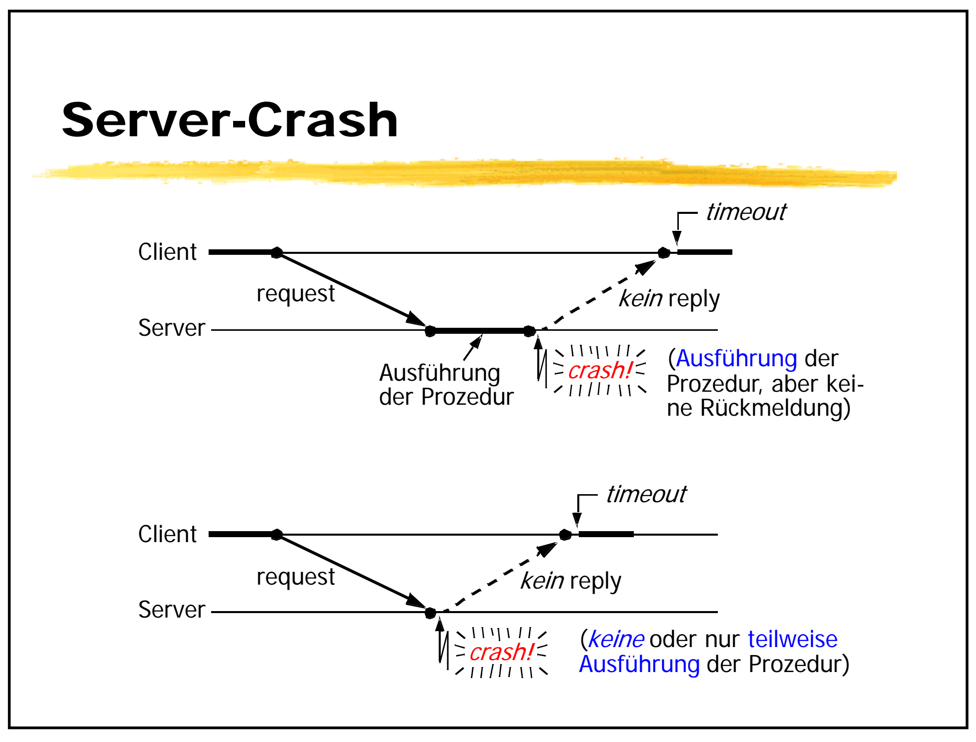
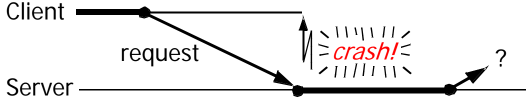


Server-Crash (2)

- Probleme: Wie soll der Client obige Fälle unterscheiden?
  - ebenso: Unterschied zu verlorenem request bzw. reply?
  - Sinn und Erfolg konkreter Gegenmassnahmen hängt u.U. davon ab!
  - Client meint evtl. zu Unrecht, dass ein Auftrag nicht ausgeführt wurde (→ falsche Sicht des Zustandes!)
- Evtl. Probleme nach einem Server-Restart
  - z.B. „Locks“, die noch bestehen (Gegenmassnahmen?) bzw. allgemein: „verschmutzter“ Zustand durch frühere Inkarnation
  - typischerweise ungenügend Information, um in alte Transaktionsszustände problemlos wieder einzusteigen

Client-Crash

- Oder auch:
  Client mittlerweile nicht mehr am reply interessiert



- Reply des Servers wird nicht abgenommen
  - Server wartet z.B. vergeblich auf eine Bestätigung (wie unterscheidet der Server dies von langsamen Clients oder langsamen Nachrichten?)
  - blockiert i.Allg. Ressourcen beim Server!

Client-Crash (2)

- Problem: „Orphans" (Waisenkinder) beim Server
  - Prozesse, deren Auftraggeber nicht mehr existiert
- Nach Restart könnte ein Client versuchen, Orphans zu terminieren (z.B. durch Benachrichtigung der Server)
  - Orphans könnten aber bereits andere RPCs abgesetzt haben, weitere Prozesse gegründet haben,...
- Pessimistischer Ansatz: Server fragt bei laufenden Auf-trägen von Zeit zu Zeit und vor wichtigen Operationen beim Client zurück (ob dieser noch existiert)
- „Sehr" alte Prozesse, die für einen Auftrag gegründet wurden, werden als Orphans angesehen und terminiert

RPC-Fehlersemantik-Klassen

- Operationale Sichtweise:
  - Wie wird nach einem Timeout auf (vermeintlich?) nicht eintreffende Nachrichten, wiederholte Requests, gecrashte Prozesse reagiert?

---

1. Maybe-Semantik:
   - Keine Wiederholung von Requests
   - Einfach und effizient
   - Keinerlei Erfolgsgarantien → nur ausnahmsweise anwendbar
   - Mögliche Anwendungsklasse: Auskunftsdienste (Anwendung kann es evtl. später noch einmal probieren, wenn keine Antwort kommt)

RPC-Fehlersemantik-Klassen (2)

2. At-least-once-Semantik:
   - Hartnäckige automatische Wiederholung von Requests
   - Keine Duplikatserkennung (zustandsloses Protokoll auf Serverseite)
   - Akzeptabel bei idempotenten Operationen (z.B. Lesen einer Datei)

   1) und 2) werden etwas euphemistisch oft als „best effort" bezeichnet

---

3. At-most-once-Semantik:
   - Erkennen von Duplikaten (Sequenznummern, log-Datei etc.)
   - Keine wiederholte Ausführung der Prozedur, sondern evtl. erneutes Senden des (gemerkten) Reply
   - Geeignet auch für nicht-idempotente Operationen

RPC-Fehlersemantik-Klassen (3)

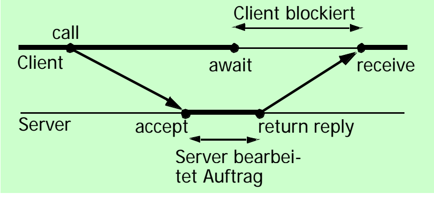
- Maybe → at-least-once → at-most-once → ...
  ist zunehmend aufwändiger zu realisieren

Ist „exactly-once" machbar?

- Man begnügt sich daher, falls es die Anwendung erlaubt, oft mit einer billigeren aber weniger perfekten Fehlersemantik
- Motto: so billig wie möglich, so „perfekt" wie nötig

Asynchroner RPC

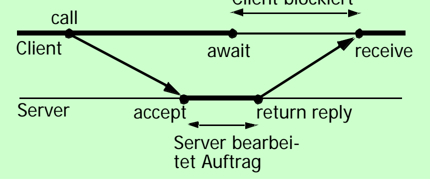
- Andere Bezeichnung: „Remote Service Invocation"
- Auftragsorientiert, d.h. also: Antwortverpflichtung



- Parallelverarbeitung von Client und Server möglich, solange Client noch nicht auf Resultat angewiesen

Asynchroner RPC: Zuordnung Auftrag/Ergebnisempfang

- Unterschiedliche Ausprägung auf Sprachebene möglich
  - „await“ könnte z.B. einen bei „call“ zurückgelieferten „handle“ als Parameter erhalten, also z.B.: `Y = call X(…); … await(Y);`



  - evtl. könnte die Antwort auch asynchron in einem eigens dafür vorgesehenen Anweisungsblock empfangen werden (vgl. Interrupt- oder Exception-Routine)

Asynchroner RPC: Future-Variable

- Spracheinbettung evtl. auch durch „Future-Variablen“
- Future-Variable = „handle“ auf das in anderem Thread parallel berechnetes Funktionsergebnis
- Programm blockiert nur dann, wenn der Wert der Variable bei ihrer Nutzung noch nicht feststeht
- Beispiel (Scala):

```
...
val x = future(callRPC())
... // continue local computation
println(x()) // await x iff necessary
```

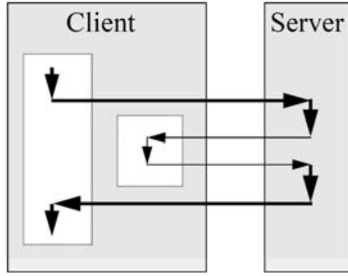
Einige Varianten / Ergänzungen von RPCs in der Praxis

Wir besprechen nachfolgend 4 Varianten / Ergänzungen:

1. Rückrufe
2. Context-Handles
3. Broadcast bzw. Multicast
4. Sicherheit

Rückrufe ("call back RPC")

- Temporärer Rollentausch von Client und Server
  - um eventuell bei langen Aktionen Zwischenresultate zurückzumelden
  - um eventuell weitere Daten vom Client anzufordern
- Client muss Rückrufadresse beim call übergeben



Context-Handles

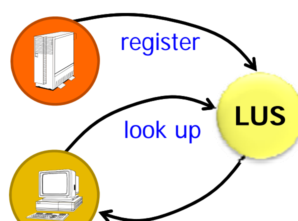
- Struktur mit Kontextinformation zur Verwaltung von Zustandsinformation über mehrere Aufrufe hinweg
  - vgl. „cookies“
- Context-Handles werden vom Server dynamisch erzeugt und an den Client (bei „reply“) zurückgegeben
- Client kann diese beim nächsten Aufruf (unverändert) wieder mitsenden
  - Server „erinnert“ sich so an den Kontext
- Vorteil: Server selbst arbeitet „zustandslos“

Weitere Varianten / Ergänzungen

- Broadcast bzw. Multicast
  - Request wird „gleichzeitig“ an mehrere Server geschickt
  - RPC ist beendet mit der ersten empfangenen Antwort oder Client hat die Möglichkeit, nach einer Antwort auf eine weitere Antwort zu warten
- Wählbare Sicherheitsstufen, z.B.:
  - Authentifizierung bei Aufbau der Verbindung (“binding”)
  - Authentifizierung pro RPC-Aufruf oder pro Nachricht
  - Verschlüsselung der Nachrichten
  - Schutz gegen Verfälschung (verschlüsselte Prüfsumme, MAC)

Lookup-Service

- Problem: Wie finden sich Client und Server?
  - haben i.Allg. verschiedene Lebenszyklen → kein gemeinsames Übersetzen / statisches Binden (fehlende gemeinsame Umgebung)
  - → Lookup Service (LUS) oder „Registry“ fungiert als „Service-Broker“



- Server gibt seinen Service (d.h. RPC-Routine) dem LUS bekannt
  - register: RPC-Schnittstelle „exportieren“ (Name, Parameter, Typen,...)
  - evtl. auch wieder abmelden
- Client erfragt beim LUS die Adresse eines geeigneten Servers
  - beim look up oder discovery Angabe des gewünschten Typs von Service
  - „importieren“ der RPC-Schnittstelle

Lookup-Service (2)

- Vorteile („Mehrwert“): im Prinzip kann LUS
  - mehrere Server für den gleichen Service registrieren (→ Fehlertoleranz; Lastausgleich)
  - Autorisierung etc. überprüfen
  - durch Polling der Server die Verfügbarkeit eines Services testen
  - verschiedene Versionen eines Dienstes verwalten
- Probleme:
  - look up kostet Ausführungszeit (gegenüber statischem Binden)
  - zentraler LUS ist ein potentieller Engpass / „single point of failure“ (Lookup-Service geeignet replizieren/verteilen?)
  - wie lernen Client oder Server die Adresse des „richtigen“ oder „zuständigen“ LUS kennen?

Web Services als Beispiel für das Client-Server-Modell

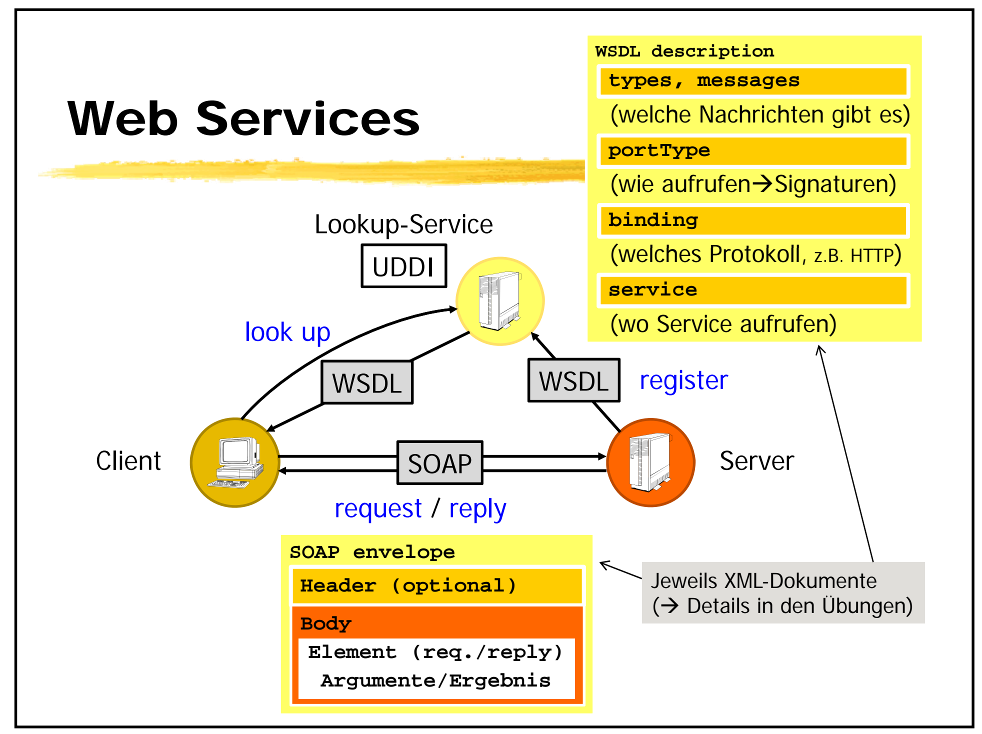
- Problem: Internet ist zu heterogen für eine einheitliche RPC-Laufzeitumgebung oder gar Sprache
  - „Lösung“: Web Services (Weiterentwicklung des XML-RPCs) als offener, plattform- bzw. sprachunabhängiger Standard, dessen Schnittstellen von diversen Plattformen implementiert werden können

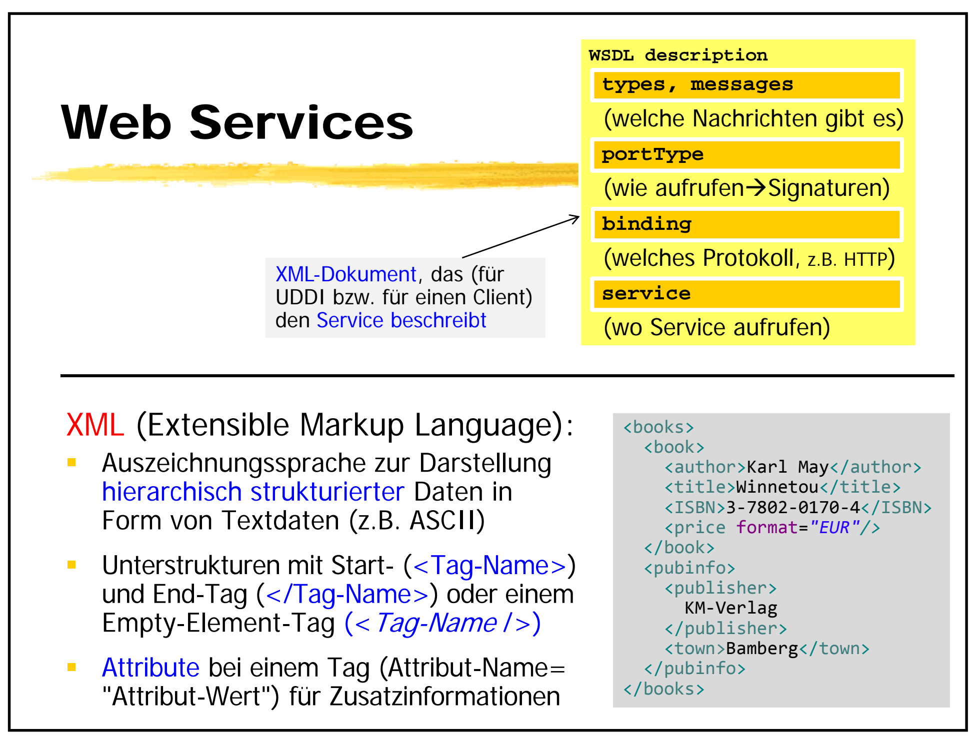
---

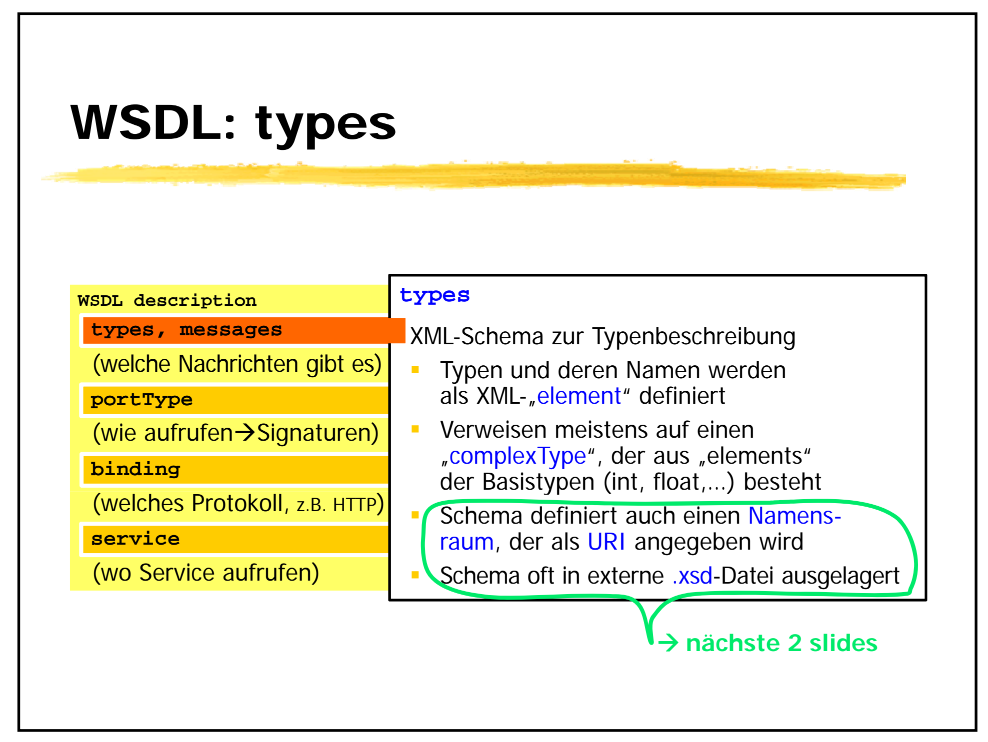
- HTTP als Transportschicht
- SOAP als plattformunabhängige Protokollspezifikation (ursprünglich: „Simple Object Access Protocol“)
- UDDI als Lookup-Service („Universal Description, Discovery and Integration“)
- WSDL als standardisierte Service-Beschreibung („Web Services Description Language“)

Alternativ zu HTTP: UDP oder SMTP, z.B. für Mitteilungen ohne Antwort oder wenn Resultatberechnung länger als typ. HTTP-Timeout dauert

Web Services




Web Services
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)
XML-Dokument, das (für UDDI bzw. für einen Client) den Service beschreibt
XML (Extensible Markup Language):
Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdaten (z.B. ASCII)
Unterstrukturen mit Start- (<Tag-Name>) und End-Tag (</Tag-Name>) oder einem Empty-Element-Tag (<Tag-Name />)
Attribute bei einem Tag (Attribut-Name= "Attribut-Wert") für Zusatzinformationen
<books>
<book>
<author>Karl May</author>
<title>Winnetou</title>
<ISBN>3-7802-0170-4</ISBN>
<price format="EUR"/>
</book>
<pubinfo>
<publisher>
KM-Verlag
</publisher>
<town>Bamberg</town>
</pubinfo>
</books>


WSDL: types
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)
types
XML-Schema zur Typenbeschreibung
Typen und deren Namen werden als XML-„element“ definiert
Verweisen meistens auf einen „complexType“, der aus „elements“ der Basistypen (int, float,...) besteht
Schema definiert auch einen Namens-raum, der als URI angegeben wird
Schema oft in externe .xsd-Datei ausgelagert
→ nächste 2 slides

WSDL-Namensräume

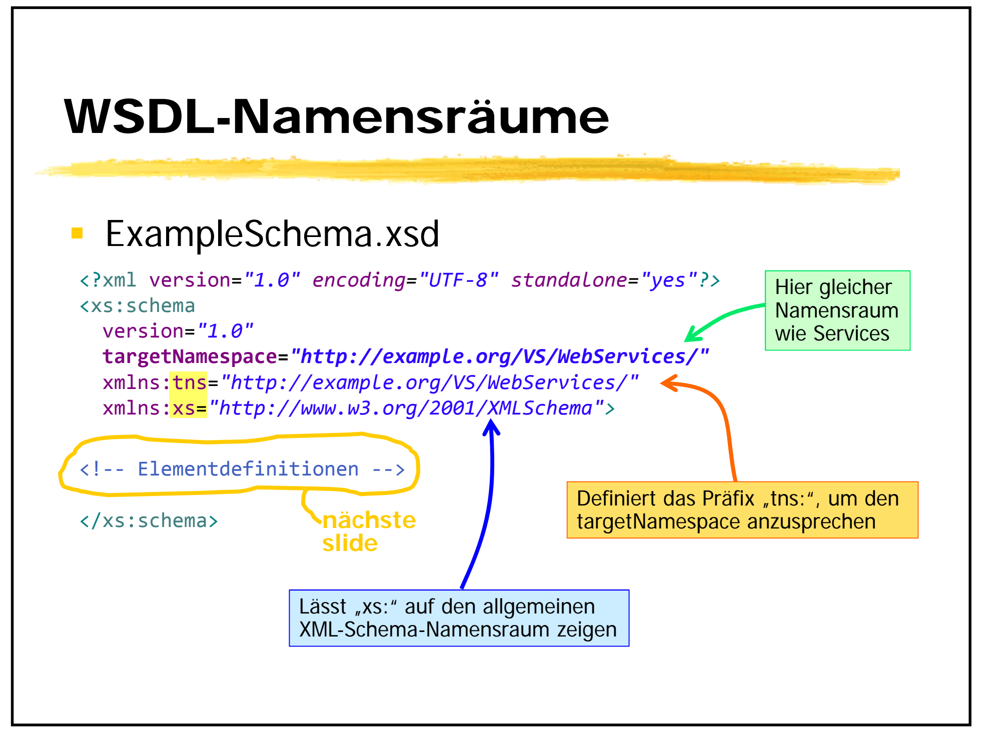
- ExampleWebServices.wsdl

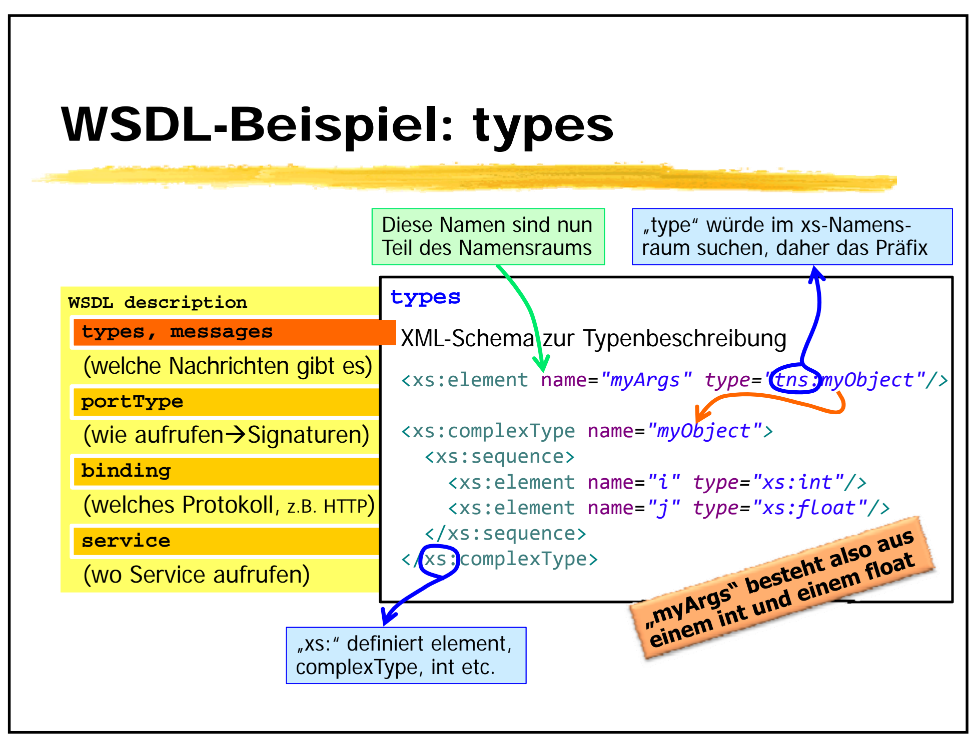
Namensraum der beschriebenen Web Services

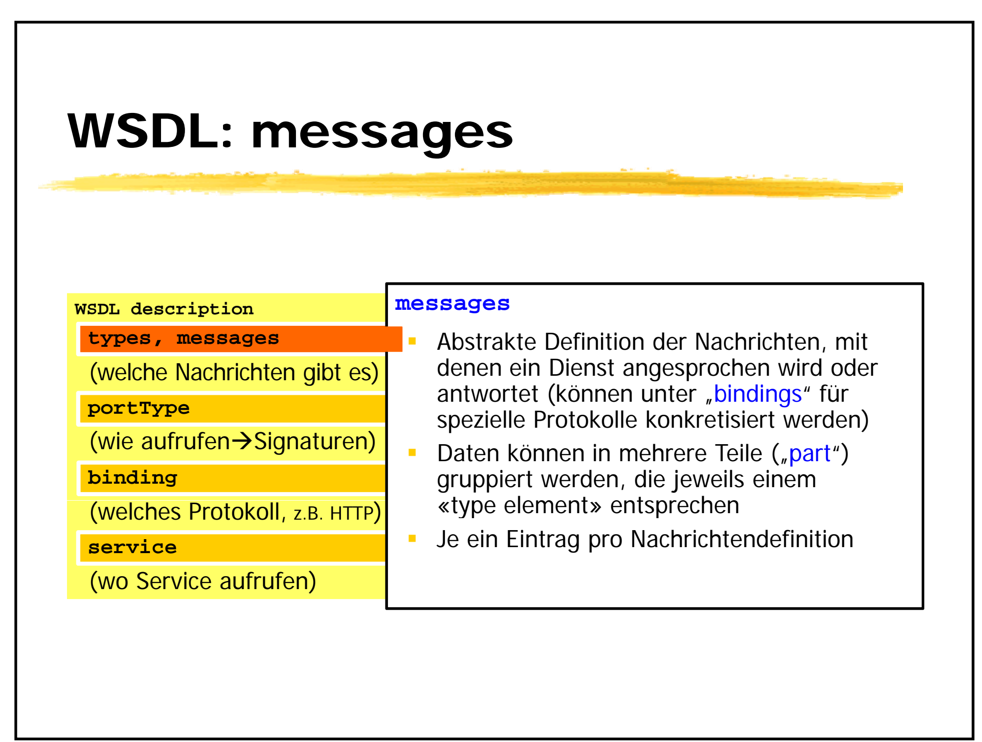
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions
  name="ExampleWebServices"
  targetNamespace="http://example.org/VS/WebServices/"
  xmlns:tns="http://example.org/VS/WebServices/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap">

  <types>
    <xsd:schema>
      <xsd:import
        namespace="http://example.org/VS/WebServices/"
        schemaLocation="ExampleSchema.xsd"/>
    </xsd:schema>
  </types>
...
```

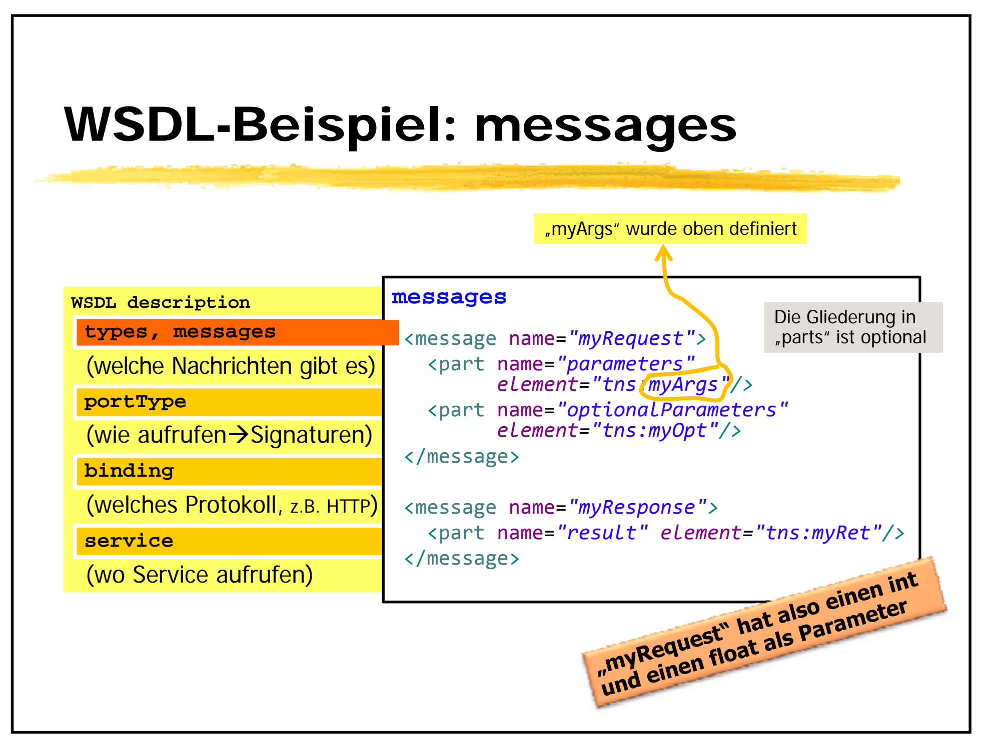
Weitere Namensräume, aus denen Tags benötigt werden

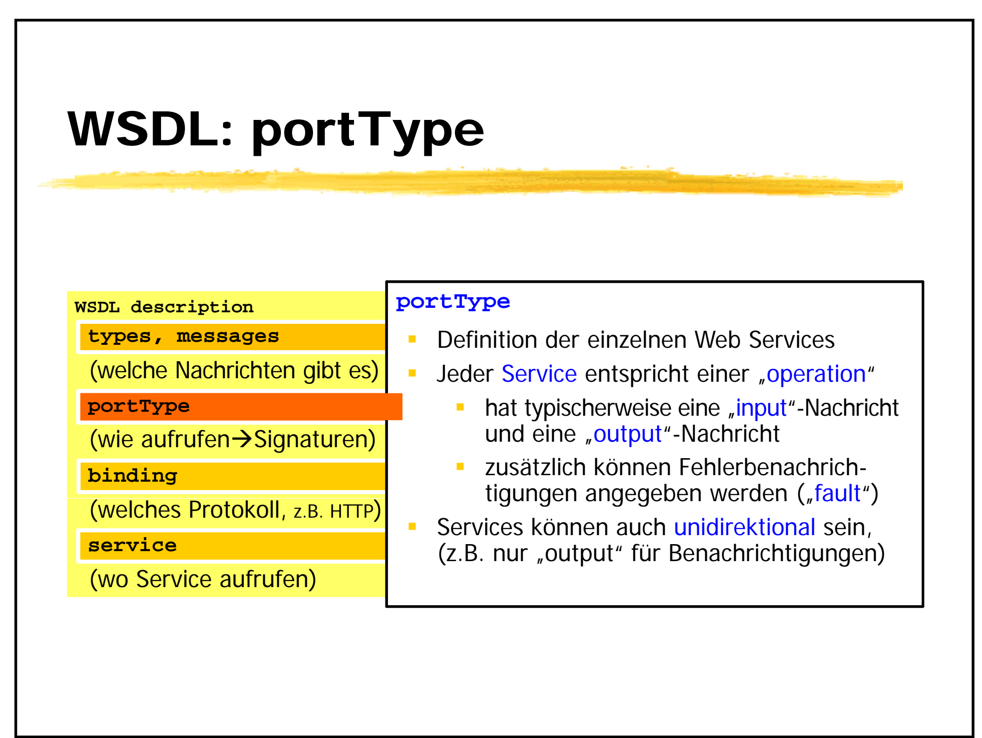
Importiert die Typendefinitionen; können auch eigenen Namensraum haben




WSDL-Beispiel: types
Diese Namen sind nun Teil des Namensraums
„type“ würde im xs-Namens-raum suchen, daher das Präfix
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)
types
XML-Schema zur Typenbeschreibung
<xs:element name="myArgs" type="tns:myObject"/>
<xs:complexType name="myObject">
<xs:sequence>
<xs:element name="i" type="xs:int"/>
<xs:element name="j" type="xs:float"/>
</xs:sequence>
</xs:complexType>
„myArgs“ besteht also aus einem int und einem float
„xs:“ definiert element, complexType, int etc.


WSDL: messages
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)
messages
Abstrakte Definition der Nachrichten, mit denen ein Dienst angesprochen wird oder antwortet (können unter „bindings“ für spezielle Protokolle konkretisiert werden)
Daten können in mehrere Teile („part“) gruppiert werden, die jeweils einem «type element» entsprechen
Je ein Eintrag pro Nachrichtendefinition


WSDL-Beispiel: messages
„myArgs“ wurde oben definiert
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)
messages
<message name="myRequest">
<part name="parameters"
element="tns:myArgs"/>
<part name="optionalParameters"
element="tns:myOpt"/>
</message>
<message name="myResponse">
<part name="result" element="tns:myRet"/>
</message>
Die Gliederung in „parts“ ist optional
„myRequest“ hat also einen int und einen float als Parameter


WSDL: portType
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)
portType
Definition der einzelnen Web Services
Jeder Service entspricht einer „operation“
hat typischerweise eine „input“-Nachricht und eine „output“-Nachricht
zusätzlich können Fehlerbenachrichtigungen angegeben werden („fault“)
Services können auch unidirektional sein, (z.B. nur „output“ für Benachrichtigungen)

WSDL-Beispiel: portType

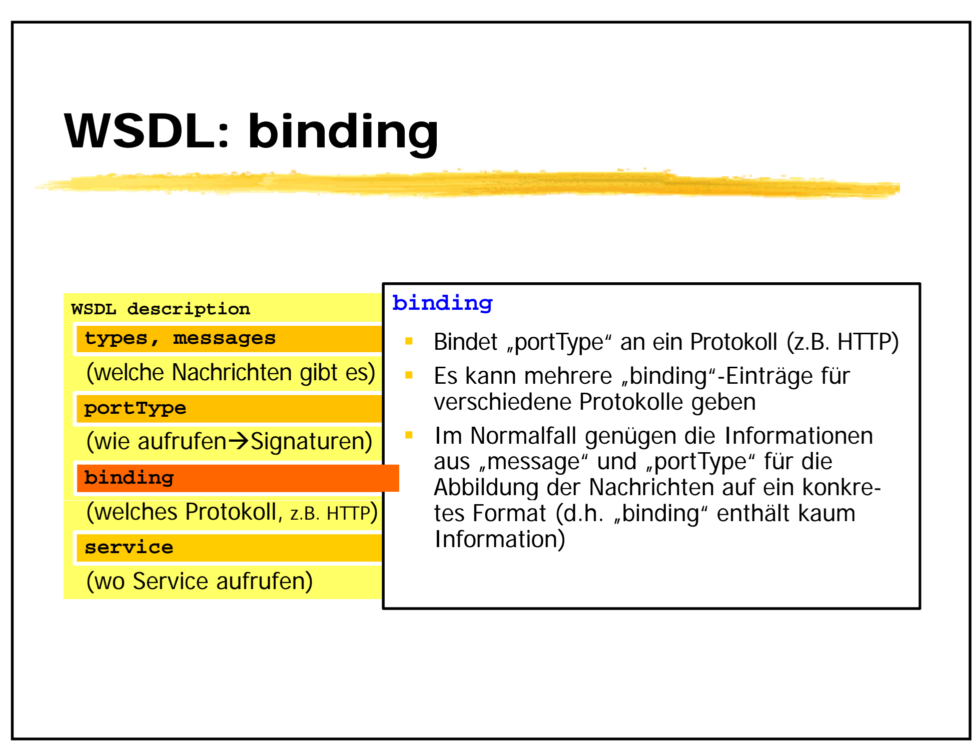
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)

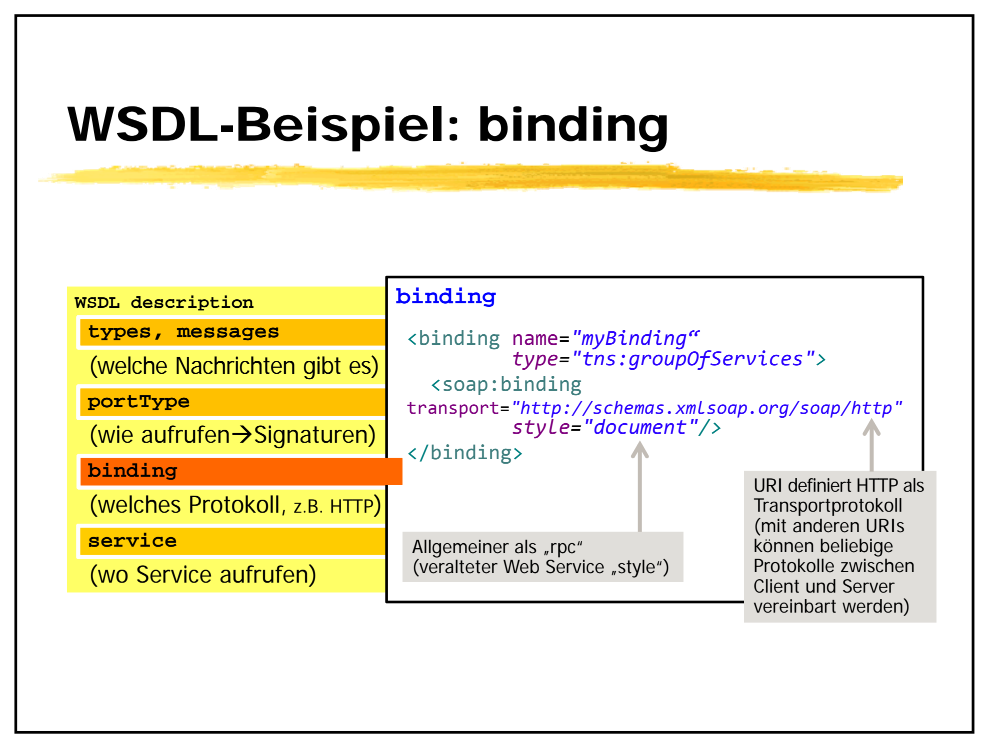
portType

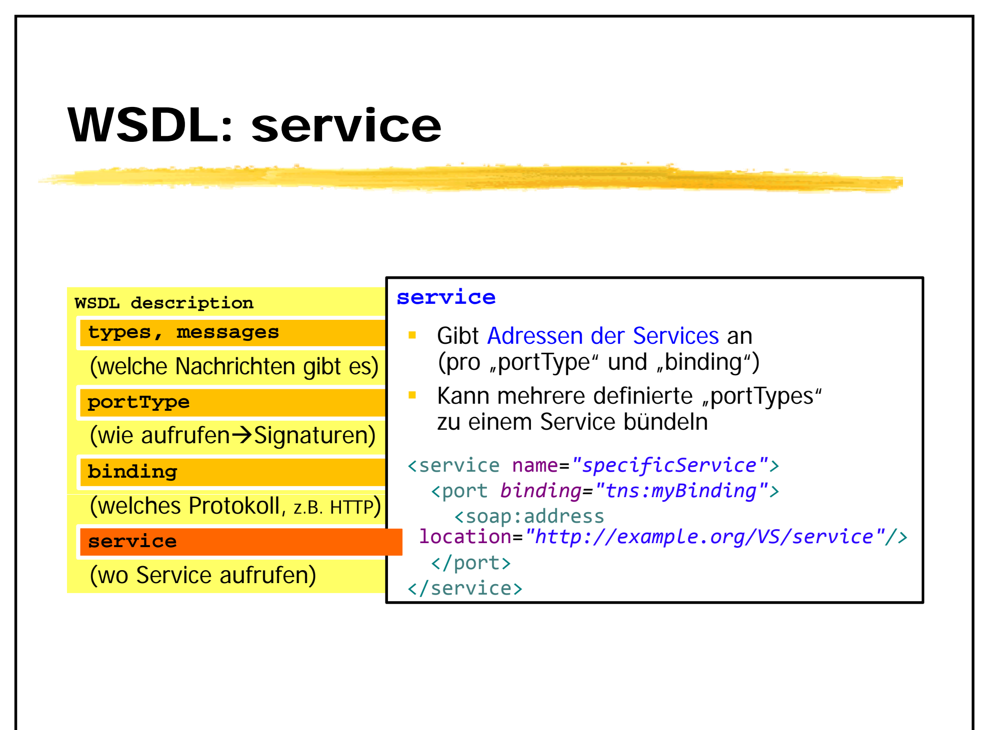
```
<portType name="groupOfServices">
  <operation name="myService">
    <input message="tns:myRequest"/>
    <output message="tns:myResponse"/>
    <fault message="tns:someFault"/>
  </operation>
</portType>
```

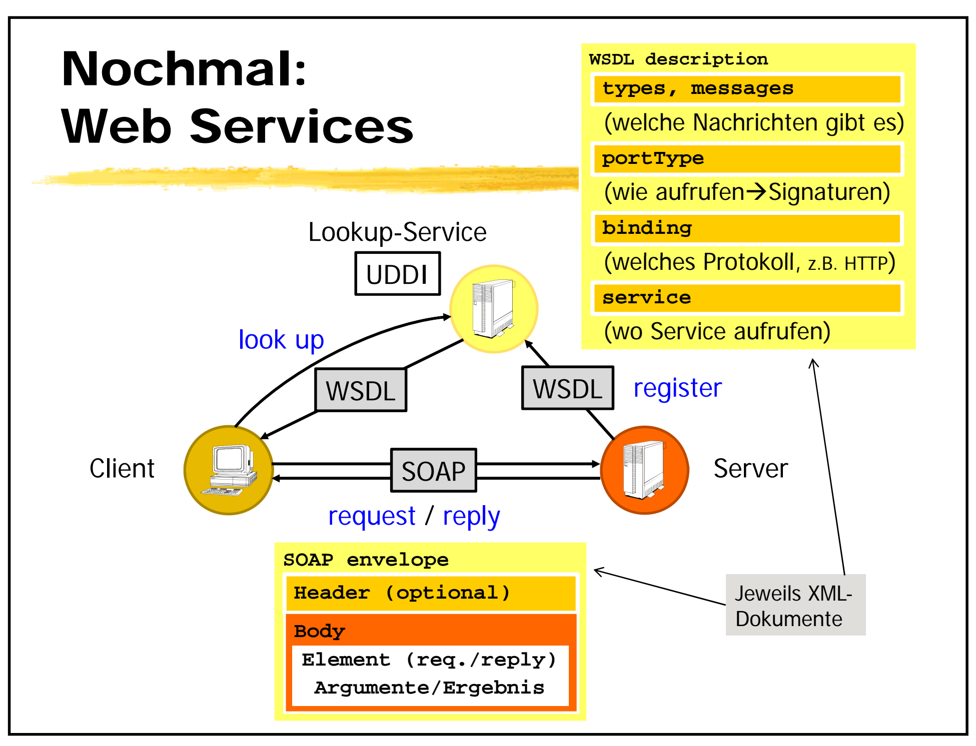
Hier könnten weitere „operation" stehen

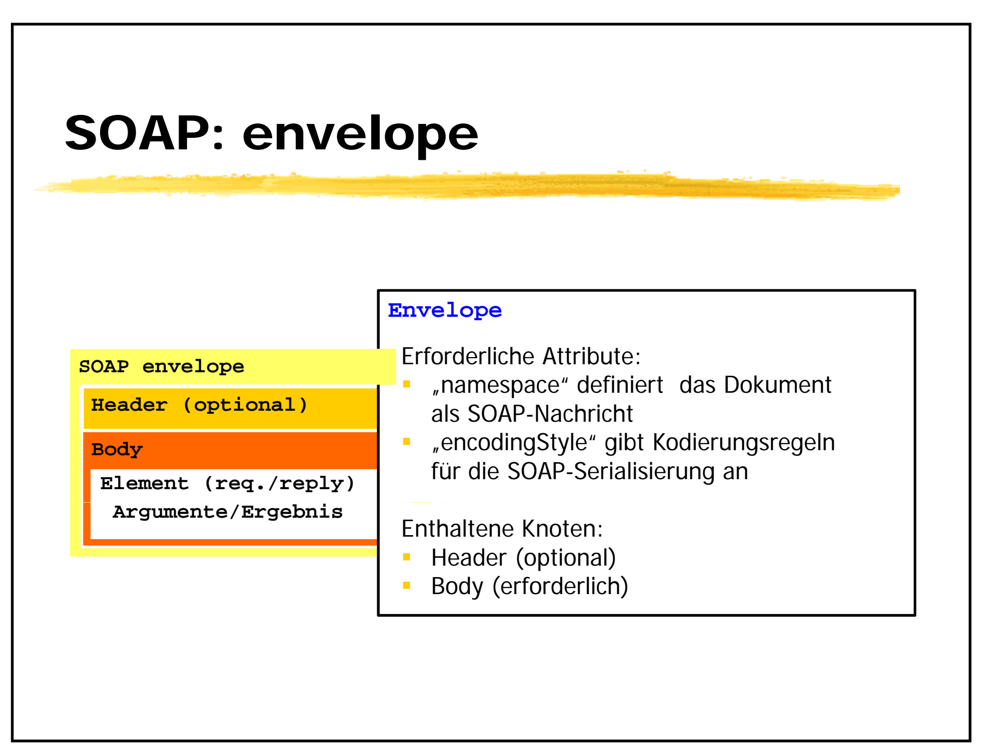
„myService" wird also mit einem int und einem float aufgerufen

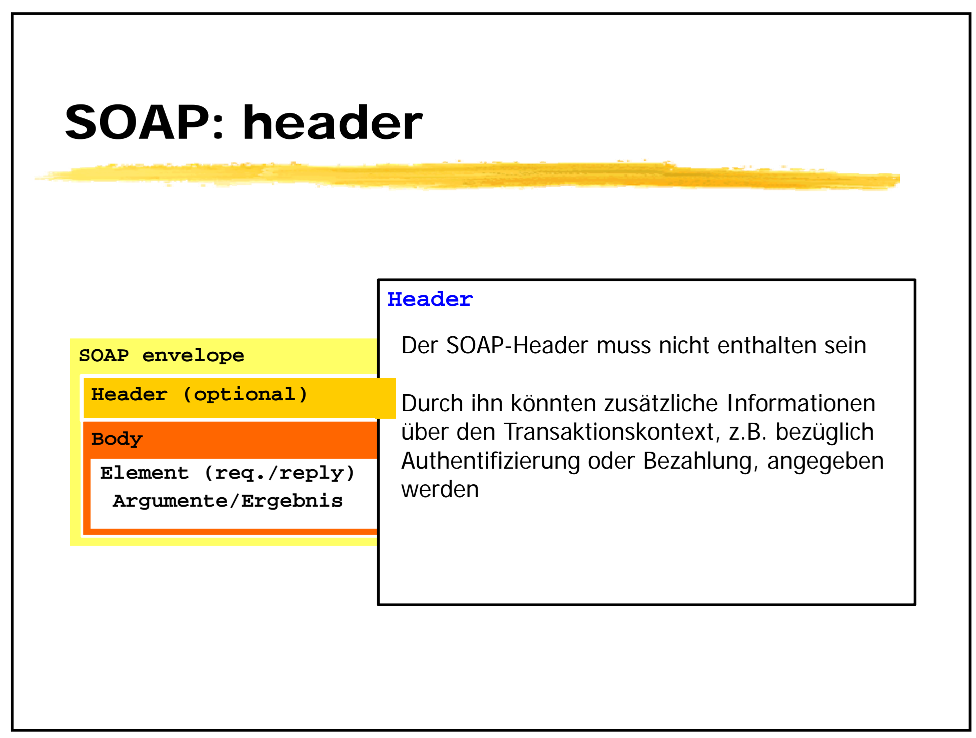


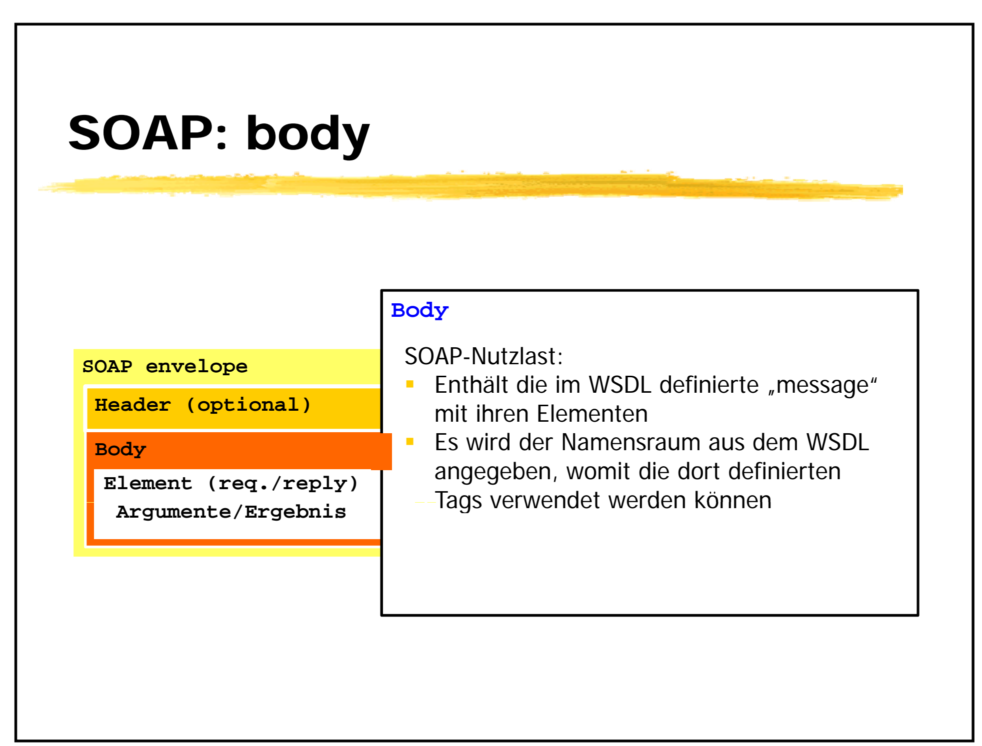

WSDL-Beispiel: binding
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)
binding
<binding name="myBinding"
type="tns:groupOfServices">
<soap:binding
transport="http://schemas.xmlsoap.org/soap/http"
style="document"/>
</binding>
Allgemeiner als „rpc"
(veralteter Web Service „style")
URI definiert HTTP als
Transportprotokoll
(mit anderen URIs
können beliebige
Protokolle zwischen
Client und Server
vereinbart werden)

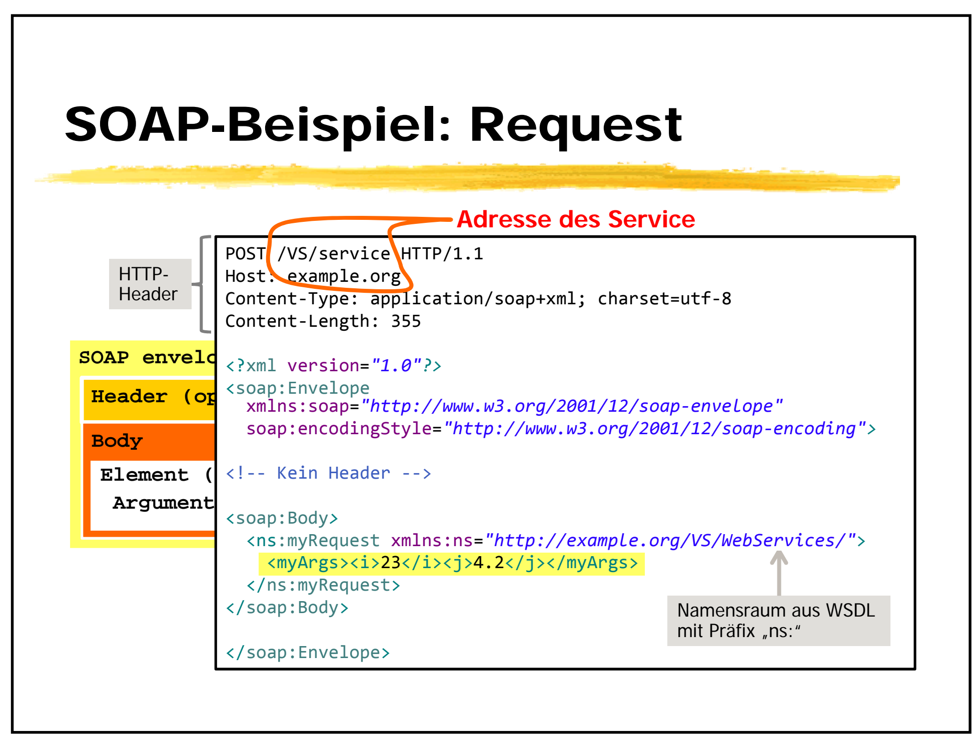

WSDL: service
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)
service
Gibt Adressen der Services an
(pro „portType" und „binding")
Kann mehrere definierte „portTypes"
zu einem Service bündeln
<service name="specificService">
<port binding="tns:myBinding">
<soap:address
location="http://example.org/VS/service"/>
</port>
</service>


Nochmal:
Web Services
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)
Lookup-Service
UDDI
look up
WSDL
WSDL
register
Client
SOAP
Server
request / reply
SOAP envelope
Header (optional)
Body
Element (req./reply)
Argumente/Ergebnis
Jeweils XML-
Dokumente


SOAP: envelope
SOAP envelope
Header (optional)
Body
Element (req./reply)
Argumente/Ergebnis
Envelope
Erforderliche Attribute:
„namespace“ definiert das Dokument als SOAP-Nachricht
„encodingStyle“ gibt Kodierungsregeln für die SOAP-Serialisierung an
Enthaltene Knoten:
Header (optional)
Body (erforderlich)


SOAP: header
SOAP envelope
Header (optional)
Body
Element (req./reply)
Argumente/Ergebnis
Header
Der SOAP-Header muss nicht enthalten sein
Durch ihn könnten zusätzliche Informationen über den Transaktionskontext, z.B. bezüglich Authentifizierung oder Bezahlung, angegeben werden


SOAP: body
SOAP envelope
Header (optional)
Body
Element (req./reply)
Argumente/Ergebnis
Body
SOAP-Nutzlast:
Enthält die im WSDL definierte „message“ mit ihren Elementen
Es wird der Namensraum aus dem WSDL angegeben, womit die dort definierten Tags verwendet werden können



SOAP-Beispiel: Response

HTTP-Header

SOAP envelo

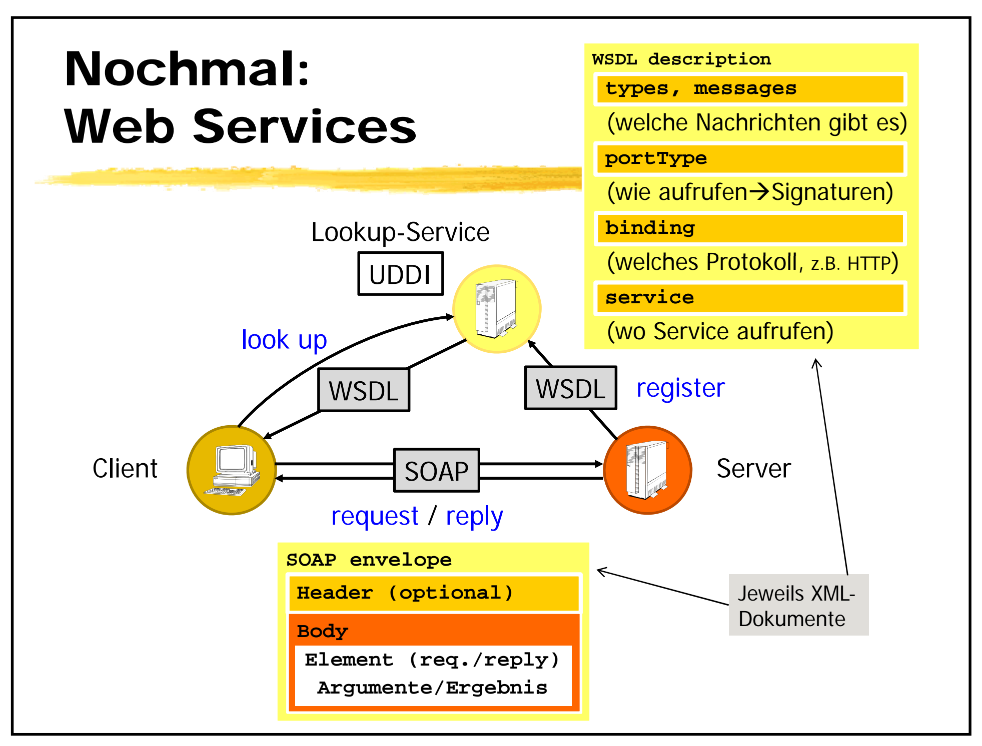
Header (op

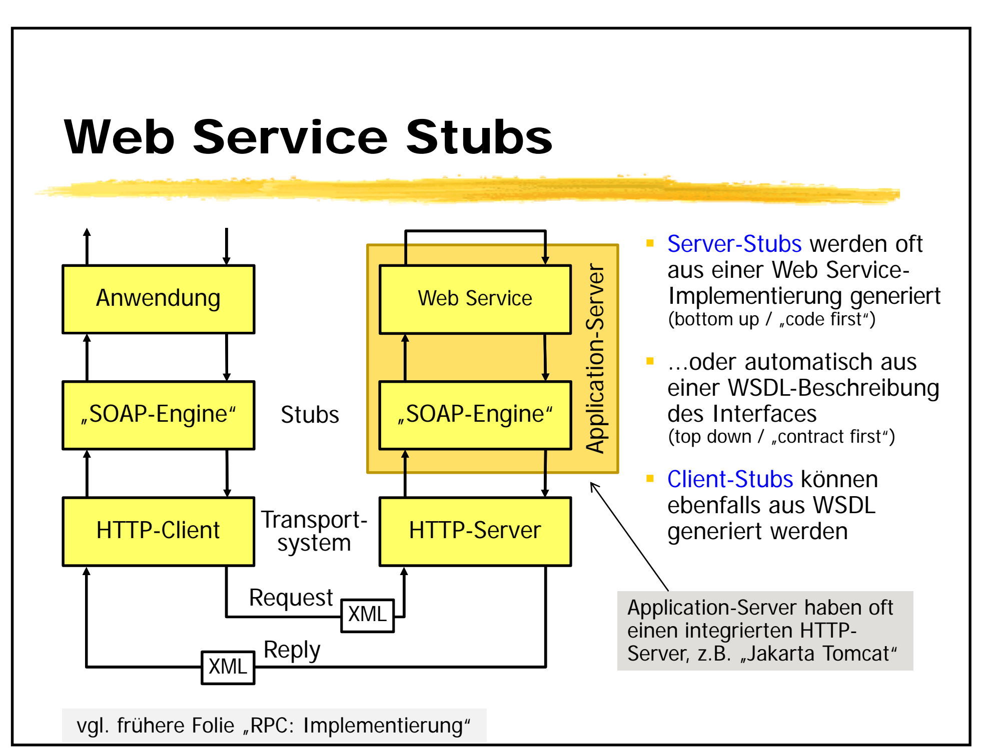
Body

Element (

Argument

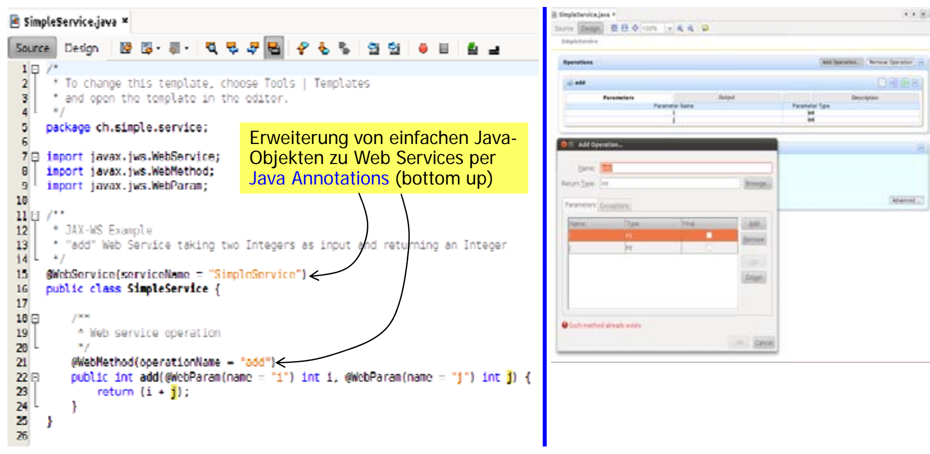
```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 340

<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<!-- Kein Header -->

<soap:Body>
  <ns:myResponse xmlns:ns="http://example.org/VS/WebServices/">
    <myRet>27.2</myRet>
  </ns:myResponse>
</soap:Body>

</soap:Envelope>
```


Nochmal:
Web Services
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)
Lookup-Service
UDDI
look up
WSDL
WSDL
register
Client
SOAP
Server
request / reply
SOAP envelope
Header (optional)
Body
Element (req./reply)
Argumente/Ergebnis
Jeweils XML-
Dokumente


Web Service Stubs
Anwendung
„SOAP-Engine"
Stubs
HTTP-Client
Transport-
system
Web Service
„SOAP-Engine"
Application-Server
HTTP-Server
Request
XML
Reply
XML
Server-Stubs werden oft aus einer Web Service-Implementierung generiert
(bottom up / „code first")
...oder automatisch aus einer WSDL-Beschreibung des Interfaces
(top down / „contract first")
Client-Stubs können ebenfalls aus WSDL generiert werden
Application-Server haben oft einen integrierten HTTP-Server, z.B. „Jakarta Tomcat"
vgl. frühere Folie „RPC: Implementierung"

Entwicklung von Web Service-Komponenten mit Java-IDE

- JAX-WS: Java API for XML Web Services (Beispiel: NetBeans)



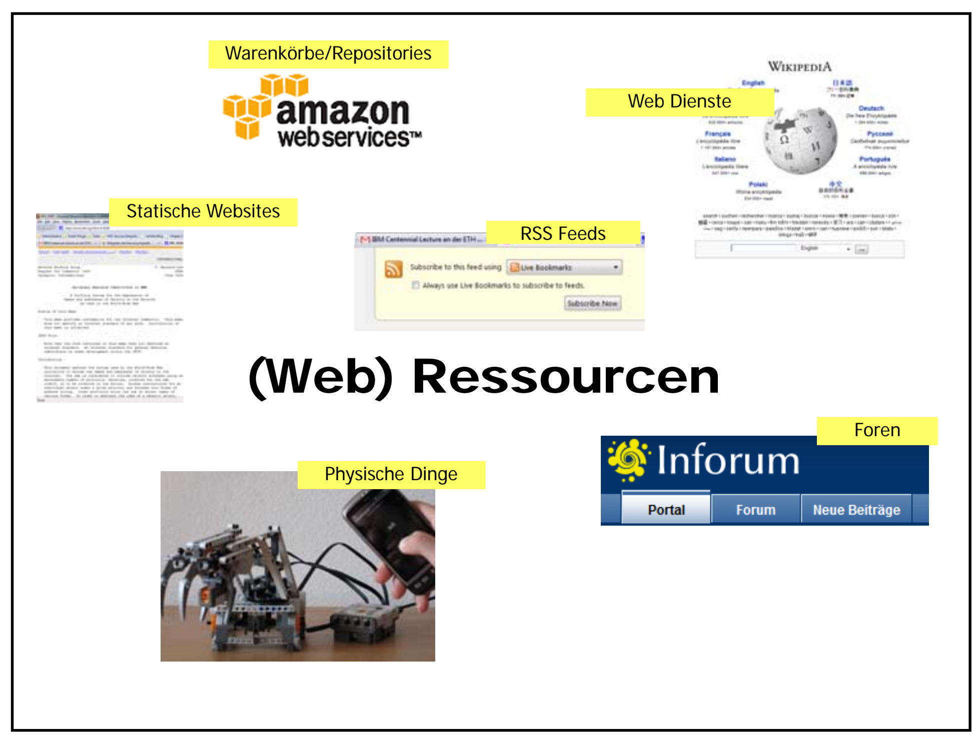
Ressourcen-orientierte Architektur (ROA)

- Funktionalität wird nicht durch Services („SOA“), sondern durch (Web) Ressourcen angeboten
- Ressource? Referent eines Uniform Resource Identifiers
  - RFC 1630 „URL“ (1994) — Implizit: „Etwas, das adressiert werden kann“
  - RFC 2396 „URI“ (1998)

    > *A resource can be anything that has identity. Familiar examples include an electronic document, an image, a service (e.g., "today's weather report for Los Angeles"), and a collection of other resources. Not all resources are network "retrievable"; e.g., human beings, corporations, and bound books in a library can also be considered resources...*

  - RFC 3986 „URI“ (2005)

    > *...Likewise, abstract concepts can be resources, such as the operators and operands of a mathematical equation, the types of a relationship...*



Repräsentation – Beispiel „Buch“

- Das Buch als abstraktes Konzept
  - es gibt verschiedene Ausgaben, Exemplare etc.
  - identifiziert per ISBN: *URN:ISBN:978-3540002130*
- Was wir ausleihen / kaufen, ist eine Repräsentation des Buches
  - z.B. Hardcover, PDF, E-Book,...
  - auch ein Bild des Covers kann eine Repräsentation sein
  - oder ein maschinenlesbares XML-Dokument für das Bibliothekssystem

REST

- REST (als idealisierte Architektur des Web) steht für
  - Representational: Nicht Ressourcen, sondern deren Repräsentationen werden übertragen
  - State Transfer: Die Übertragung löst Zustandsübergänge aus und verändert damit die Ressourcen
- REST: Entwicklung
  - Zurückführen des Erfolgs des WWW (z.B. bzgl. Skalierbarkeit) auf Eigenschaften der verwendeten Protokolle und Mechanismen: Abstraktion von HTTP und Formulierung einer idealisierten Architektur: REST
  - Mit REST wird versucht, die Möglichkeiten, die das Web (bzw. HTTP) bietet, optimal auszunutzen (Caching; HTTP-Verben wie PUT und DELETE, die Web-Browser nicht kennen,...)

Eigenschaften von REST

- Zustandslosigkeit
  - entschärft Crash-Problematik und Orphans
  - erlaubt Caching und bessere Skalierbarkeit
- Einheitliche, a-priori bekannte, Schnittstelle für alle Ressourcen
  - einheitliche Aufrufe, z.B. GET, POST bei HTTP
  - Adressierung direkt durch URIs
  - selbstbeschreibende Nachrichten: alle benötigten Metadaten sind enthalten, z.B. im HTTP-Header
  - bekannte Repräsentationen, z.B. MIME-Typen

auch andere Protokolle möglich!

- Bevorzugte Repräsentation wählbar
  - HTML, XML, JSON,...