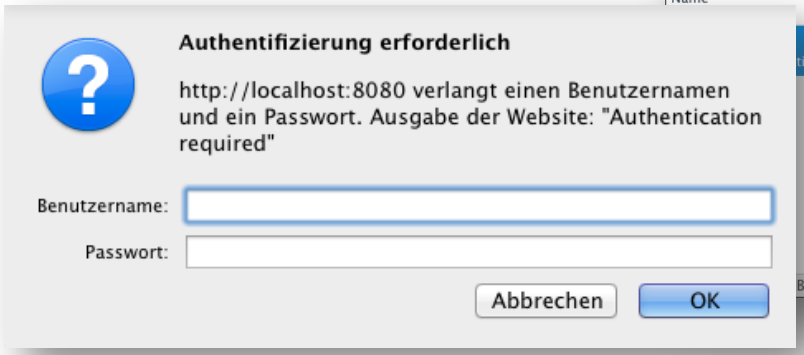
```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
         <faultcode>S:Server</faultcode>
         <faultstring>Wert zu klein.</faultstring>
         <detail>
            <ns2:ConverterFault xmlns:ns2="http://currency.ase.mmberg.net/">
               <faultCode>err01</faultCode>
               <faultString>Wert zu klein.</faultString>
            </ns2:ConverterFault>
         </detail>
      </S:Fault>
   </S:Body>
</S:Envelope>
```

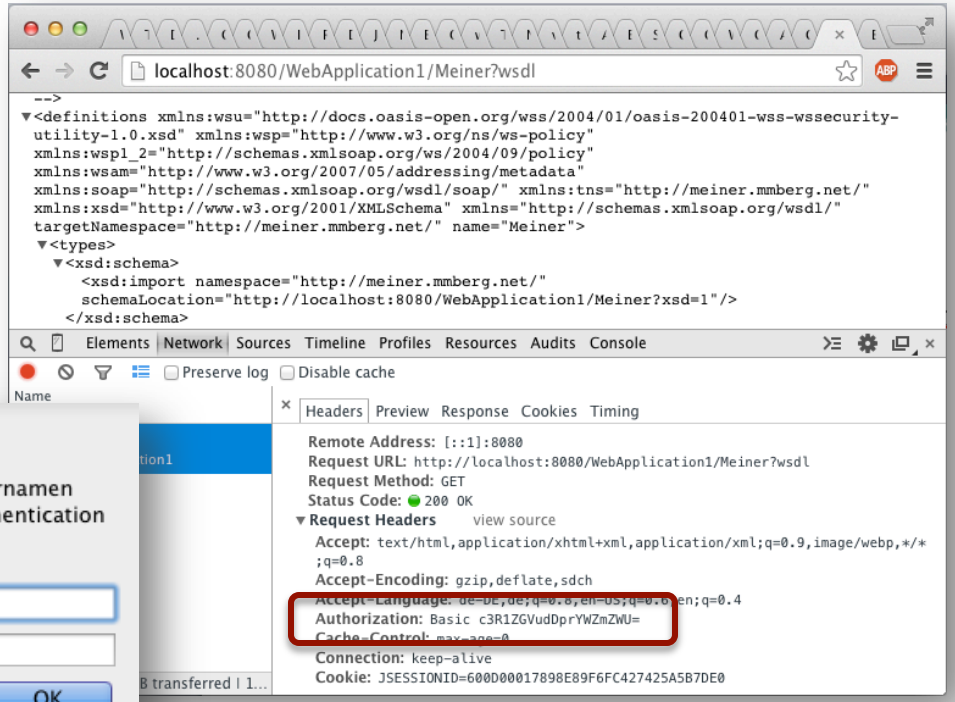
Sicherheit

- Prinzipiell können alle Sicherheitsmechanismen genutzt werden, die bei jeglicher Kommunikation über HTTP Anwendung finden (z.B. HTTP Basic Auth, SSL,…)
- Allerdings sind wir zusätzlich interessiert an:
  - End-to-End-Verschlüsselung
  - Feingranulare Definition der Sicherheitsmaßnahmen bzw. deren Beschreibung
  - Aushandlung bzw. Anbieten mehrerer Varianten
  - Frei abrufbare WSDL

HTTP Basic Auth

- Dem HTTP-Header wird ein „Authorization“-Attribut hinzugefügt mit dem Inhalt:
  - „Basic “ + base64(user:pass)
  - z.B.
    `Basic c3R1ZGVudDprYWZmZWU=`
  - Online Base64 Encoder: https://www.base64encode.org/





HTTP Basic Auth: web.xml

```
 <security-role>
   <description>WS User Role</description>
   <role-name>wsrole</role-name>
 </security-role>

<security-constraint>
     <web-resource-collection>
       <web-resource-name>WS User Role</web-resource-name>
       <url-pattern>/Meiner</url-pattern>
     </web-resource-collection>

     <auth-constraint>
           <role-name>wsrole</role-name>
     </auth-constraint>
     <user-data-constraint>
           <transport-guarantee>NONE</transport-guarantee>
     </user-data-constraint>
</security-constraint>

<login-config>
     <auth-method>BASIC</auth-method>
</login-config>
```

URL, die geschützt werden soll

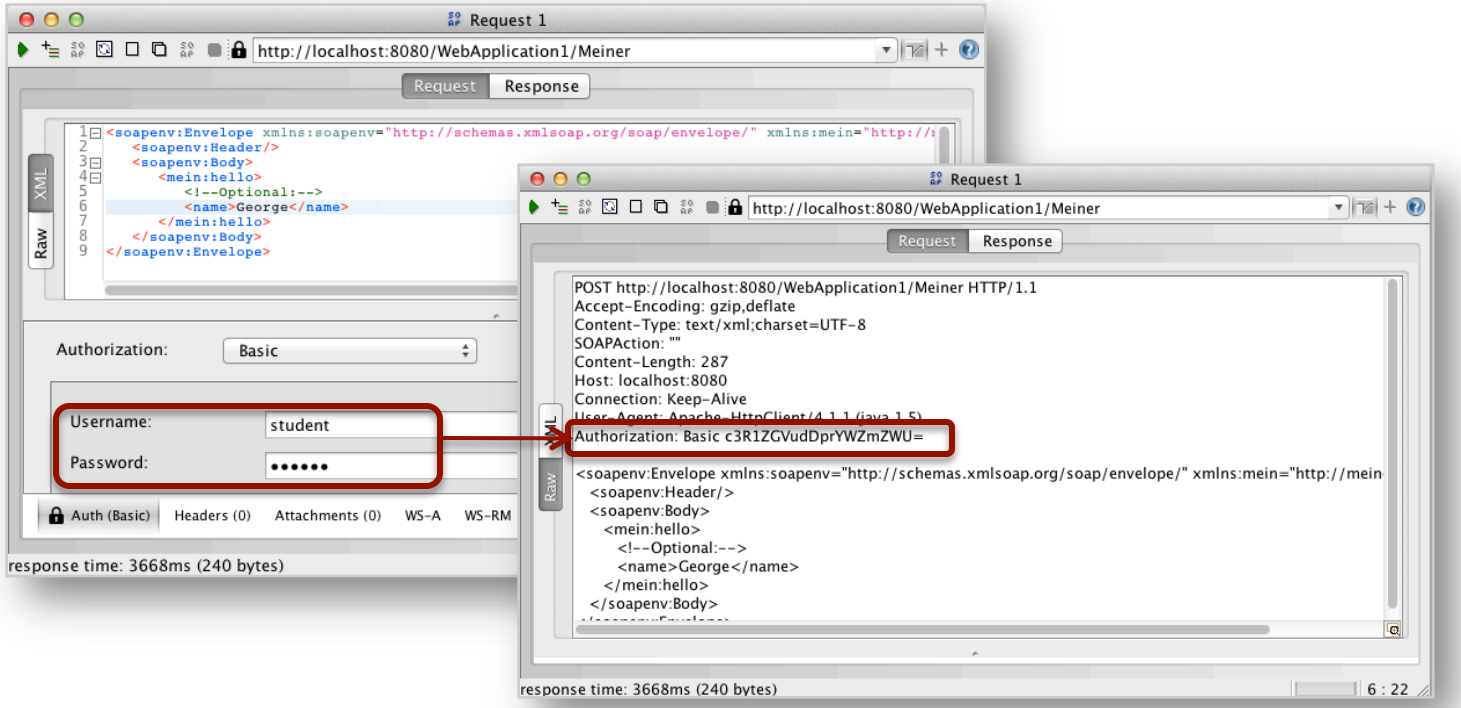
Nutzerrolle, die dem Application Server (z.B. Tomcat) bekannt ist

```
<role rolename="wsrole"/>
<user password="kaffee" roles="wsrole" username="student"/>
```

tomcat-users.xml

Nutzer dieser Rolle, der somit zum Einloggen genutzt werden kann

HTTP Basic Auth: Soap UI



- Nachteil: alle Dateien unterhalb der angegebenen URL befinden sich hinter dem Passwortschutz
  - D.h. auch zum Abrufen der WSDL muss ein Passwort eingegeben werden

WS Security (WSS)

- Daher Sicherheit (d.h. Informationen zu Authentifizierung bzw. Verschlüsselung und Signatur) nicht in den HTTP- sondern in den SOAP-Header integrieren
  - z.B.: XML Signature:
    - Algorithmus, Schlüssel, Signaturwert
  - z.B. Authentifizierung
    - UsernameToken: Username + Password

```
<UsernameToken wsu:Id="BeispielID">
  <Username>...</Username>
  <Password Type="...">...</Password>
  <Nonce EncodingType="...">...</Nonce>
  <Created>...</Created>
</UsernameToken>
```

- Es werden verschiedene „Policies“ je nach Art der Sicherheit angeboten
- Beschreibbar über WSDL

WS Security

- Beispiel SOAP-Request mit UsernameToken

```
<soapenv:Header>
<wsse:Security soapenv:mustUnderstand="1">
...
  <wsse:UsernameToken wsu:Id="UsernameToken-Bsp-123">
    <wsse:Username>student</wsse:Username>
    <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/
     oasis-200401-wss-username-token-profile-1.0#
     PasswordText">kaffee</wsse:Password>
    ...
  </wsse:UsernameToken>
</wsse:Security>
</soapenv:Header>
```

WS Policy

- Policies beschreiben welche Sicherheitsfeatures der Service erfordert bzw. unterstützt
- WS Policy dient zur Beschreibung solcher Policies
  - Angabe welche Elemente signiert und welche verschlüsselt werden müssen (als Bestandteil der WSDL-Datei)
    - Hier: entweder Body signieren oder ihn verschlüsseln
- Wird in WSDL angegeben

```
<wsp:Policy
        xmlns:sp="http://docs.oasis-
open.org/ws-sx/ws-securitypolicy/200702"
        xmlns:wsp="http://www.w3.org/ns/
ws-policy">
   <wsp:ExactlyOne> <!– or -->
     <wsp:All>
        <sp:SignedParts>
           <sp:Body/>
        </sp:SignedParts>
     </wsp:All>
     <wsp:All>
        <sp:EncryptedParts>
           <sp:Body/>
        </sp:EncryptedParts>
     </wsp:All>
   </wsp:ExactlyOne>
</wsp:Policy>
```

WS Security: WSDL

```
<wsdl:binding name="MyBinding" type="wns:myPortType">
    <wsp:PolicyReference xmlns:wsp="http://www.w3.org/ns/ws-policy"
        URI="#UsernameToken"/>
...
</wsdl:binding>
```

```
<!-- Policy for Username Token with plaintext password, sent from client to
    server only -->
  <wsp:Policy wsu:Id="UsernameToken" xmlns:wsu=
     "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
     xmlns:wsp="http://www.w3.org/ns/ws-policy>
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:SupportingTokens
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <sp:UsernameToken sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient"/>
          </wsp:Policy>
        </sp:SupportingTokens>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
```

http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.2/errata01/os/schemas/ws-securitypolicy-1.2.xsd

WS Security

- Sehr komplex und oftmals problematisch zu integrieren
- Konfiguration vereinfacht durch diverse Middlewares, z.B. Oracle WS Security Policy
  - oracle/wss_username_token_**service**_policy
  - oracle/wss_username_token_**client**_policy
  - oracle/wss_username_token_**over_ssl**_service_policy

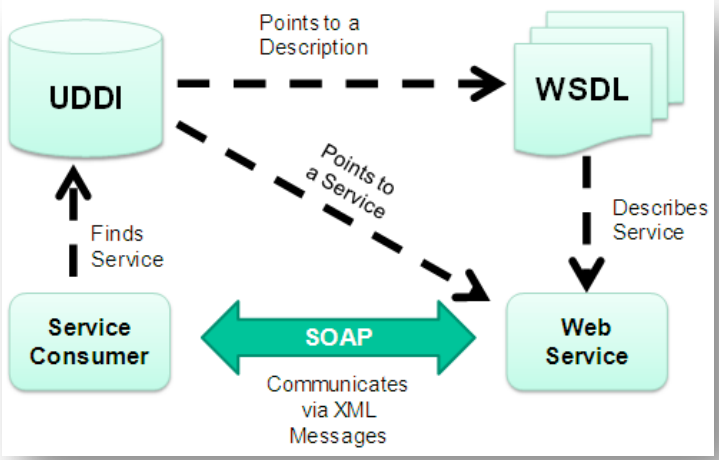
  This policy uses the credentials in the UsernameToken WS-Security SOAP header to authenticate users against the configured identity store. Both plain text and digest mechanisms are supported.

- Bei Nutzung des WebLogic Application Servers, Auswahl der Policy über Annotations

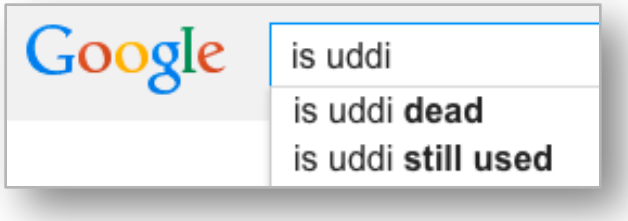
```
@SecurityPolicy(uri="policy:oracle/
wss10_username_token_with_message_protection_server_policy“)
```

Verzeichnisse: UDDI

- *Universal Description, Discovery and Integration*
- Dient zum Registrieren und Auffinden von Web Services
- Listet hierzu WSDLs und Informationen über den Betreiber des Dienstes auf
- Idee: Für bestimmte Bereiche gibt es entsprechende Verzeichnisse, z.B. Flugsuche
  - Alle Fluganbieter könnten ihr Interface (WSDL) dort hinterlegen, sodass Interessierte den Service finden und ansprechen können
- Nutzung in der Praxis? …



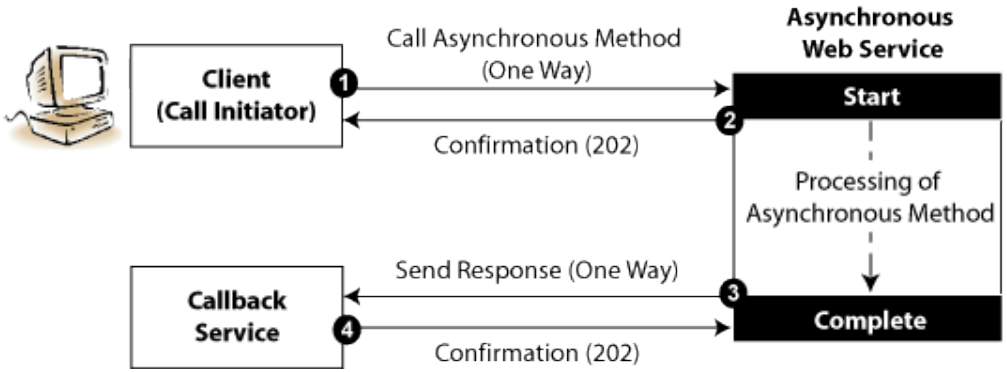
http://www.wst.univie.ac.at/workgroups/sem-nessi.



Asynchrone Webservices

- Bis jetzt: Request-Response
  - Oder: One-Way
- In bestimmten Anwendungsfällen dauert die Verarbeitung so lange, dass es zwischenzeitlich zum Timeout kommt
- Hier kommen asynchrone Webservices zum Einsatz
  - Eigentlich zwei One-Way-Dienste
  - Konsument fragt Dienst an und teilt ihm eine Antwortadresse (Callback) mit
    - Unter dieser Adresse steht ein Dienst zur Verfügung, der die Antwort verarbeitet

Asynchrone Webservices



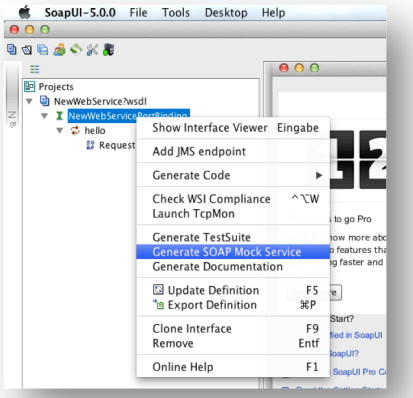
http://docs.oracle.com/cd/E23943_01/web.1111/e15184/asynch.htm

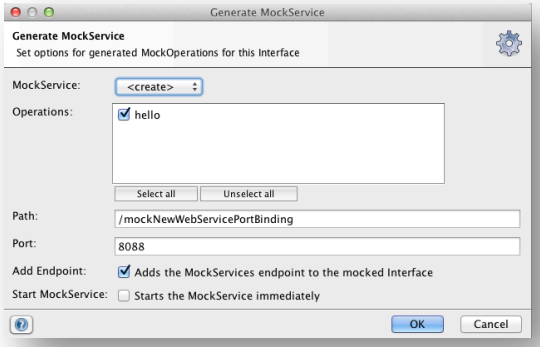
- Zwei PortTypes mit je einer (Input-)Message

```
<wsdl:portType name="HelloService">
    <wsdl:operation name="hello">
        <wsdl:input message="tns:helloInput"
               xmlns:ns1="http://www.w3.org/2006/05/addressing/wsdl"
               ns1:Action=""/>
    </wsdl:operation>
</wsdl:portType>
<wsdl:portType name="HelloServiceResponse">
    <wsdl:operation name="helloResponse">
        <wsdl:input message="tns:helloOutput"
               xmlns:ns1="http://www.w3.org/2006/05/addressing/wsdl"
               ns1:Action=""/>
    </wsdl:operation>
</wsdl:portType>
```

Asynchrone Webservices

- Testen eines asynchronen Services ist aufwändiger, da ein eigener Service betrieben werden muss, der die Antwort entgegennimmt
  - In Soap UI: Mock Service





- WS-Adressing: Im Request wird angegeben wohin die Antwort geschickt werden soll (Callback)

```
<wsa:Address>http://www.example.org/callback/path/</wsa:Address>
```

Web Service Interaktion

- WebServices dienen der Kommunikation zwischen Maschinen
  - Kommunikation kann auch mehr als zwei Parteien umfassen
- Oftmals konsumieren wir innerhalb eines Dienstes weitere WebServices
  - z.B. ein Service zur Ermittlung des günstigsten Autovermieters muss seinerseits die Services der verschiedenen Autovermieter abfragen und das günstigste Resultat an den Anfrager weiterleiten
  - Wir sprechen von „Mehrwertdiensten“ oder „Metadiensten“, da die Kombination von mehreren existierenden Diensten zu einer neuen Funktionalität führt
  - Ergo: Ein Service kann auch Client sein
  - Hauptaufgabe des Metadienstes:
    - Anpassen der entsprechenden Datenformate für Requests und Responses
    - Überschaubare Business Logik zur Auswertung der Ergebnisse und Bereitstellung der finalen Antwort
- Dies nennen wir ***Orchestrierung***
  (mehr dazu in der übernächsten Vorlesung)

Quellen und weiterführende Literatur

- http://docs.oracle.com/cd/E19316-01/819-3669/bnccv/index.html
- http://www.oio.de/public/xml/usernametoken-tutorial-webservice-security-artikel.htm
- http://docs.oracle.com/cd/E15523_01/web.1111/e13713/owsm_appendix.htm
- http://www.w3.org/TR/ws-policy/
- http://www.ibm.com/developerworks/java/library/j-jws10/index.html
- http://msdn.microsoft.com/en-us/library/ms977327.aspx
- http://msdn.microsoft.com/en-us/library/ms951253.aspx
- http://docs.oracle.com/cd/E23943_01/web.1111/e15184/asynch.htm
- https://blog.idrsolutions.com/2013/10/web-services-exception-handling/
- http://download.oracle.com/otn-pub/jcp/jaxws-2.2-mrel3-evalu-oth-JSpec/jaxws-2_2-mrel3-spec.pdf?AuthParam=1414343586_27e7f7dc256c5cf2212402lc241f229e

Teil III

REST Web Services

Status Quo

- Bis jetzt haben wir SOAP/WSDL als Grundlage für Webservices kennen gelernt
  - HTTP (meistens)
  - Auf Grundlage von XML
    - Nutzt Serialisierung / Deserialisierung
  - Beschreibung der Schnittstelle über WSDL
    - Automatische Proxygenerierung
    - Methoden in WSDL spezifiziert
      - Aufruf erinnert an RPC
  - Streng typisiert
    - Schema für Schnittstellenbeschreibung
    - Schema für Nachrichtenformat
    - Schema für Nachrichteninhalt
  - Validierbar

Nachteile

- Teilweise schwergewichtig
  - Viele Standards & Protokolle
  - Relativ hoher Aufwand fürs Testen (Client-Generierung)
  - XML vergrößert Datenmenge enorm

REST

- Representational State Transfer
- Ist ein Architekturstil (vgl. RPC)
  - Kein Framework oder Protokoll (vgl. SOAP)
- Idee: Roy Thomas Fielding „Architectural Styles and the Design of Network-based Software Architectures“ (2000, PhD-Thesis)
- Basiert auf HTTP und den entsprechenden Methoden („Verben“)
- REST ist nicht RPC!
  - Keine Übergabe von Methodenname und Argumentliste
  - Semantisch näher an SQL (SELECT, UPDATE, INSERT, DELETE)

REST-Prinzipien

- Uniforme Schnittstelle durch **HTTP Operationen**
  - … und deren **strikte Einhaltung**
  - Hierdurch plattform- und programmiersprachenunabhängig
- Eindeutig **identifizierbare Ressourcen** (Adressierbarkeit)
  - Links: das Web besteht aus verlinkten Seiten (bzw. Ressourcen)
  - Auch Ressourcen eines Dienstes können verlinkt werden
- **Zustandslosigkeit**
  - Keine Sessions!
  - Keine Cookies!
- Verschiedene Repräsentationen (XML, JSON)
  - Je nach Client
  - Auch RSS, vCard etc. vorstellbar

HTTP-Verben: Übersicht

- Entsprechend CRUD: Create, Read, Update, Delete
- GET
  - Ruft eine Ressource ab und verändert sie niemals
  - Ansonsten könnten Links/Crawler Änderungen bewirken!
- POST
  - Einer bestehenden Ressource Informationen hinzufügen
  - D.h. Hinzufügen einer neuen Ressource (unterhalb der angegebenen Ressource) ohne den Namen der neuen Ressource zu kennen
    - z.B. ein neues Produkt hinzufügen ohne dessen ID zu wissen (wird vom System zugewiesen)
  - Gibt Adresse der angelegten Ressource zurück
- PUT
  - Aktualisiert eine Ressource mit veränderten Daten oder legt diese an (falls der Name bekannt ist)
    - Z.B. neuen Mitarbeiter namens „Berg“ anlegen
  - gibt Adresse der angelegten/veränderten Ressource zurück
- DELETE
  - Löscht die angegebene Ressource

HTTP-Verben: Verhalten

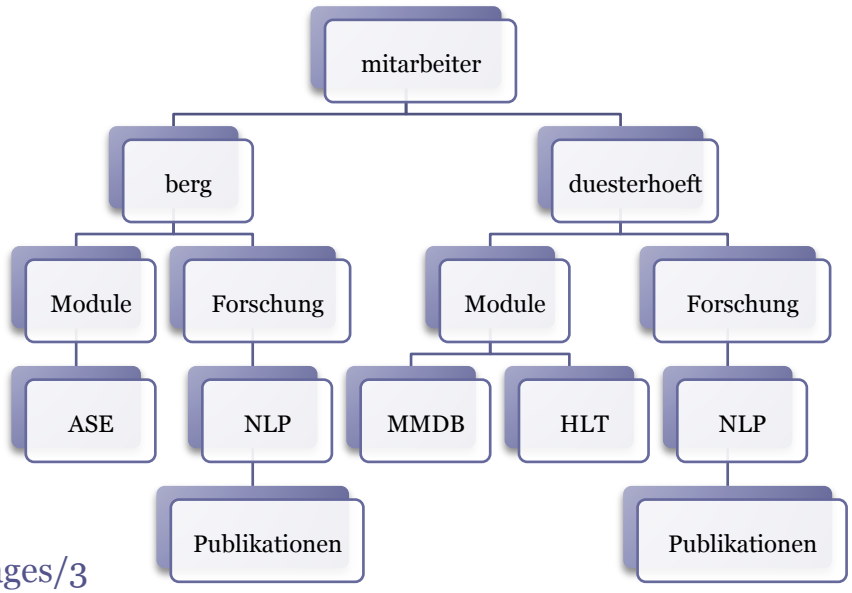
- GET hat keine Seiteneffekte
- PUT und DELETE sind idempotent
  - D.h. selbst bei mehrfacher Ausführung kommt es nicht zu Fehlern (d.h. in dem Fall erfolgt einfach keine Aktion)
    - Die Ressource wird nicht doppelt angelegt
    - Das Löschen einer Ressource die gelöscht wurde führt nicht zu einem Fehler

Ressourcen

- Ressourcen haben einen Namen und repräsentieren etwas (~ Entität, ~Konzept, ~ Objekt)
- Sie können adressiert und verlinkt werden
  - Mit Hilfe von URIs
- Die Interaktion mit Ressourcen erfolgt über HTTP
  - Nicht jeder HTTP-Aufruf ist REST
    - Bsp.: Ein GET-Aufruf, der eine Ressource verändert ist nicht RESTful
- Ressourcen sind selbstbeschreibend
  - D.h. alle Informationen die benötigt werden, um mit der Ressource zu interagieren sind im Request enthalten
- Was sind die Ressourcen meines Systems?
  - Die Mitarbeiter der Hochschule
  - Die Produkte eines Online Shops
  - Die Seiten eines Buches
  - Die Artikel einer Webseite
  - Die Ausgaben einer Zeitschrift
  - ...

URIs

- URIs identifizieren (=Name) Ressourcen
  - URLs lokalisieren Ressourcen (=Adresse) zusätzlich (eine URL ist eine URI)
  - URL gibt somit zusätzlich Informationen über Art und Ort des Zugriffs an (Protokoll und Server)
- URI als selbstdokumentierende Schnittstelle (leicht zu raten/merken/verstehen)
- URI ist dabei nicht nur ein String, sondern als ein bestimmter Pfad eines Baumes zu verstehen
- /mitarbeiter/{name}/module/{fach}
  - /mitarbeiter/duesterhoeft/module/hlt
  - /mitarbeiter/berg/module/ase
- Merkmale / Empfehlungen:
  - Lower case
  - Keine Leerzeichen
    - Stattdessen Bindestriche
  - Keine Dateien/Dateiendungen
    - index.html
  - Query-Strings vermeiden wenn es keine Query ist
    - ?page=3
    - Besser eine Ressource draus machen: /pages/3



URIs und Ressourcen

- Aufbau der URI ist individuell
  - /mitarbeiter/{name}/module/{fach}
  - Statische Elemente empfohlen bei Listen, z.B. /items/{itemid}
  - Kann auch komplett dynamisch sein (ohne statische Elemente)
    - z.B. bei einer Zeitschrift: /{year}/{issue}/{article}

Resource	URI	Verbs
All years	"/ "	GET
A particular year's issues	"/{year}"	GET, PUT
A particular issue	"/{year}/{issue}"	GET, PUT
An article	"/{year}/{issue}/{article}"	GET, POST (the article number will be assigned by the system), PUT, DELETE (delete would be turned off once an issue has been published)

MSDN

- Nicht jede URL identifiziert eine Ressource (im REST-Sinn):

/typo3/eui.html
/mitarbeiter/mueller/page2.php
/show-mitarbeiter?name=mueller

Technische Realisierung (php-Datei) einer Ressource uninteressant für deren Identifizierung

/category/studenten/faq/semesterablaufplan
/faculty/fiw/department/eui/manual/page/3

URI nicht selbstbeschreibend. Bei Requests für verschiedene Personen immer gleich (nur Query unterscheidet sich). Vermeiden von Verben!

Ressourcen

REST stellt Daten (Ressourcen) statt Methoden zur Verfügung.

- Mit Ressourcen kann gearbeitet werden (implizite Operationen)
  - /mitarbeiter → getAllMitarbeiter()
  - /mitarbeiter/mueller → getMitarbeiter(mueller)
- Ressourcen nutzen Substantive statt Verben
  - Beschreiben was sie „sind“ statt was sie „machen“
  - Das „Machen“ kommt durch das HTTP-Verb
  - Somit entsteht aus HTTP-Verb und Ressource implizit ein aus SOAP oder Java bekannter Methodenname, z.B. `getMitarbeiter(berg)` oder `putMitarbeiter(berg)` (vgl. `setMitarbeiter`)

Anzeigen eines Mitarbeiters:
GET /mitarbeiter/berg
(statt /berg.html)

Anlegen eines Mitarbeiters:
POST /mitarbeiter
Body: name=berg
(statt: /addMitarbeiter?name=berg)

Löschen eines Mitarbeiters
DELETE /mitarbeiter/berg
(statt: /deleteMitarbeiter?name=berg)

>> *http://twbk.de/ASE14*

Warum wäre die URI /article/pages/show/1 redundant?

A	Da durch das HTTP-Verb „GET“ klar ist, dass es um das Anzeigen einer Seite geht
B	Weil klar ist, dass ein Artikel Seiten hat

>> *http://twbk.de/ASE14*

Ist die URI /mitarbeiter/anlegen REST-konform?

A	**Ja**
B	**Nein**

>> *http://twbk.de/ASE14*

Mit welcher Methode werden neue Ressourcen unbekannten Namens unterhalb einer gegebenen Ressource angelegt?

A	POST
B	PUT

Physische vs. logische URIs

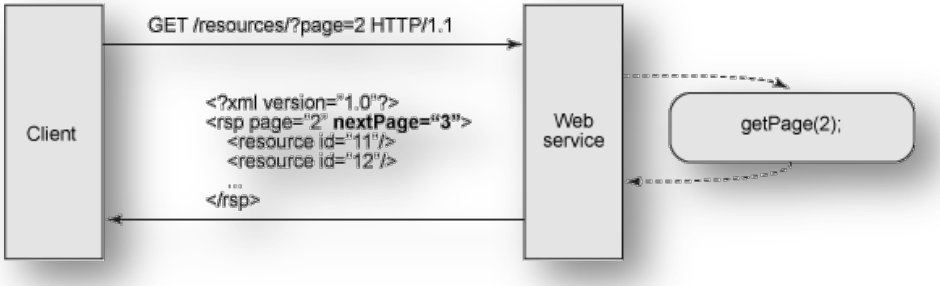
- Der Server hat nicht eine HTML-Seite für jeden Mitarbeiter
  - Stattdessen werden die Seiten dynamisch generiert
  - URLs mit Platzhaltern
    - /mitarbeiter/{name}

Vorteile von URIs

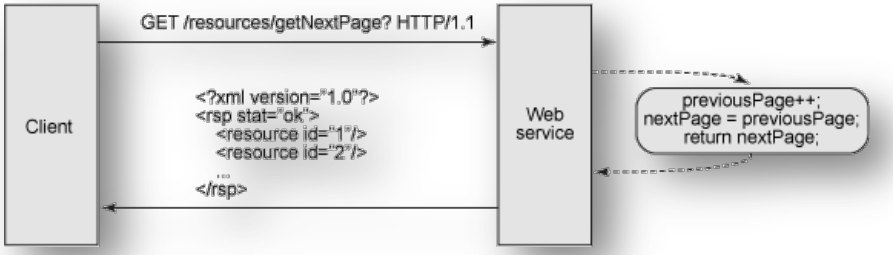
- Durch Suchmaschinen indizierbar
- Als Lesezeichen setzbar bzw. weiterleitbar (E-Mail, Chat)
- Zugriffe über Firewall steuerbar
  - z.B. nur lesender Zugriff auf eine Ressource
    - `DENY POST /mitarbeiter`

Stateless vs. Stateful

- Stateless
  - Jeder Request ist unabhängig vom vorherigen
  - Client muss dem Server alle Informationen mitteilen, die der Server zur Verarbeitung benötigt
  - z.B. HTTP
  - Bsp.: Server teilt Client nextPage in Antwort mit



- Stateful
  - Server besitzt einen dem Client zugeordneten internen Zustand
  - D.h. Server merkt sich Informationen und kann in weiteren Requests darauf zugreifen
  - z.B. FTP
  - Bsp.: Server merkt sich nextPage

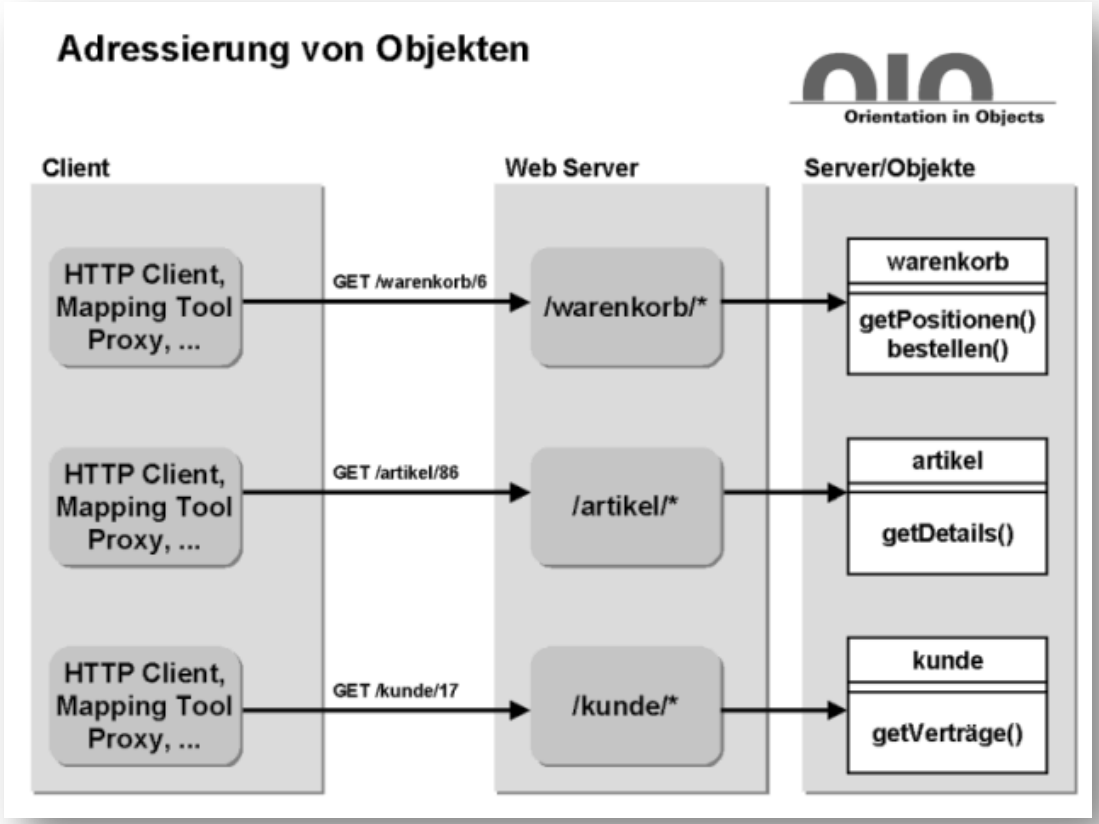


http://www.ibm.com/developerworks/library/ws-restful/

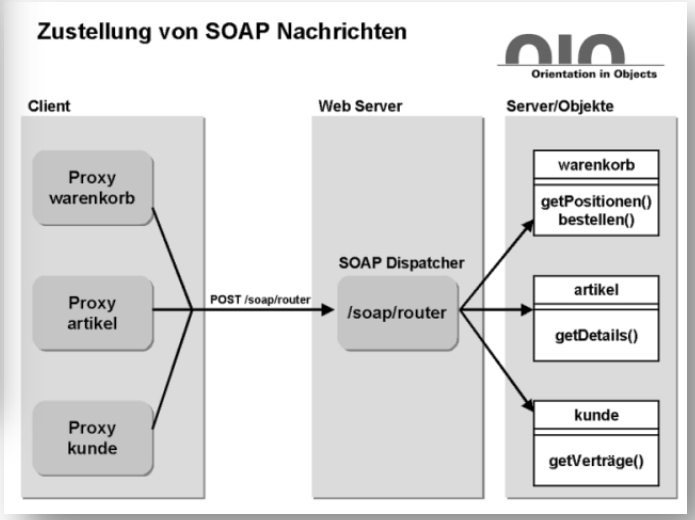
Stateless vs. Stateful II

- Ein weiteres Beispiel
  - Einkauf im Online Store. Frage an den Partner: „Habe ich an alles gedacht? Guck mal drüber! Hier ist die URL…“
  - Stateful: `/warenkorb`
    - User über Session identifiziert
    - Jeder sieht „seinen“ Warenkorb (je nach Login)
    - Nicht möglich ihn zu verlinken
  - Stateless: `/userA/warenkorb`
    - Warenkorb von UserA
    - Alle Informationen im Request
    - Verlinkbar
    - (Sicherheitsaspekt für dieses Beispiel außer Acht gelassen)

Adressierung: REST vs. SOAP

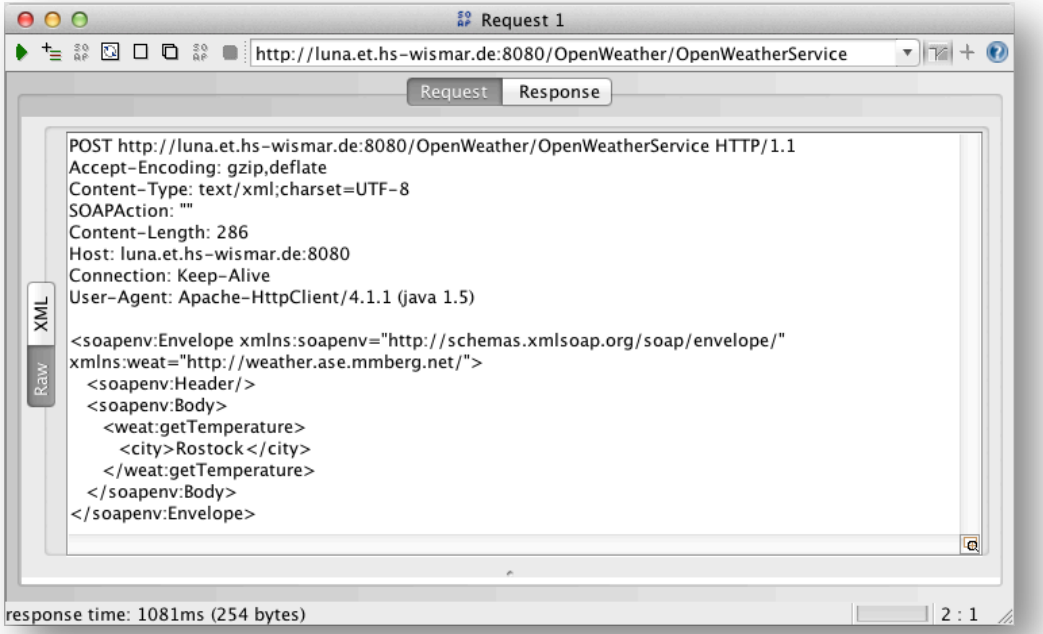


http://www.oio.de/public/xml/rest-webservices.htm



SOAP-Request vs. REST-Request

- SOAP: POST
  - Client erforderlich



- REST: GET
  - „aus dem Kopf“ möglich
  - Per Browser

```
GET http://luna.et.hs-wismar.de/wetter/rostock
```

Interoperabilität

- REST benötigt lediglich HTTP Stack (Client/Server)
  - Auf nahezu allen Geräten gegeben
- HTTP ist standardisiert
- SOAP + WS-* ist ebenfalls standardisiert, jedoch gibt es zahlreiche Versionen und verschiedene Standards
- Testen von RESTful Services
  - GET: Einfach im Browser aufrufen
  - POST: HTML-Formular
  - POST, DELETE, PUT: z.B. JavaScript (Ajax), curl,...

Sicherheit

- Keine eigene (vgl. WS Security)
  - D.h. weder Verschlüsselung, Signierung noch Authentifizierung über REST selbst
- Stattdessen Standard-HTTP-Mittel (SSL, BasicAuth)
  - D.h. ganz oder gar nicht (nicht möglich einige Teile der Nachricht unverschlüsselt lassen)
  - Transportverschlüsselung (nicht Ende-zu-Ende)

WADL

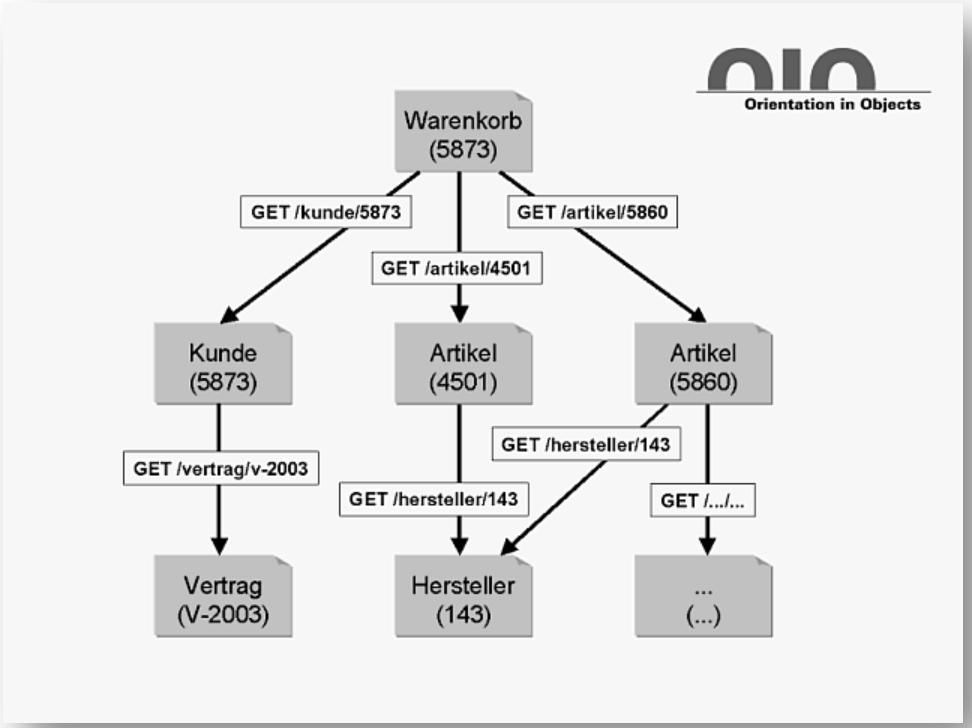
- Bestrebungen eine WSDL für REST zu schaffen
- z.B. YahooSearch.wadl

```
<application ...>
...
<resources base="http://api.search.yahoo.com/NewsSearchService/V1/">
    <resource path="newsSearch">
      <param name="appid" type="xsd:string" required="true" style="form"/>
     <method href="#search"/>
    </resource>
</resources>

<method name="GET" id="search">
    <doc xml:lang="en" title="Search news stories by keyword"/>
    <request>
      <param name="query" type="xsd:string" required="true">
        <doc xml:lang="en" title="Space separated keywords to search for"/>
      </param>
      ...
    </request>
    <response>
       <representation mediaType="application/xml" element="yn:ResultSet“/>
       <fault id="SearchError" status="400" mediaType="application/xml" element="ya:Error"/>
    </response>
</method>
```

Verlinkte Ressourcen

- z.B. mittels XLink
  - XML-Äquivalent zum HTML „<a>“
- Referenzen auf weitere Ressourcen
- Warenkorb besteht aus Artikeln, Artikel haben Hersteller,....



http://www.oio.de/public/xml/rest-webservices.htm

HATEOAS

- Hypermedia As The Engine Of Application State
  - Hypermedia = Links
  - Engine → Zustandsautomat
  - Application = die Anwendung bzw. Ressource
  - State = der Zustand des Automaten
- REST: REpresentational State Transfer
  - Die Repräsentation (einer Ressource) beschreibt ihren Zustand („selbstbeschreibend“)
- Idee: Es muss nur die Root-URL eines Dienstes bekannt sein. Der Rest wird durch Links erschlossen.
  - D.h. Zustandsübergänge durch Links
  - Konsequenz: Kein WADL nötig!
    - Dynamisches Interface statt fester Vertrag

HATEOAS: Beispiel

```
PUT /booking/1616163 HTTP/1.1
Host: www.mycompany.co.nz
Accept: application/vnd.mycompany.myapp-v1+xml
Authorization: OAuth ...

<booking>
  <flight>/flight/15263</flight>
  <seat>17F</seat>
  <meal>halal</meal>
</booking>

HTTP/1.1 200 OK
Conent-type: application/vnd.mycompany.myapp-v1.0+xml

<booking>
  <link rel="details" href="/booking/1616163"
        method="GET" type="application/vnd.mycompany.myapp+xml">
  <link rel="payment" href="/payment/booking"
        method="POST" type="application/vnd.mycompany.myapp+xml">
  <link rel="cancel" href="/booking/1616163"
        method="DELETE" type="application/vnd.mycompany.myapp+xml">
</booking>
```

http://de.slideshare.net/XEmacs/representational-state-transfer-rest-and-hateoas

Nachteile?

- Keine asynchronen Dienste
  - Natürlich kann ein Dienst asynchron aufgerufen werden sodass der Client nicht blockiert, die asynchrone Antwort ist und bleibt jedoch eine Antwort (und kein Request des Dienstes an den Konsumenten) und muss innerhalb des Timeouts geschehen
- Keine Beschreibung der Schnittstelle
  - Keine WSDL: „Erraten“ der Schnittstelle
    - Alternativen: WADL oder HATEOAS
  - Keine Proxyklassen

Leichtgewichtig vs. einfach

- SOAP ist komplex und vergleichsweise schwergewichtig
  - Die Nutzung ist jedoch enorm einfach (durch Proxy-Klassen und Frameworks)
- REST ist übersichtlich und leichtgewichtig
  - Die Nutzung ist u.U. aufwändiger, da hier der Code selbst geschrieben werden muss und nicht über Proxy-Klassen abstrahiert wird und auch kein RPC stattfindet

JSON

- Java Script Object Notation
  - Jedes JSON Dokument ist Java Script (-Objekt)
- Parser für die meisten Sprachen verfügbar
- Datenaustauschformat
- Alternative zu XML
  - Kürzer (u.a. keine Endtags)
  - Keine Attribute
  - Allerdings keine Typisierung / Schemas
- Beispiel:

JSON Example

```
{"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]}
```

http://www.w3schools.com/json/

XML Example

```
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>
```

JSON Syntax

- Daten werden als Schlüssel-Wert-Paare angegeben
  - Keys in Anführungsstrichen
    - Werte auch, falls es sich um Strings handelt
  - Getrennt durch Doppelpunkt

```
"Key":"Value"
```

- Daten werden in Objekten gekapselt
  - Kommagetrennt
  - Geschweifte Klammern

```
{"Key":"Value", "Key":"Value"}
```

- Arrays (kommasepariert, eckige Klammern)
  - Einfache Werte:

```
[A, B, C]
```

  - Objekte:

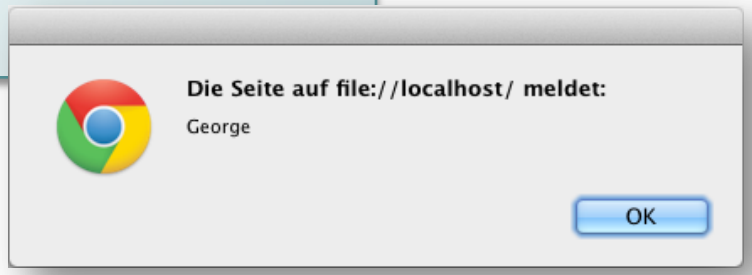
```
[
{"Key":"Value","Key":"Value"},
{"Key":"Value","Key":"Value"}
]
```

JSON & JavaScript & AJAX

- Da JSON JavaScript ist, kann direkt auf die Inhalte zugegriffen werden

```
var persons = [{"name":"Bill", "age":50},
               {"name":"George", "age":30}];

alert(persons[1].name);
```



- Daher beliebt in Verbindung mit REST und asynchronem Java Script (AJAX, hier mit JQuery)

Service liefert JSON mit Personen zurück als Variable „data“ (Callback)

URI zu REST-Service (oder auch: personen/bill)

```
$.get("personen", function(data) {
alert(data[1].name);
});
```

Entwicklung mit Jax RS

- Java API zur Erstellung von REST-Services
- Referenzimplementierung: Jersey
- Steuerung über Annotationen
  - Ähnlich wie Jax WS
- Ebenfalls lauffähig auf Tomcat bzw. Application Server

https://jersey.java.net/

web.xml vs. Application

- web.xml

```
<servlet>
     <servlet-name>javax.ws.rs.core.Application</servlet-name>
     <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
     <servlet-name>javax.ws.rs.core.Application</servlet-name>
     <url-pattern>/*</url-pattern>
</servlet-mapping>
```

- Application (ab Jersey 2.0)
  - Klasse ableiten von `javax.ws.rs.core.Application`
  - Annotation: `@ApplicationPath("/")`
  - Ggf. getClasses() überschreiben und alle Klassen, die Ressourcen darstellen (also mit @Path annotiert sind) registrieren

```
@Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        resources.add(net.mmberg.ase.resources.Mitarbeiter.class);
        return resources;
    }
```

Beispiel: Mitarbeiterverzeichnis

```java
@Path("/")
public class Service {

    private HashMap<String, Mitarbeiter> mitarbeiter=new HashMap<>();

    public Service(){
        mitarbeiter.put("berg",new Mitarbeiter("markus","berg"));
        mitarbeiter.put("jonas",new Mitarbeiter("ernst","jonas"));
        mitarbeiter.put("duesterhoeft",new Mitarbeiter("antje","düsterhöft"));
    }

    @GET
    @Path("mitarbeiter")
    @Produces(MediaType.APPLICATION_XML)
    public Collection<Mitarbeiter> getMitarbeiter(){
        return mitarbeiter.values();
    }

    @GET
    @Path("mitarbeiter/{name}")
    @Produces(MediaType.APPLICATION_JSON)
    public Mitarbeiter getMitarbeiter(@PathParam("name") String name){
        return mitarbeiter.get(name);
    }
}
```

```java
@XmlRootElement
public class Mitarbeiter {

    private String nachname;
    private String vorname;

    public Mitarbeiter(){

    }

    public Mitarbeiter(String vorname, String nachname){
        this.vorname=vorname;
        this.nachname=nachname;
    }

    /**
     * @return the nachname
     */
    public String getNachname() {
        return nachname;
    }
```

Ausgabeformat

- Definiert durch Annotation @Produces

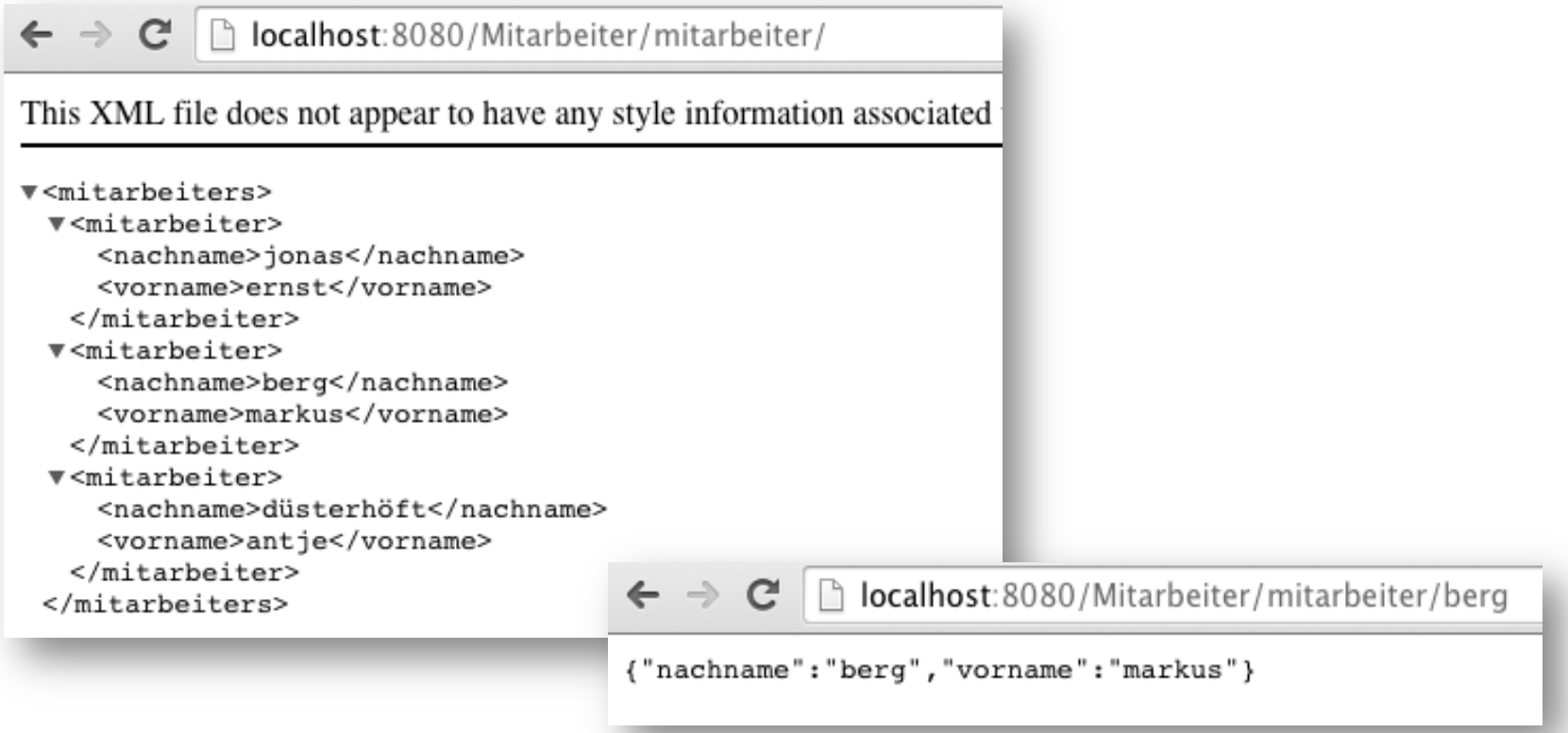
```
@Produces(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_XML)
@Produces(MediaType.TEXT_PLAIN)
```

- Auch mehrere Formate gleichzeitig möglich
  - Auswählbar durch HTTP-Header

```
"Accept: application/xml"
```

Client: Browser

- GET-Request aller Mitarbeiter/ eines bestimmten Mitarbeiters



Client: curl

```
curl -v -H "Accept: application/xml" http://
localhost:8080/Mitarbeiter/mitarbeiter/
```

```
curl -v -H "Accept: application/json" http://
localhost:8080/Mitarbeiter/mitarbeiter/
```

```
GET /Mitarbeiter/mitarbeiter/ HTTP/1.1
> User-Agent: curl/7.21.4 (universal-apple-darwin11.0) libcurl/
7.21.4 OpenSSL/0.9.8z zlib/1.2.5
> Host: localhost:8080
> Accept: application/json
>
< HTTP/1.1 200 OK
< Server: Apache-Coyote/1.1
< Content-Type: application/json
< Content-Length: 125
< Date: Mon, 10 Nov 2014 21:04:13 GMT
<
* Connection #0 to host localhost left intact
* Closing connection #0
[{"nachname":"jonas","vorname":"ernst"},
{"nachname":"berg","vorname":"markus"},
{"nachname":"düsterhöft","vorname":"antje"}]
```

Beispiel: Mitarbeiterverzeichnis (POST)

```
@POST
@Consumes("application/x-www-form-urlencoded")
@Produces(MediaType.APPLICATION_JSON)
@Path("mitarbeiter")
public Response addMitarbeiter(@FormParam("vorname") String vorname, @FormParam("nachname") String nachname){
    mitarbeiter.put(nachname.toLowerCase(),new Mitarbeiter(vorname, nachname));
    return Response.created(context.getAbsolutePathBuilder().path(nachname.toLowerCase()).build()).build();
}
```

```
curl -v --data "vorname=Herbert&nachname=Litschke"
  http://localhost:8080/Mitarbeiter/mitarbeiter
```

POST

```
< HTTP/1.1 201 Created
< Server: Apache-Coyote/1.1
< Location: http://localhost:8080/
Mitarbeiter/mitarbeiter/litschke
< Content-Length: 0
< Date: Mon, 10 Nov 2014 21:15:22 GMT
```

```
▼<mitarbeiters>
  ▼<mitarbeiter>
     <nachname>jonas</nachname>
     <vorname>ernst</vorname>
   </mitarbeiter>
  ▼<mitarbeiter>
     <nachname>berg</nachname>
     <vorname>markus</vorname>
   </mitarbeiter>
  ▼<mitarbeiter>
     <nachname>Litschke</nachname>
     <vorname>Herbert</vorname>
   </mitarbeiter>
  ▼<mitarbeiter>
     <nachname>düsterhöft</nachname>
     <vorname>antje</vorname>
   </mitarbeiter>
 </mitarbeiters>
```

GET

WADL

- http://localhost:8080/Mitarbeiter/application.wadl

```
▼<grammars>
  ▼<include href="application.wadl/xsd0.xsd">
     <doc title="Generated" xml:lang="en"/>
   </include>
 </grammars>
▼<resources base="http://localhost:8080/Mitarbeiter/">
```

```
▼<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0">
   <xs:element name="mitarbeiter" type="mitarbeiter"/>
  ▼<xs:complexType name="mitarbeiter">
    ▼<xs:sequence>
       <xs:element name="nachname" type="xs:string" minOccurs="0"/>
       <xs:element name="vorname" type="xs:string" minOccurs="0"/>
     </xs:sequence>
   </xs:complexType>
 </xs:schema>
```

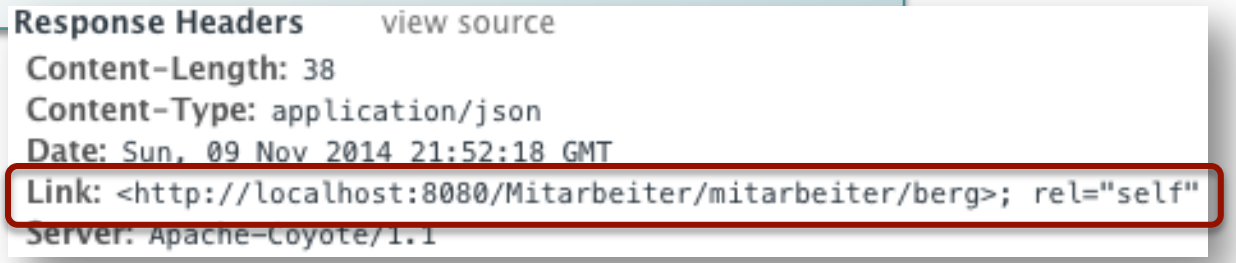
```
▼<resource path="mitarbeiter/{name}">
   <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="name" style="template" type="xs:string"/>
  ▼<method id="getMitarbeiter" name="GET">
    ▼<response>
       <ns2:representation xmlns:ns2="http://wadl.dev.java.net/2009/02" xmlns="" element="mitarbeiter" mediaType="application/json"/>
     </response>
   </method>
 </resource>
```

HATEOAS

- Ab Jersey 2.9: `@InjectLink`
- Alternative (Response Header): `javax.ws.rs.core.Link`

```
@Context
private UriInfo context;

@GET
@Path("mitarbeiter/{name}")
@Produces(MediaType.APPLICATION_JSON)
public Response getAllMitarbeiter(@PathParam("name") String name){
return Response.ok(mitarbeiter.get(name))
  .link(context.getBaseUriBuilder().path("mitarbeiter").path(name).build(), "self")
  .build();
```



- Alternative (Response Body):
  d.h. als Teil des Objektes
  - Link-Element mit „rel“ und „href“

```
<Person>
  <nachname>Berg</nachname>
  <vorname>Markus</vorname>
  <link rel="self" href="/http://localhost:8080/
Mitarbeiter/mitarbeiter/berg">
</Person>
```

Quellen und weiterführende Literatur

- http://openbook.galileocomputing.de/java7/1507_13_002.html
- http://jaxenter.de/artikel/REST-bessere-Web-Service-167838
- http://www.oio.de/public/xml/rest-webservices.htm
- http://www.ibm.com/developerworks/library/ws-restful/
- http://msdn.microsoft.com/en-us/magazine/dd315413.aspx
- http://blog.2partsmagic.com/restful-uri-design/
- http://msdn.microsoft.com/en-us/magazine/dd942839.aspx
- http://social.technet.microsoft.com/wiki/contents/articles/23838.project-siena-creating-a-wadl-configuration-file.aspx
- http://www.w3schools.com/json/
- http://jaxenter.de/artikel/rest-hateoas-fielding-176645
- http://de.slideshare.net/XEmacs/representational-state-transfer-rest-and-hateoas
- https://docs.oracle.com/javaee/7/tutorial/doc/jaxrs002.htm