SOAP

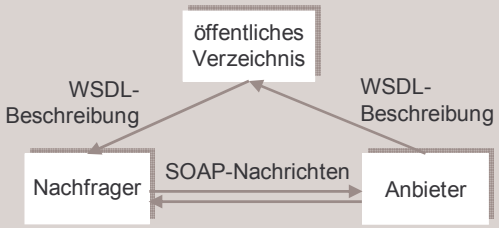
Heutige Vorlesung

SOAP im Detail

- prinzipieller Aufbau
- SOAP für RPCs
- Verarbeitung
- Übertragung
- Vor- und Nachteile

Warum SOAP?

- SOAP: Format, in dem Anwendungen Daten austauschen
- Warum spezielles Format hierfür und nicht einfach *beliebige* XML-Syntax zulassen?



- Es muss festgelegt werden wie:
  - Aufruf proc(param-1, …, param-n) in XML kodiert wird
  - Fehlermeldungen in XML kodiert werden
  - Arrays type[] und Matrizen type[][] in XML kodiert werden
- Und genau dies macht SOAP!

Prinzipieller Aufbau

Prinzipieller Aufbau


```
<?xml version='1.0' encoding='UTF-8'?>
<env:Envelope
        xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header>
     Zusatzinformationen
  </env:Header>
  <env:Body>
     Nachrichtinhalt
   </env:Body>
</env:Envelope>
```

- Wurzel-Element: Envelope aus SOAP-Namensraum
- kein W3C-Namensraum
- aktuelle Version trotzdem W3C Recommendation
- Header: optional
- Body: obligatorisch

Versionen

SOAP 1.1

W3C Note (2000)

- kein offizieller W3C-Standard, aber heute noch weit verbreitet
- wird von Google benutzt
- wird von WSDL 1.1 benutzt

SOAP 1.2

W3C Recommendation (2003)

- *einzige* Version, die vom W3C als Standard offiziell akzeptiert wurde
- Vorlesung benutzt diese Version

Zum Unterschied

- http://www.hadleynet.org/marc/whatsnew.html

SOAP-Version = Envelope-Namensraum

SOAP 1.1

- W3C-Note (2000)

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env=" http://schemas.xmlsoap.org/soap/envelope">
   …
</env:Envelope>
```

SOAP 1.2

aktuelle Version

- W3C-Standard (2003)

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/envelope">
   …
</env:Envelope>
```


Nachrichteninhalt


```
<env:Envelope …>
   <env:Body xmlns:ns="URI">
      <ns:Nachrichtinhalt-Teil-1>…</ns:Nachrichteninhalt-Teil-1>
            …
      <ns:Nachrichteninhalt-Teil-n>…</ns:Nachrichteninhalt-Teil-n>
   </env:Body>
</env:Envelope>
```

- Body: beliebige XML-Inhalte erlaubt
- Struktur von Anwendung festgelegt, z.B. durch:
  - speziellen Namensraum oder
  - WSDL-Beschreibung

Briefkopf

```
<env:Envelope …>
  <env:Header xmlns:ns="URI" >
    <ns:Zusatzinformation-1>…</ns:Zusatzinformation-1>
        …
    <ns:Zusatzinformation-n>…</ns:Zusatzinformation-n>
  </env:Header>
  <env:Body>…</env:Body>
</env:Envelope>
```

Header Blocks

- **Header**: beliebige XML-Inhalte erlaubt
- Struktur von Anwendung festgelegt
- **Header Blocks**:
  - Kind-Elemente von Header
  - Zusatzinformation zur eigentlichen Nachricht

Beispiel

```
<env:Envelope …>
  <env:Header>
    <alertcontrol xmlns="http://example.org/alertcontrol">
      <priority>1</priority>
      <expires>2003-10-12T14:00:00-05:00</expires>
    </alertcontrol>
  </env:Header>
  <env:Body>
    <alert-msg xmlns="http://example.org/alert">
      Pick up Mary at 2pm!
    </alert-msg>
  </env:Body>
</env:Envelope>
```

Zusatz-information (*Header Block*)

Nachricht

➔ Erweiterung des ursprünglichen Nachrichtenformates

Obligatorische & optionale Header Blocks

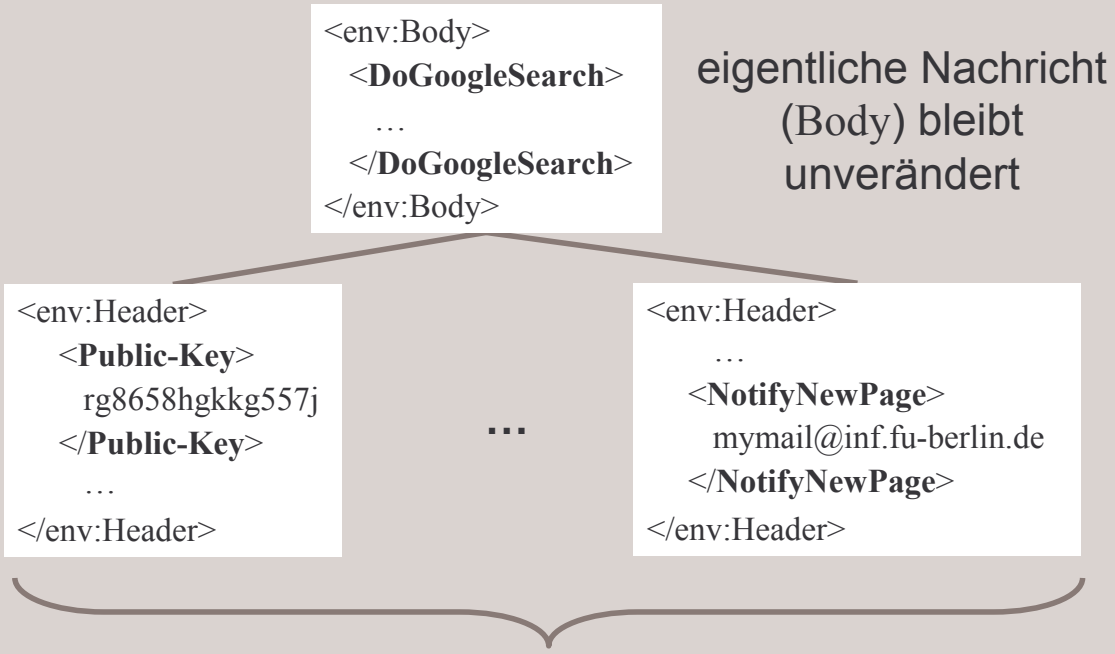

```
<env:Header>
    <alertcontrol xmlns="http://example.org/alertcontrol"
                  env:mustUnderstand="true">
        …
    </alertcontrol>
</env:Header>
```

- mustUnderstand="true": Empfänger *muss* Header Block verstehen oder mit Fehlermeldung antworten
- mustUnderstand="false": Empfänger kann Header Block (ohne Fehlermeldung) ignorieren
- kann für jeden Header Block unterschiedlich sein
- Beachte: Standard-Wert ist "false"

Erweiterbarkeit von SOAP-Nachrichten

- Konzept: Trennung der Zusatzinformationen (Header) von eigentlicher Nachricht (Body)
- Nachrichtenformat kann durch Header Blocks erweitert werden, ohne ursprüngliches Format (Body) zu modifizieren.
- Erweiterungen jeweils obligatorisch oder optional
- einzelne Erweiterungen unabhängig voneinander

Beispiel: Erweiterung von DoGoogleSearch



SOAP für RPCs

SOAP für RPCs

- SOAP auch Nachrichtenformat für entfernte Prozeduraufrufe (RPCs)
- eigentlichen RPCs werden aber von Werkzeugen wie .net oder J2EE realisiert
- SOAP selbst unterstützt nur Einweg-Kommunikation
- Anfrage-Antwort-Muster können auf verschiedene Weise realisiert werden:
  1. SOAP mit HTTP übertragen
  2. eindeutige Referenz im Briefkopf

     Vorteil: Unabhängig vom Übertragungsprotokoll

Referenz im Briefkopf

- Anfrage enthält eindeutige Referenz („Mein Zeichen"), worauf anschließend verwiesen werden kann:

```
<env:Header>
  <wsu:identifier xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility/">
        URI als eindeutige Referenz
  </wsu:identifier>
</env:Header>
```

- beliebig komplexe Interaktionen möglich
- oft verwendet, aber noch *kein* etablierter Standard
- Alternative zu wsu:identifier:

  MessageID aus Namensraum von WS-Addressing

➔ eingeschränkte Interoperabilität

Kodierung von RPCs

Procedure(In-Parameter-1="val-1",…,In-Parameter-n ="val-n")

```
<env:Envelope …>
  <env:Body>
     <m:Procedure xmlns:m="URI">
       <m:In-Parameter-1>val-1</m:In-Parameter-1>
          …
       <m:In-Parameter-n>val-n</m:In-Parameter-n>
     </m:Procedure>
  </env:Body>
</env:Envelope>
```

- Name der Prozedur Kind-Element von Body
- Eingangsparameter Kind-Elemente der Prozedur
- Beachte: Reihenfolge der Parameter egal!
- Beachte: grundsätzlich *Call-by-Value* !

Kodierung von RPCs

Wo soll die Prozedur aufgerufen werden (URI)?

- außerhalb von SOAP im Transportprotokoll spezifiziert
- kann aber auch explizit im Briefkopf angegeben werden:

```
<env:Header>
  <wsa:EndpointReference
          xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing">
     <wsa:Address>http://api.google.com/search/beta2</wsa:Address>
     <wsa:PortType>ns1:GoogleSearchPort</wsa:PortType>
  </wsa:EndpointReference>
</env:Header>
```

- WS-Addressing allerdings *kein* etablierter Standard

Kodierung des Ergebnisses eines RPCs

public Out-Parameter-i Procedure(…)

```
<env:Envelope …>
  <env:Body>
     <m:ProcedureResponse xmlns:m="URI"
              xmlns:rpc="http://www.w3.org/2003/05/soap-rpc">
       <rpc:result>m:Out-Parameter-i</rpc:result>
       <m:Out-Parameter-1>…</m:Out-Parameter-1>
          …
       <m:Out-Parameter-n>…</m:Out-Parameter-n>
    </m:ProcedureResponse>
  </env:Body>
</env:Envelope>
```

- Beachte: Name für *ProcedureResponse* beliebig
- Kind-Elemente von *ProcedureResponse*: Ausgangsparameter und eigentliches Ergebnis

Kodierung des Ergebnisses eines RPCs

public Out-Parameter-i Procedure(…)

```
<env:Envelope …>
  <env:Body>
     <m:ProcedureResponse xmlns:m="URI"
              xmlns:rpc="http://www.w3.org/2003/05/soap-rpc">
       <rpc:result>m:Out-Parameter-i</rpc:result>
       <m:Out-Parameter-1>…</m:Out-Parameter-1>
          …
       <m:Out-Parameter-n>…</m:Out-Parameter-n>
    </m:ProcedureResponse>
  </env:Body>
</env:Envelope>
```

- In-Out-Parameter erscheinen im Aufruf *und* der Antwort
- rpc:result: Verweis auf ausgezeichnetes Ergebnis (optional)
- Namensraum **…/soap-rpc** Teil der SOAP-Spezifikation

Kodierung von Fehlermeldungen

```
<env:Envelope …>
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Sender</env:Value>
      </env:Code>
      <env:Reason>
        <env:Text xml:lang="en-US">Processing error</env:Text>
        <env:Text xml:lang="cs">Chyba zpracování</env:Text>
      </env:Reason>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

- Code und Reason obligatorisch
- Code: für maschinelle Verarbeitung
- Reason: zusätzliche Information, nicht für maschinelle Verarbeitung

Kodierung von Fehlermeldungen: Reason

```
<env:Envelope …>
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Sender</env:Value>
      </env:Code>
      <env:Reason>
        <env:Text xml:lang="en-US">Processing error</env:Text>
        <env:Text xml:lang="cs">Chyba zpracování</env:Text>
      </env:Reason>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

- mindestens ein Text-Element
- xml:lang-Attribut obligatorsich

Kodierung von Fehlermeldungen: Code


```
<env:Envelope …>
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Sender</env:Value>
      </env:Code>
      <env:Reason>
        <env:Text xml:lang="en-US">Processing error</env:Text>
        <env:Text xml:lang="cs">Chyba zpracování</env:Text>
      </env:Reason>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

- Value obligatorsich
- env:Sender: Anfrage nicht korrekt, nochmalige korrigierte Anfrage erwartet

Kodierung von Fehlermeldungen: Subcode


```
<env:Envelope … xmlns:rpc="http://www.w3.org/2003/05/soap-rpc">
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Sender</env:Value>
        <env:Subcode>
          <env:Value>rpc:BadArguments</env:Value>
        </env:Subcode>
      </env:Code>
      <env:Reason>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

- Subcode optional
- Subcode genauso strukturiert, wie Code: Value obligatorisch, Subcode optional
- rpc:BadArguments: standardisierter Fehlercode

Kodierung von Arrays und Matrizen

- Beispiel: Als Parameter soll Array int[3] von drei Zahlen übergeben werden.
- Wie soll dieses Array dargestellt werden?

so?

```
<array>
  <number xsi:type="xsd:int">108</number>
  <number xsi:type="xsd:int">99</number>
  <number xsi:type="xsd:int">205</number>
</array>
```

oder so?

```
<array elementType="xsd:int">
  108 99 205
</array>
```

- Und wie Matrizen (multidimensionale Arrays) darstellen?

SOAP-Arrays


```
<numbers xmlns:enc="http://www.w3.org/2003/05/soap-encoding"
         enc:itemType="xsd:int" enc:arraySize="3">
  <number>1</number>
  <number>2</number>
  <number>2</number>
</numbers>
```

- Array int[3] mit zwei Elementen vom Typ xsd:int
- Element-Namen (hier numbers und number) beliebig, entscheidend sind Attribute enc:itemType und enc:arraySize
- Namensraum **…/soap-encoding** Teil der SOAP-Spezifikation
- enc:arraySize="*": Array int[] mit beliebig vielen Elementen

Mehrdimensionale SOAP-Arrays

```
<numbers enc:itemType="xsd:int" enc:arraySize="3 2">
  <number>1</number>      ➔ a1 b1
  <number>2</number>        ➔ a2 b1
  <number>3</number>          ➔ a3 b1
  <number>4</number>            ➔ a1 b2
  <number>5</number>              ➔ a2 b2
  <number>6</number>                ➔ a3 b2
</numbers>
```

b

a

- 3x2-Matrix mit Elementen vom Typ xsd:int.
- enc:arraySize="* 2": nx2-Matrix
- Beachte: * nur an erster Stelle erlaubt

⇨ eindeutig auflösbar

Beispiel enc:arraySize="* 2"

```
<numbers enc:itemType="xsd:int" enc:arraySize="* 2">
  <number>1</number>      ➔ a1 b1
  <number>2</number>        ➔ a2 b1
  <number>3</number>          ➔ a3 b1
  <number>4</number>            ➔ a1 b2
  <number>5</number>              ➔ a2 b2
  <number>6</number>                ➔ a3 b2
</numbers>
```

b

a

- #Elemente = 6 = n x 2

⇨ eindeutige Lösung: n = 3

- für #Elemente = 6 = n x m gäbe es keine eindeutige Lösung

⇨ enc:arraySize="* *" nicht erlaubt

env:encodingStyle

- die vorgestellte Kodierung für RPCs und Arrays *muss* nicht verwendet werden
- wird sie verwendet, dann in SOAP-Nachricht folg. Kodierungsschema angeben:

```
env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
```

- Kodierungsschema kann anwendungsspezifisch sein:

```
env:encodingStyle="http://www.ibm.com/soap-encoding"  (fiktiv)
```

- Anwendung muss dann entspr. Kodierungsschema kennen

Beispiel Google

```
<?xml version='1.0' encoding='UTF-8'?>
<env:Envelope
          xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Body>
    <doGoogleSearch xmlns="urn:GoogleSearch"
          env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <key xsi:type="xsd:string">3289754870548097</key>
      <q xsi:type="xsd:string">Eine Anfrage</q>
      <start xsi:type="xsd:int">0</start>
      <maxResults xsi:type="xsd:int">10</maxResults>
        …
    </doGoogleSearch>
  </env:Body>
</env:Envelope>
```

Verarbeitung von SOAP-Nachrichten

Allgemeine Anforderungen

Empfänger *muss* verarbeiten

- Body
- Header Blocks mit mustUnderstand="true"

Empfänger *darf* ignorieren

- Header Blocks mit mustUnderstand="false"
- Header Blocks ohne mustUnderstand-Attribut
  Grund: "false" Standardwert von mustUnderstand

Schrittweise Verarbeitung

SOAP unterstützt die schrittweise Verarbeitung von Nachrichten, z.B.:

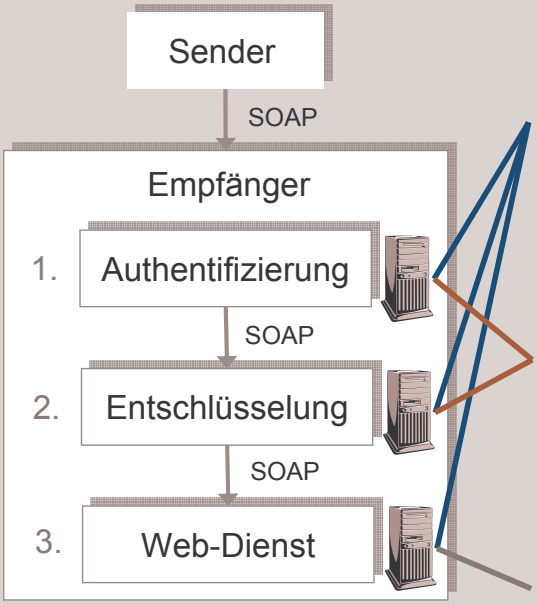
1. Authentifizierung: Verifizierung einer digitalen Signatur in einem Header Block
2. Entschlüsselung des Body
3. Aufruf des eigentlichen Web-Dienstes

Vorteile der schrittweisen Verarbeitung

Aufgabenteilung

- spezialisierte Server
- z.B. Authentifizierungs-Server und Server, auf dem der eigentliche Dienst läuft
- Authentifizierungs-Server kann *vor* der Firewall liegen, die anderen Server *hinter* der Firewall

SOAP-Knoten



SOAP-Knoten: Rechner, die Teil einer SOAP-Nachricht verarbeiten

Zwischenknoten (*intermediary*)

Endknoten (*ultimate receiver*)

Identifizierung von SOAP-Knoten

- SOAP-Knoten werden mit URIs identifiziert
- zwei Möglichkeiten:

1. **spezifische URI**

- z.B. www.example.org/Log
- muss von verarbeitenden SOAP-Knoten interpretiert werden können

2. **standardisierte URI**

- **http://www.w3.org/2003/05/envelope/role/next**
  aktueller Empfänger (Zwischen- oder Endknoten)
- **http://www.w3.org/2003/05/envelope/role/ultimateReceiver**
  Endknoten

Festlegung der Zuständigkeiten

```
<env:Header>
  <oneBlock env:role="www.example.org/Log">
    ...
  </oneBlock>
  <anotherBlock
        env:role="http://www.w3.org/2003/05/envelope/role/next">
    ...
  </anotherBlock>
  <aThirdBlock>
        ...
  </aThirdBlock>
</env:Header>
```

➔ „Log“ zuständig

➔ aktueller Empfänger zuständig

➔ Endknoten zuständig

- role: zuständiger SOAP-Knoten (URI)
- Beachte: fehlt role-Attribut, dann ist Endknoten zuständig

Aufgabe eines Zwischenknotens

1. verarbeitet Header Blocks mit
   - role=" http://www.w3.org/2003/05/envelope/role/**next**"
   - role="*URI*", wobei "*URI*" den betreffenden Zwischenknoten bezeichnet

   alle anderen Header Blocks und Body werden nicht verarbeitet
2. löscht alle verarbeiteten Header Blocks !
3. fügt evtl. neue Header Blocks hinzu
4. entscheidet, welcher SOAP-Knoten nächster Knoten in Verarbeitungsskette sein soll
5. leitet modifizierte SOAP-Nachricht an diesen SOAP-Knoten weiter

Tiefe der Verarbeitung

- Tiefe der Verarbeitung durch mustUnderstand-Attribut bestimmt:

mustUnderstand="false"

- Empfänger kann Header Block ignorieren
- Löschen ohne Header Block zu verstehen erlaubt

mustUnderstand="true"

- Empfänger muss Header Block verstehen, ansonsten Fehlermeldung
- Löschen ohne Header Block zu verstehen *nicht* möglich

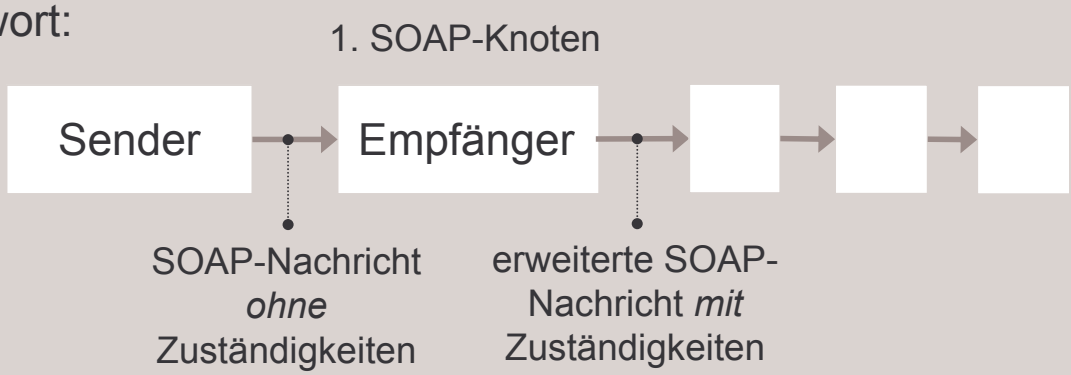
Aufgabe des Endknoten

1. verarbeitet Header Blocks
   - mit role="http://www.w3.org/2003/05/envelope/role/**ultimateReceiver**"
   - mit role=" http://www.w3.org/2003/05/envelope/role/**next**"
   - ohne role-Attribut
2. verarbeitet Body (den er auch verstehen muss)

Festlegung der Zuständigkeiten in SOAP

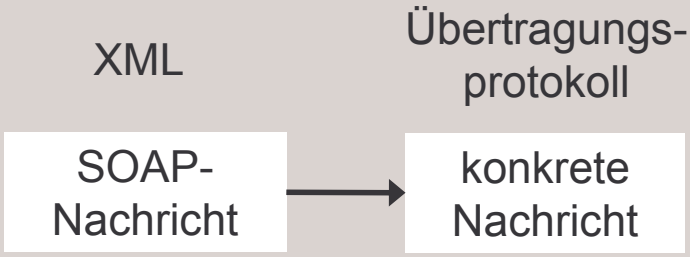
- Vorteile der schrittweisen Verarbeitung klar:
  Aufgabenverteilung auf spezialisierte Server
- Warum aber Zuständigkeiten in *SOAP-Nachricht* festlegen?
- Zuständigkeiten sollten doch für Sender *transparent* (nicht sichtbar) sein!

Antwort:



Übertragung von SOAP-Nachrichten

Protokoll-Bindung (*Binding*)



- Spezifikation, wie SOAP-Nachrichten mit einem bestimmten Protokoll (z.B. HTTP POST) übertragen werden
- beschreibt also, wie SOAP-Nachrichten serialisiert (kodiert) werden.
- SOAP-Spezifikation schreibt *nicht* vor, *wie* Protokoll-Bindung spezifiziert wird

Anforderungen an Protokoll-Bindungen

- konkrete Nachricht meist XML, kann aber auch beliebig anderes Format sein:

  z.B. komprimiertes Binärformat
- Bedingung: Serialisierung *ohne* Informationsverlust
- Serialisierung *s* also symmetrisch:

  $s^{-1}(s(N)) = N$, für alle SOAP-Nachrichten N

Standardisierte Protokoll-Bindungen

- HTTP-Binding bisher als *einzige* Protokoll-Bindung für SOAP standardisiert
- zwei unterschiedliche HTTP-Bindungen:
  - für HTTP-POST
  - für HTTP-GET

Zur Erinnerung!

HTTP

- Anfrage-Antwort-Muster
- zustandsloses Protokoll
  - jedes Anfrage-Antwort-Paar isolierte, abgeschlossene Einheit
  - Daten von vorherigen Anfragen oder Antworten stehen später nicht mehr zur Verfügung

Die wichtigsten HTTP-Formate

HTTP GET

- fragt Web-Ressource mit einer URL ab
- Parameter können in URL kodiert werden, z.B.:
- http://google.com/doGoogleSearch?q=Beginning+XML
- entspricht

  http://google.com/doGoogleSearch(q="Beginning XML")

HTTP POST

- fragt Web-Ressource ab, übermittelt gleichzeitig Daten als Anhang

SOAP über HTTP POST: Anfrage


```
POST /search/beta2/doGoogleSearch HTTP/1.1
Host: api.google.com
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn
<?xml version='1.0' encoding='UTF-8'?>
<env:Envelope …>
 <env:Body>
   <doGoogleSearch xmlns="urn:GoogleSearch">
     <key xsi:type="xsd:string">3289754870548097</key>
     <q xsi:type="xsd:string">Eine Anfrage</q>
       …
   </doGoogleSearch>
 </env:Body>
</env:Envelope>
```

SOAP über HTTP POST: Antwort


```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope …>
    <env:Body>
      <ns1:doGoogleSearchResponse xmlns:ns1="urn:GoogleSearch"
            env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <return xsi:type="ns1:GoogleSearchResult">…</return>
    </ns1:doGoogleSearchResponse>
    </env:Body>
</env:Envelope>
```


SOAP über HTTP GET


```
GET /search/beta2/doGoogleSearch?q=Beginning+XML HTTP/1.1
Host: api.google.com
Accept: application/soap+xml
```

- ruft doGoogleSearch(q="Beginning XML") auf
- gesamte SOAP-Nachricht als URL kodiert
- Antwort: SOAP (XML)
- Amazon bietet HTTP-GET-Schnittstelle an, Google jedoch *nicht*

HTTP POST vs. HTTP GET

- REST-Architektur des WWW (Fielding 2000):
  jede Web-Ressource soll eindeutig über eine URI identifiziert werden
- Beispiel: online gebuchte Reise ist eine Web-Ressource
- gebuchte Reise sollte daher auch über eine URI eindeutig angefragt werden

REST oder nicht?

HTTP GET

➔ entspricht REST-Grundsatz

- würde
  travel.com/Reservations/itinerary?reservationCode=FT35ZBQ
  oder so ähnlich anfragen
- URL identifiziert eindeutig gebuchte Reise

HTTP POST

➔ entspricht *nicht* REST-Grundsatz

- hingegen würde
  travel.com/Reservations/
  anfragen mit
  itinerary(reservationCode="FT35ZBQ")
  als SOAP-RPC im Anhang
- URL identifiziert *nicht* gebuchte Reise

HTTP POST vs. HTTP GET

- SOAP-Spezifikation empfiehlt HTTP GET, wenn Parameter Web-Ressourcen identifizieren

⇨ Übereinstimmung mit REST-Grundsatz

- Problem: Wie komplexe Parameter als URI kodieren?
- Beispiel: reservationCode könnte aus String und Datum bestehen
- so kodieren?

  travel.com/Reservations/itinerary?reservationCode=FT35ZBQ+22/6/2005
- oder so?

  travel.com/Reservations/itinerary?reservationString=FT35ZBQ reservationDate=22/6/2005

Vor- und Nachteile von SOAP

Vorteile

+ etablierter Standard, wird u.a. in .net verwendet
+ unabhängig von Übertragungsprotokollen
+ sowohl für RPCs als auch für Messaging geeignet
+ RPCs über Firewalls hinweg möglich
+ einfach erweiterbar
+ Erweiterungen unabhängig voneinander

Nachteile

- RPCs über Firewalls hinweg oft *nicht* erwünscht
- zusätzlicher Verarbeitungsaufwand
- für viele notwendige Erweiterungen noch kein etablierter Standard

  Beispiel: wsu:identifier vs. wsa:MessageID
- Protokoll-Bindungen können unterschiedliche Semantik haben

  Beispiel: SMTP-Binding asynchron, HTTP-Binding synchron
- heutzutage noch *nicht* vollständig interoperabel

  ➔ http://www.ws-i.org

Wie geht es weiter?

heutige Vorlesung

- ☑ SOAP im Detail

heutige Übung

- Datenmodellierung mit XML
- Lose Kopplung mit Messaging

Vorlesung nächste Woche

- WSDL im Detail