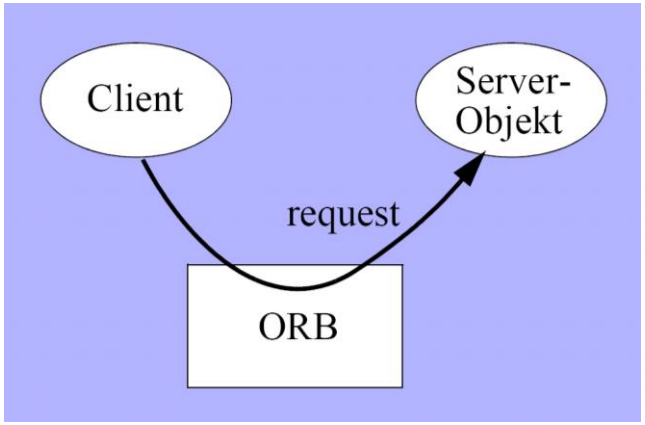
Middleware, Web Services

Middleware-Entwicklung, die Jini historisch voranging

1. RPC-Bibliotheken (z.B. von SUN für UNIX)
   - Unterstützung des Client-Server-Paradigmas
   - einfache Schnittstellenbeschreibungssprache, Stubgeneratoren
   - erste Sicherheitskonzepte: Authentifizierung, Autorisierung, Verschlüsselung
2. Client-Server-Verteilungsplattformen (z.B. DCE)
   - Verzeichnisdienst, globaler Namensraum, globales Dateisystem
   - Programmierunterstützung u.a. für Multithreading und Synchronisation
3. Objektbasierte Verteilungsplattformen (z.B. CORBA)
   - Kooperation zwischen verteilten Objekten
   - objektorientierte Schnittstellenbeschreibungssprache
   - „Object Request Broker" als Vermittlungsinstanz

CORBA

- Common Object Request Broker Architecture
  - ORB (Object Request Broker) als Vermittlungsinfrastruktur (Weiterleitung von Methodenaufrufen etc.)
  - IDL (Interface Description Language) mit Stub-Generatoren
  - Systemfunktionen und Basisdienste in Form von Object Services (z.B. Semaphore, Persistierung)



Methodenaufruf unterschiedlicher Semantik:

synchron (mit Rückgabewerten; analog zu RPC)

verzögert synchron (Aufrufer wartet nicht auf das Ergebnis, sondern holt es sich später ab)

one way (asynchron: Aufrufer wartet nicht; keine Ergebnisrückgabe)

CORBA

- Ab ca. 2000 entstand der Wunsch nach einer wesentlichen Erweiterung der CORBA-Funktionalität aufgrund
  - neuer Anforderungen durch grosse E-Commerce-Anwendungen
  - Ausbreitung des WWW
  - Aufkommen von Java
  - Aufkommen mobiler Geräte

Lehrreich diesbezüglich: Michi Henning: *The rise and fall of CORBA. Commun. ACM, Vol. 51, No. 8 (Aug. 2008), 52-57*

- Allerdings geriet die CORBA-Weiterentwicklung ins Stocken
  - zu weitreichende Anforderungen → komplex / ineffizient
  - man versuchte, es jedem Recht zu machen (widersprüchliche Interessen, barocke Konstrukte durch Kompromisse)
  - kommerzielle Implementierungen der Erweiterungen nur zögerlich
  - fehlende Unterstützung durch Microsoft (→ eigene Architektur: DCOM und .NET) aufkommende
  - Konkurrenzsysteme (z.B. Web Services), die z.T. direkter und besser an die neuen Anforderungen angepasst waren

Web Services als Beispiel für das Client-Server-Modell

- Problem: Internet ist zu heterogen für eine einheitliche Programmiersprache oder RPC-Laufzeitumgebung
  - „Lösung“: Web Services als offener, plattform- bzw. sprach-unabhängiger Standard, bei dem nur die Schnittstellen definiert werden und von diversen Plattformen implementiert werden können

---

- HTTP fungiert 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 typischer HTTP-Timeout dauert

Web Services als Beispiel für das Client-Server-Modell

- Problem: Internet ist zu heterogen für eine einheitliche Programmiersprache oder RPC-Laufzeitumgebung
  - „Lösung“: Web Services als offener, plattform- bzw. sprach-unabhängiger Standard, bei dem nur die Schnittstellen definiert werden und von diversen Plattformen implementiert werden können

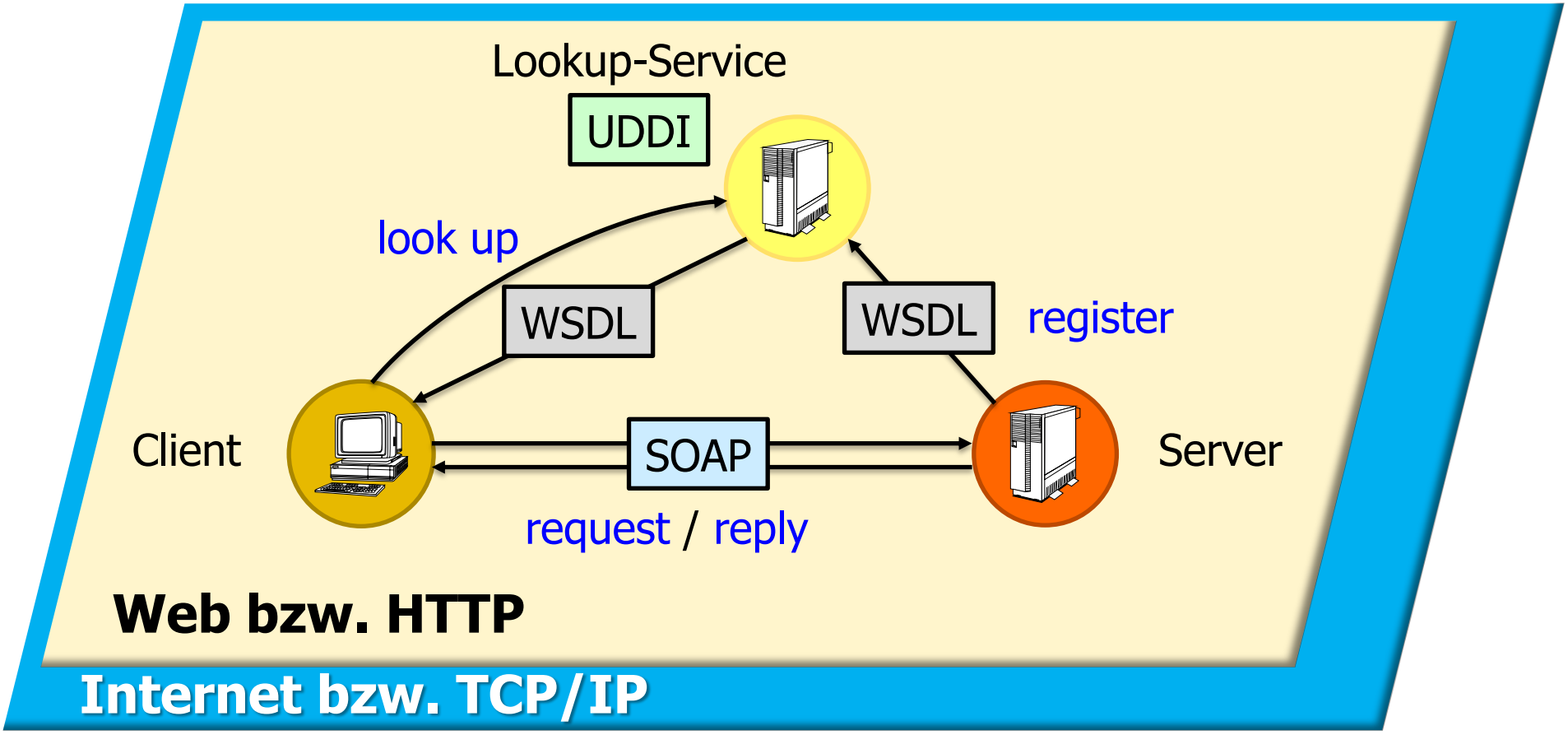
---

XML (Extensible Markup Language):

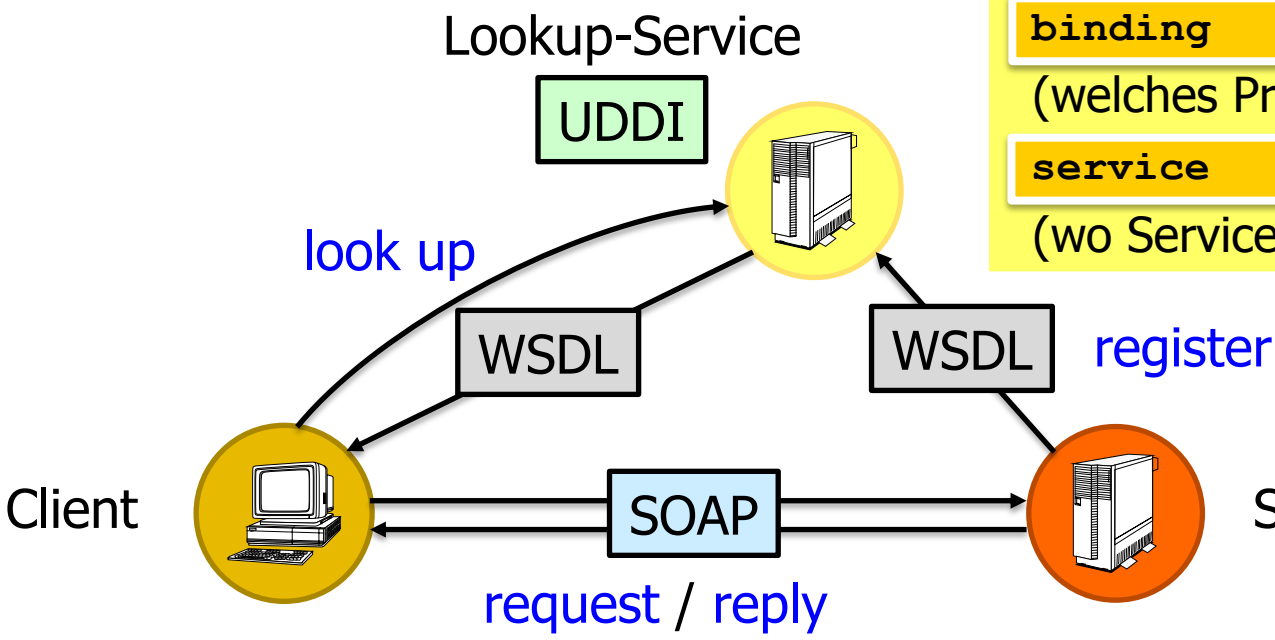
- Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdaten (z.B. ASCII, UTF-8)
- Unterstrukturen mit Start- (<Tag-Name>) und End-Tag (</Tag-Name>) oder einem Empty-Element-Tag (<*Tag-Name* />)
- Attribute bei einem Tag (Attribut-Name= "Attribut-Wert") für Zusatzinformationen

```
<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>
```

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)

SOAP envelope

- **Header (optional)**
- **Body**
  - **Element (req./reply)**
    Argumente/Ergebnis

Jeweils XML-Dokumente

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

- Definition von Typen (bzw. deren Namen) als XML-„element“
- Meistens als Verweis auf einen „complexType“, der aus „elements“ der Basistypen (int, float,…) besteht
- Das 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

```
<?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>
...
```

Namensraum der hier beschriebenen Web Services

Definiert Präfix „tns:“, um den targetNamespace anzusprechen

Weitere Präfixe für Namensräume, aus denen Tags benötigt werden

Importiert die Typendefinitionen

nächste slide

WSDL-Namensräume

- ExampleSchema.xsd

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema
  version="1.0"
  targetNamespace="http://example.org/VS/WebServices/"
  xmlns:tns="http://example.org/VS/WebServices/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- Elementdefinitionen -->

</xs:schema>
```

Hier gleicher Namensraum wie Services

Definiert Präfix „tns:", um den targetNamespace einfacher anzusprechen

Lässt „xs:" auf den allgemeinen XML-Schema-Namensraum zeigen (vgl. „xsd" im WSDL Dokument)

nächste slide

WSDL-Beispiel: types

Diese Namen sind nun Teil des Namensraums

„type“ kommt aus dem Namens-raum der definierten Services

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)

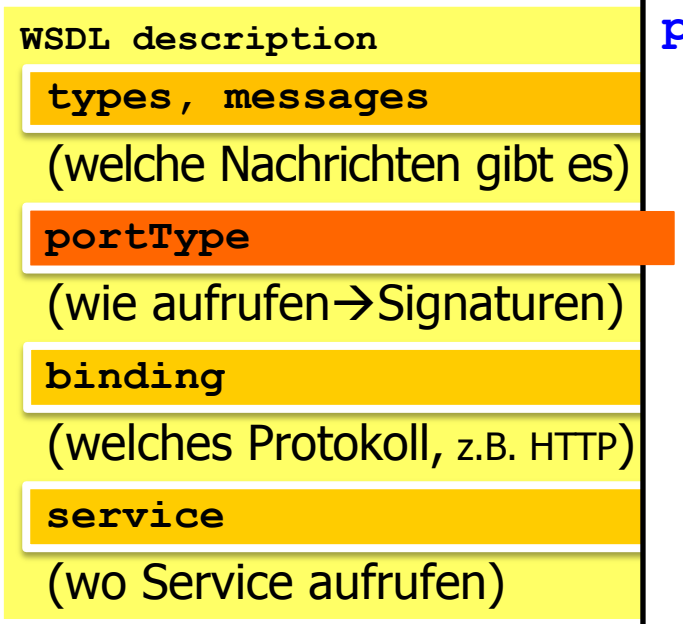
messages

```
<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>
```

„message“ kann aus null oder mehr „parts“ bestehen

„myRequest“ hat also einen int und einen float als Parameter

WSDL: portType



portType

- Definition der einzelnen Service Methoden
- Jede Methode entspricht einer „operation"
  - hat typischerweise eine „input"-Nachricht und eine „output"-Nachricht
  - zusätzlich können Fehlerbenachrich-tigungen angegeben werden („fault")
- Methoden 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)

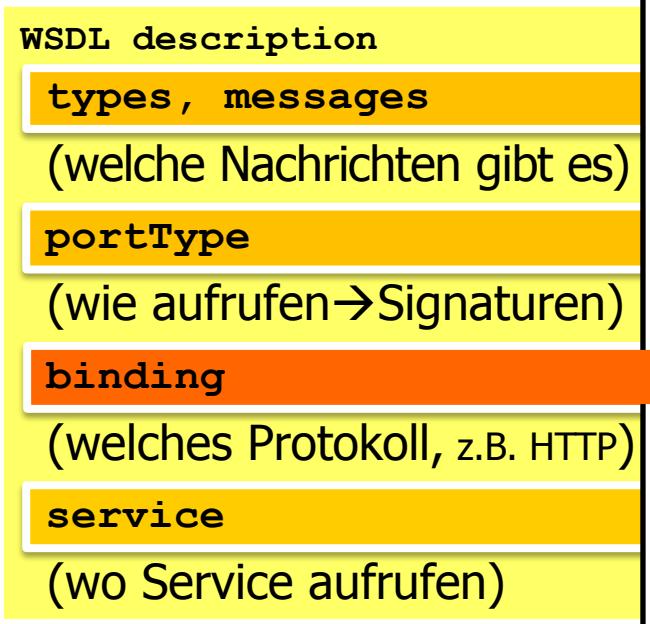
portType

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

Hier könnten weitere „operation" stehen

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

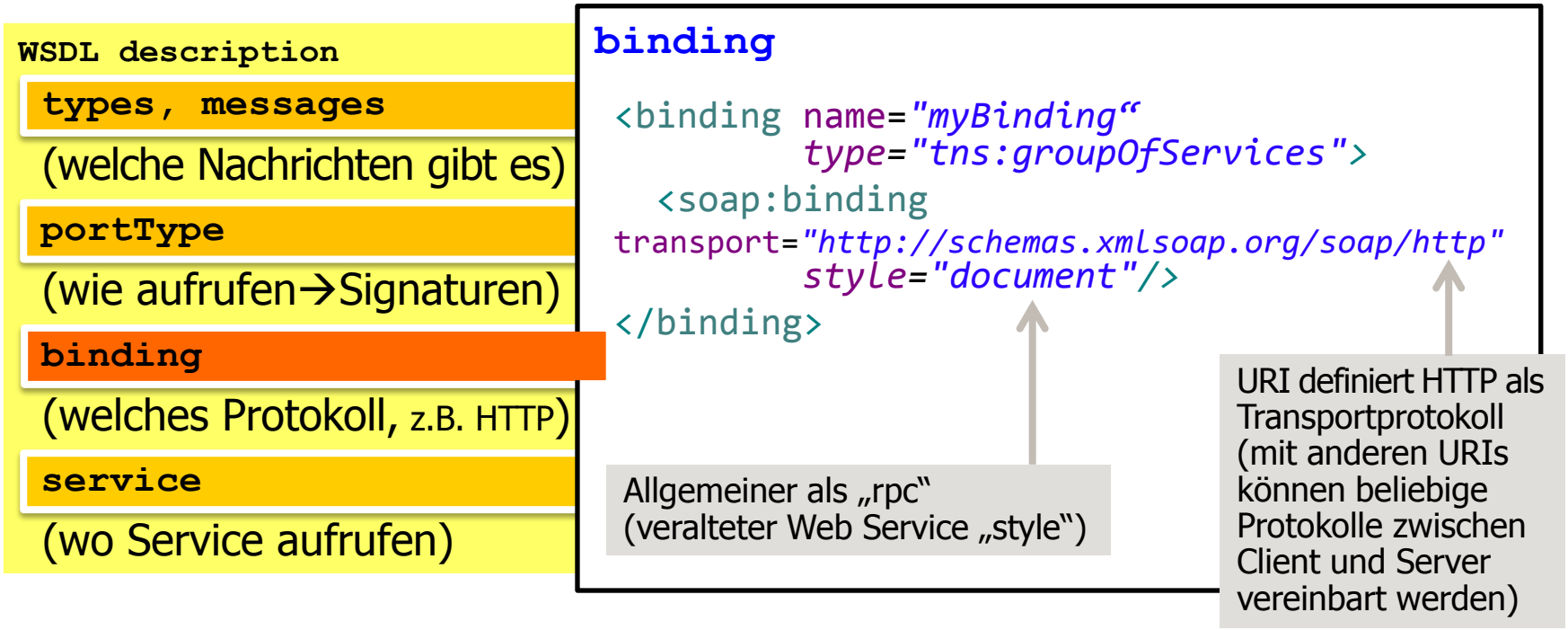
WSDL: binding



binding

- Bindet „portType" an ein Protokoll (z.B. HTTP)
- Es kann mehrere „binding"-Einträge für verschiedene Protokolle geben (Tools unterstützen oft nur einen Eintrag)
- (Im Normalfall genügen die Informationen aus „message" und „portType" für die Abbildung der Nachrichten auf ein konkretes Format, d.h. „binding" enthält kaum Information; Abbildung kann für Spezialfälle genau definiert werden)

WSDL-Beispiel: binding



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

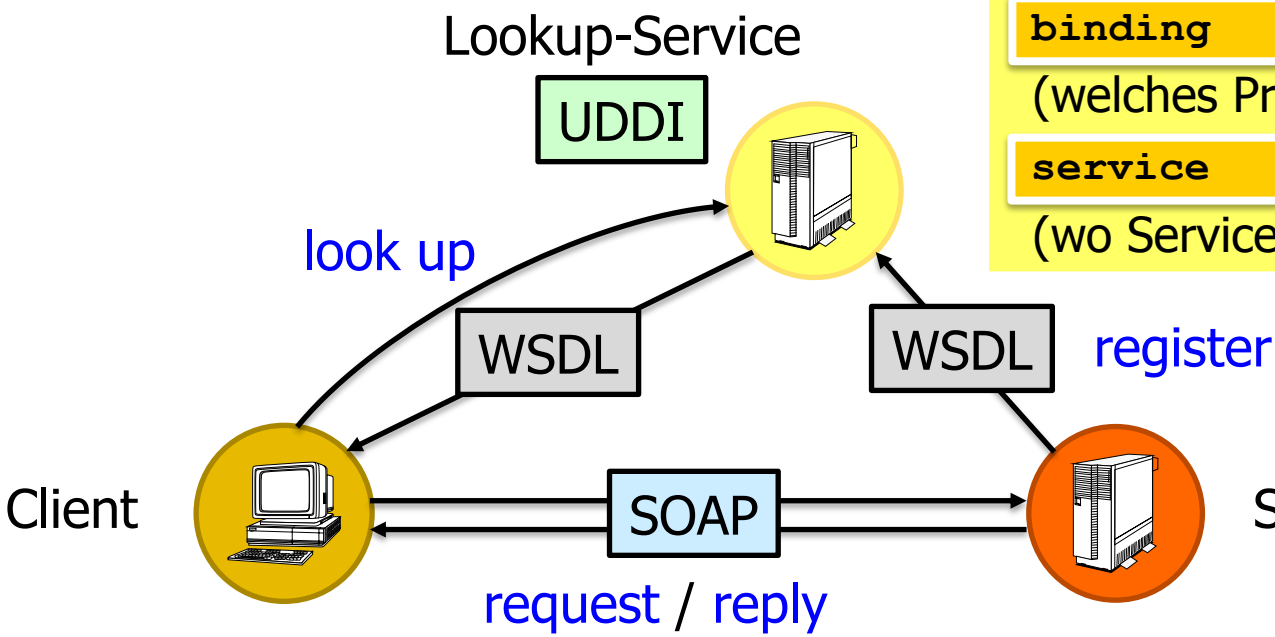
- Gibt Adresse des Web Services an (via „port" → „binding" → „portType")
- Ein Web Service kann mehrere „ports" haben (ist wieder nicht von allen Tools unterstützt)

```
<service name="myWebService">
  <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)

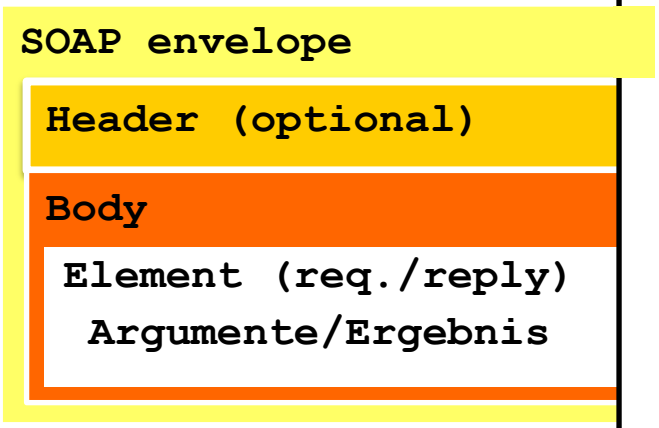


SOAP envelope

- **Header (optional)**
- **Body**
  - **Element (req./reply)**
    Argumente/Ergebnis

Jeweils XML-Dokumente

SOAP: envelope



Envelope

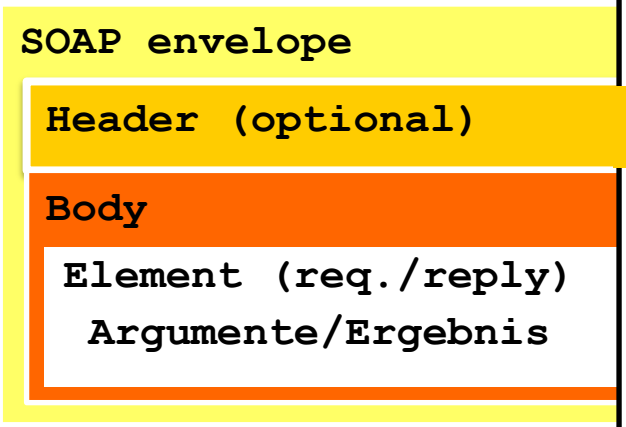
Erforderliche Attribute:

- Charakteristischer Namensraum (der das Dokument als SOAP-Nachricht definiert)
- „encodingStyle" gibt Kodierungsregeln für die SOAP-Serialisierung an

Enthaltene Teilstrukturen:

- Header (optional)
- Body (erforderlich)

SOAP: header

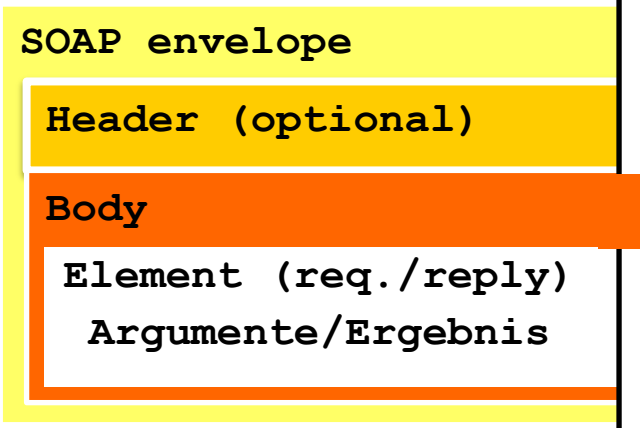


Header

Der SOAP-Header ist optional

Durch ihn könnten zusätzliche Informationen über den Transaktionskontext, z.B. bezüglich Authentifizierung oder Bezahlung, angegeben werden. Diese sind in zusätzlichen WS-* Spezifikationen definiert (z.B. OASIS-Standard)

SOAP: body

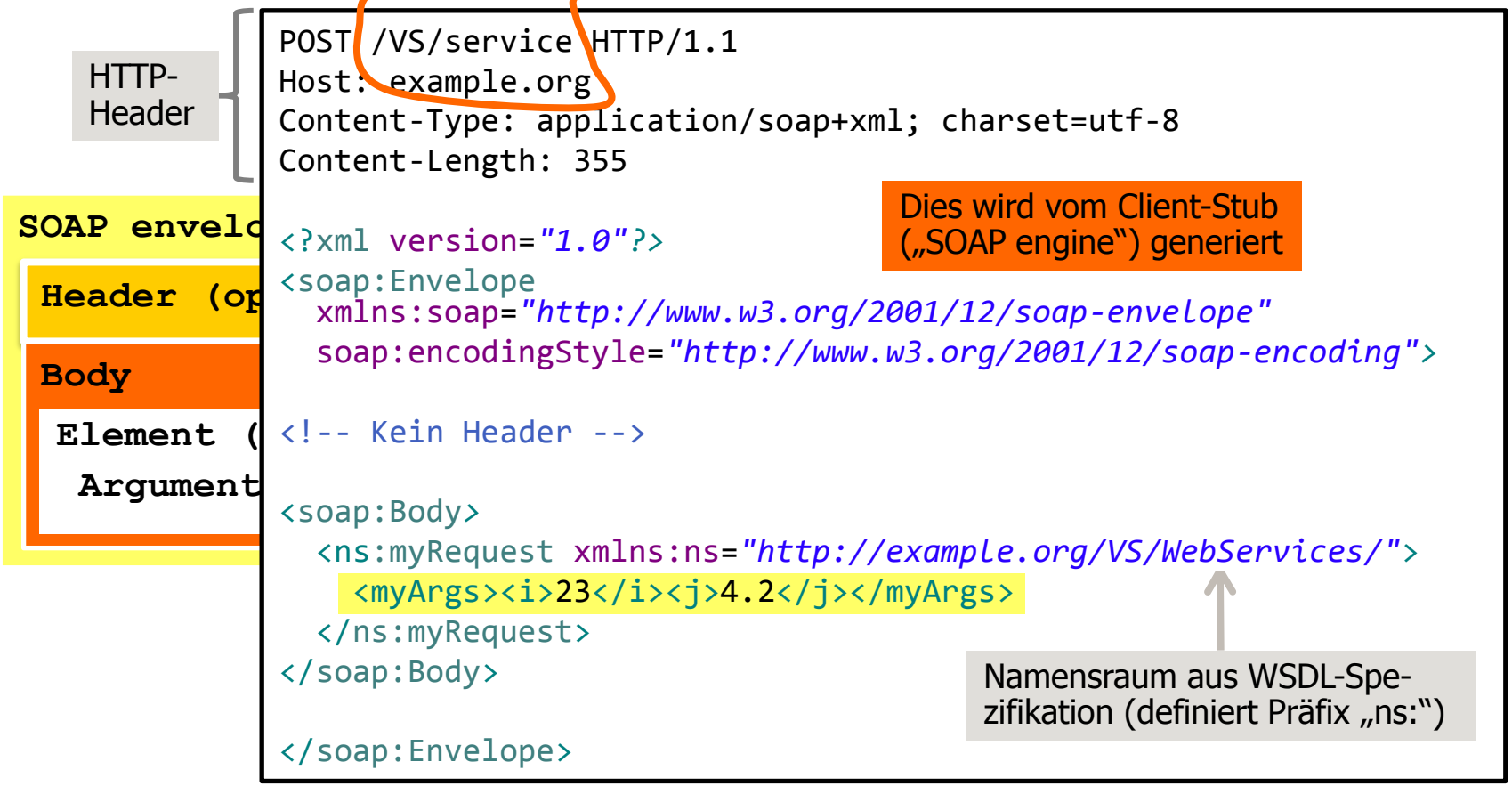


Body

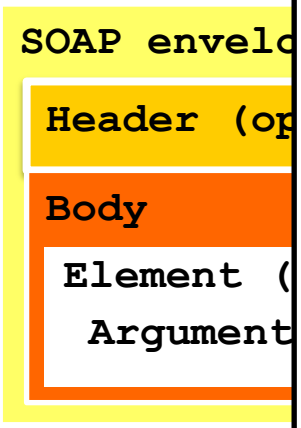
SOAP-Nutzlast:

- Enthält die (mit WSDL bzgl. ihrer Struktur definierte) „message“ mit ihren Elementen
- Es wird der Namensraum der WSDL-Spezifikation angegeben, womit die dort definierten Tags verwendet werden können

SOAP-Beispiel: Request



SOAP-Beispiel: Response



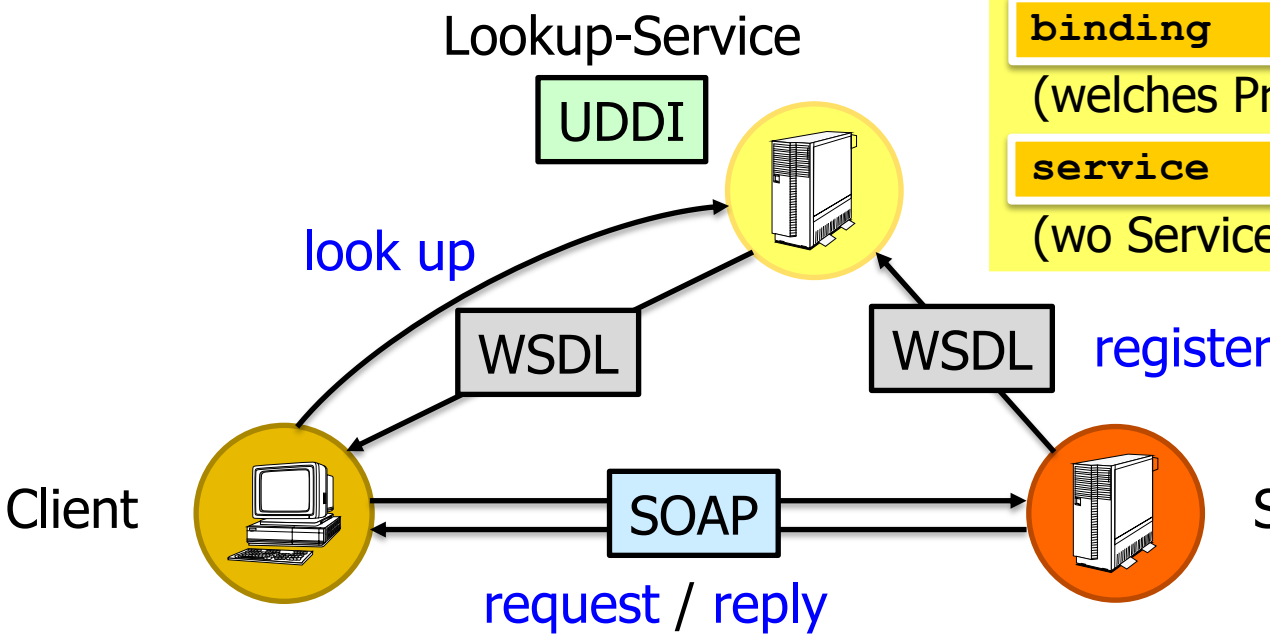
HTTP-Header

```
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)

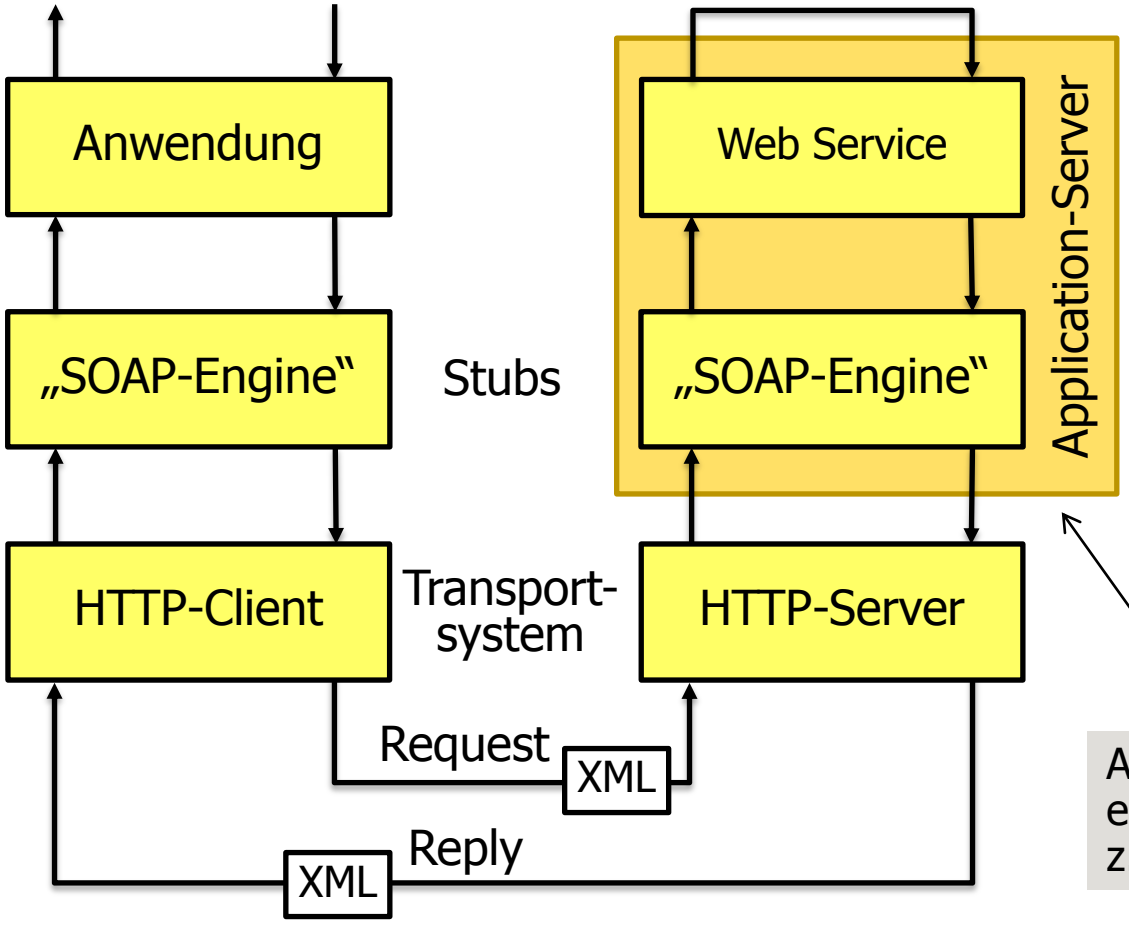


SOAP envelope

- **Header (optional)**
- **Body**
  - **Element (req./reply)**
    Argumente/Ergebnis

Jeweils XML-Dokumente

Web Service Stubs



- 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. „Apache Tomcat“, „Jetty“

vgl. frühere Folie „RPC: Implementierung“

Entwicklung von Web Service-Komponenten mit Java-IDE

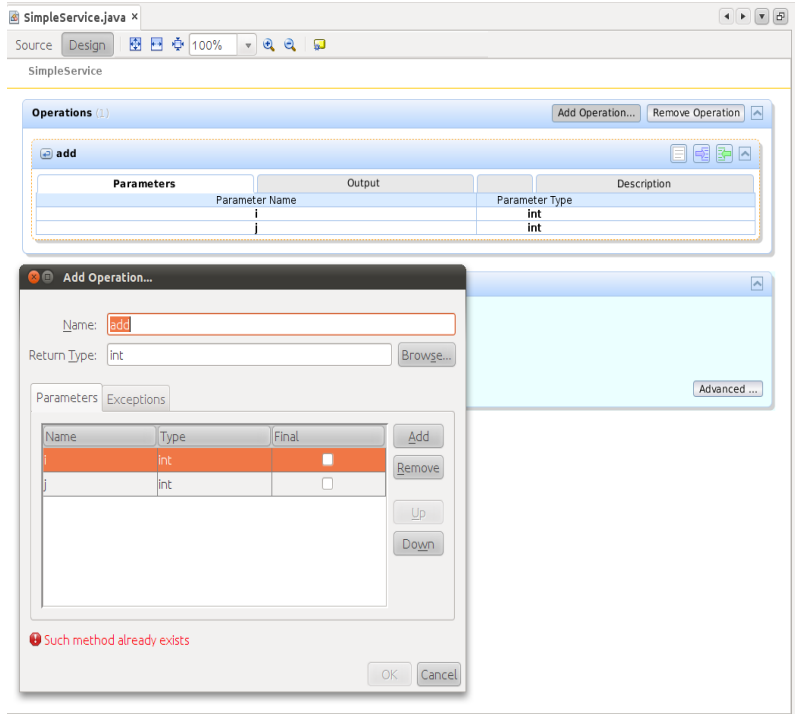
- JAX-WS: Java API for XML Web Services
  (Beispiel: NetBeans-Entwicklungsumgebung)

SimpleService.java ×

Source | Design

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  package ch.simple.service;
6
7  import javax.jws.WebService;
8  import javax.jws.WebMethod;
9  import javax.jws.WebParam;
10
11 /**
12  * JAX-WS Example
13  * "add" Web Service taking two Integers as input and returning an Integer
14  */
15 @WebService(serviceName = "SimpleService")
16 public class SimpleService {
17
18     /**
19      * Web service operation
20      */
21     @WebMethod(operationName = "add")
22     public int add(@WebParam(name = "i") int i, @WebParam(name = "j") int j) {
23         return (i + j);
24     }
25 }
26
```

Erweiterung von einfachen Java-Objekten zu Web Services per Java Annotations (bottom up)



Zusammenfassung

- Web Services spezifizieren die Schnittstellen und erlauben viele Konfigurationsmöglichkeiten
- Eine Grosszahl an Zusatzspezifikationen („WS-*“) deckt viele geschäftsrelevante Anforderungen ab (vorteilhaft wenn Dienste für Banken oder Krankenhäuser zertifiziert werden müssen)
- Relativ grosser Overhead für Aufrufe
- Konfigurationsmöglichkeiten machen WS-* sehr komplex und nur mit Werkzeugen beherrschbar
- Globaler UDDI-Service hat sich aus kommerziellen Gründen nicht durchsetzen können
- Erforderliche Code-Generierung und Softwareupdates bei Änderungen skalieren nicht für offene Webanwendungen

REST

Ressourcen-orientierte Architektur (ROA)

- Funktionalität wird nicht durch Services („SOA"), sondern durch (Web-) Ressourcen angeboten
- Ressource? Bezugsobjekt 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)

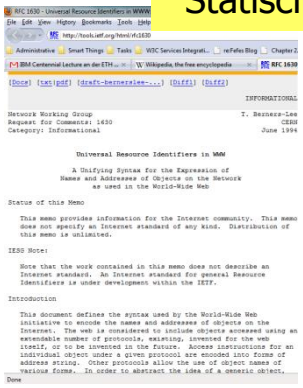
> *…Likewise, abstract concepts can be resources, such as the operators and operands of a mathematical equation, the types of a relationship…*

(Web-) Ressourcen

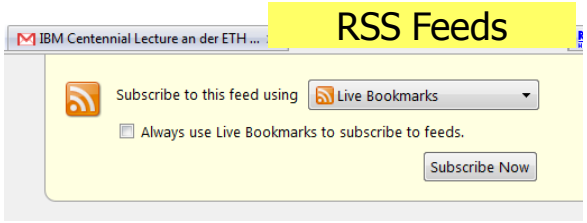
Warenkörbe/Repositories

Web-Dienste

Statische Websites



RSS Feeds



Foren

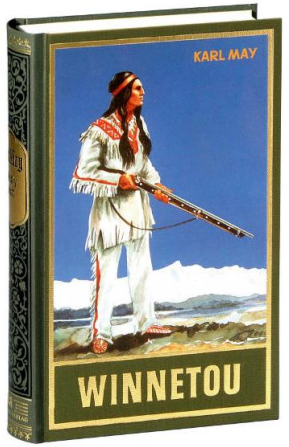
Physische Dinge

Repräsentation – Beispiel „Buch“

- Nachfolgend wird der Begriff „Repräsentation“ (einer Web-Ressource) verwendet
  - wir erläutern den Begriff an einem gleichnishaften Beispiel

---

- Das Buch als abstraktes Konzept
  - es gibt verschiedene Ausgaben, Exemplare etc.
  - identifiziert per ISBN: *ISBN-13: 978-3780200075*
  - oder URI: *urn:isbn:978-3780200075*
- Was wir kaufen oder ausleihen 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

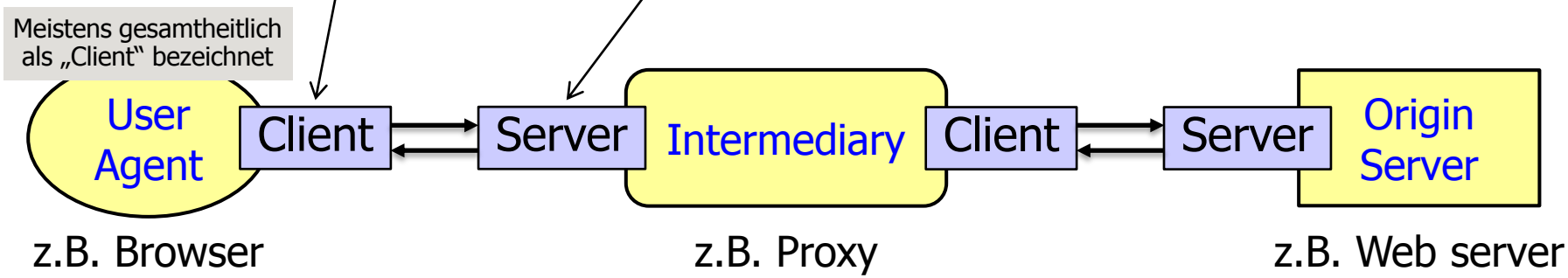


REST

- REST (als idealisierte Architektur des Web) steht für
  - Representational: Nicht Ressourcen, sondern deren Repräsentationen werden übertragen
  - State Transfer: Über diese erhält man Zugriff auf den Zustand von Ressourcen, kann diesen lokal ändern und wieder zurück an die Ressource übertragen
- Menge an Prinzipien („REST constraints“)
- Motivation und Entwicklung
  - Erfolg des World Wide Webs in technischer Hinsicht (z.B. bzgl. Skalierbarkeit) beruht auf Eigenschaften der zugrundeliegenden Protokolle und Mechanismen
  - Ableitung eines Architekturstils für verteilte Systeme allgemein, um die Möglichkeiten, die das Web (bzw. HTTP) bietet, bestmöglich auszunutzen

REST Prinzipien (1)

- Client-Server
  - REST-System besteht aus Komponenten, die entweder einen Client-Konnektor, Server-Konnektor oder beides haben können



  - User Agent hat die Initiative und erstellt Requests
  - Intermediary leitet Requests (ggf. mit Übersetzung) weiter
  - Origin Server hat die Hoheit über Ressourcen

REST Prinzipien (2)

- Zustandslosigkeit
  - Request muss alle Informationen zur Bearbeitung enthalten
  - d.h. der Kontext wird nur beim Client und nicht beim Server gehalten
  - entschärft Crash-Problematik und Orphans
  - verbessert Skalierbarkeit und Beobachtbarkeit
  - Ermöglicht Caching durch Wiederverwendbarkeit von Antworten

vgl. Formular bei Behörde, welches von jedem Beamten bearbeitet werden kann

- Caching
  - Antworten müssen Metadaten zur Gültigkeitsdauer enthalten (z.B. HTTP „Cache-Control" Header-Field)
  - Clients und Intermediarys können Antworten speichern und weitere Requests lokal bzw. direkt beantworten

REST Prinzipien (3)

- Einheitliche Schnittstellen
  - Adressierung immer durch URIs
  - einheitliche Aufrufe (z.B. GET, POST, PUT, DELETE bei HTTP)
  - Standard-Repräsentationsformate (z.B. Internet Media Types)
  - Ressourcen können mehrere Formate anbieten, aus denen der Client wählen kann (z.B. HTML, XML, JSON, …)
- Geschichtetes System
  - Clients sehen nicht, wie System hinter dem Server aussieht
  - Intermediaries können an beliebiger Stelle eingefügt werden (z.B. Proxies, Load-Balancers, Gateways für Legacy-Systeme)
- Code bei Bedarf
  - Server kann Logik an Client auslagern (z.B. JavaScript)

Keine anwendungsspezifischen Typen und Operationen wie bei Service-orientierten Architekturen

Eigenschaften von REST

- Skalierbarkeit
  - Zustandslosigkeit erlaubt effiziente Server und Lastverteilung
  - Caching reduziert Kommunikation und somit Systemlast
- Anpassungsfähigkeit
  - Einheitliche Schnittstellen entkoppeln Client und Server
  - Schichtung erlaubt nachträgliche Topologieänderungen
  - Code bei Bedarf erlaubt das Nachrüsten von Clients im Betrieb
- Beobachtbarkeit und Zuverlässigkeit
  - Requests mit allen Informationen sind leicht nachverfolgbar
  - Einheitliche Schnittschnellen, Caching und Anpassungsfähigkeit ermöglichen hohe Zuverlässigkeit (z.B. auch durch Redundanz)

Entwicklung von REST-Komponenten mit Java-IDE

- JAX-RS: Java API for RESTful Web Services (Beispiel: Eclipse)

ShoppingCartResource.java

```
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;

@Path("/cart/{cartID}")
public class ShoppingCartResource {

    @GET @Produces("text/html")
    public Response getCartContent(
            @PathParam("cartID") String cartID) {

        return Response.ok("Books in your cart: ...").build();
    }

    @POST @Consumes("application/json") @Produces("text/html")
    public Response addBookToCart(
            @PathParam("cartID") String cartID,
            @QueryParam("bookID") String bookID) {

        return Response.created(c.bookURIInCart).build();
    }
}
```

Erweiterung von einfachen Java-Objekten zu Ressourcen per Java Annotations

@Path: Pfad zur Ressource

Definition der verwendeten Media Types (@Consumes und @Produces)

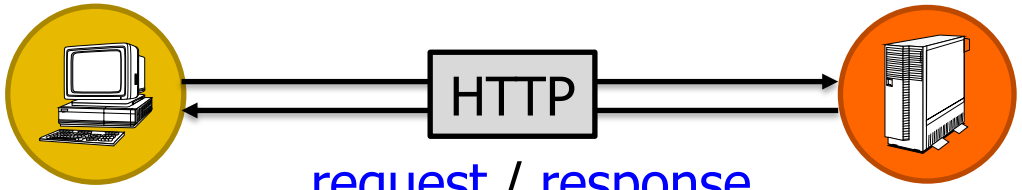
Extraktion von Parametern aus dem Request mittels:

@PathParam: z.B. *{cartID}*

@QueryParam: z.B *?bookID=123*

REST-Anwendungsmodell

- „Hypermedia as the Engine of Application State“
  - Client kennt ausschliesslich die Basis-URI des Dienstes sowie die Repräsentationsformate, die verwendet werden
  - Server leitet durch die Anwendungszustände durch Bekanntgabe von Wahlmöglichkeiten (hyperlinks, forms)



- Der Anwendungszustand besteht aus zwei Komponenten
  - Ressourcenzustand beim Origin Server (kann auch statisch sein)
  - Client-Zustand (ursprünglich „application state“ genannt, da Ressourcen meist statisch waren)

REST: Zustandsspeicherung

- Ressourcenzustand
  - Klassisch die Sammlung an statischen Dokumenten auf dem Server
  - Bei dynamischen Anwendungen oft nur die relevanten Werte; der Rest der Repräsentationen wird von der (statischen) Anwendungslogik oft mithilfe von Templates generiert
- Client-Zustand
  - Bezeichnet den aktuellen Schritt oder Kontext in der Anwendung, z.B. die aktuell gerenderte Repräsentation, sowie deren Historie
  - Bookmarks ergeben Sinn: vollständige URI und Client weiss, in welchem Kontext Bookmark angelegt wird
  - Back button im Browser ergibt Sinn: er führt zu einem früheren Zustand (mit im Client gecachten Repräsentationen)

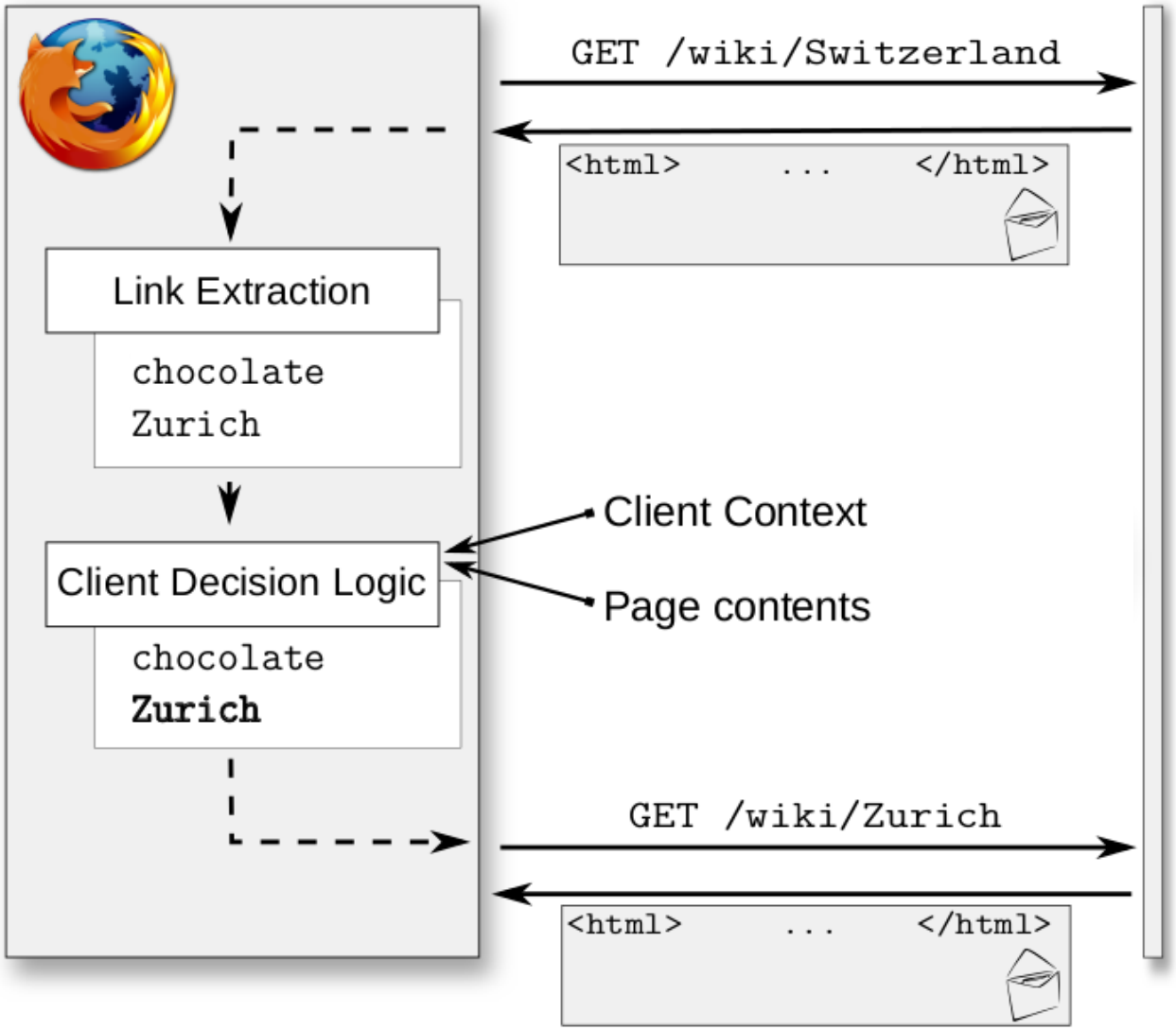
REST: Zustandsspeicherung

- Zustandslosigkeit (der Kommunikation) heisst Ressourcenzustand und Client-Zustand sind strikt getrennt
  - Der Client hinterlegt keine Informationen beim Server, die für Folge-Requests gelten sollen (vgl. Sessions z.B. bei FTP oder SSH)
  - Der Server hat keinen direkten Einfluss auf den Client-Zustand; er kann nur indirekt durch die Bekanntgabe von Auswahl-möglichkeiten durch eine Anwendung leiten, Entscheidung liegt aber vollständig beim Client
- Nur so sind Client und Server entkoppelt, so dass die Vorteile von REST gelten

Beispiel:

Client

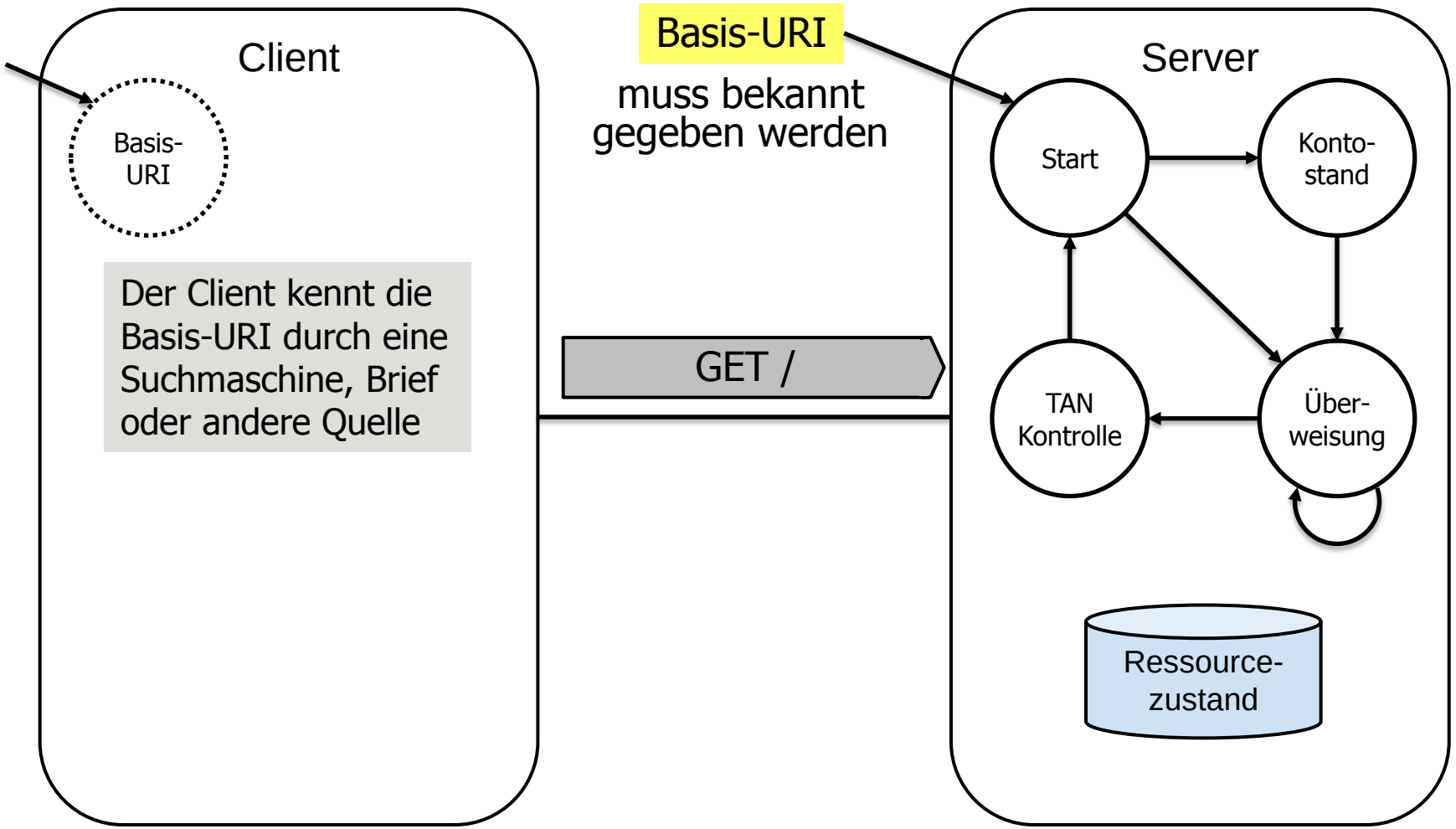
(ein Mensch an einem Browser)



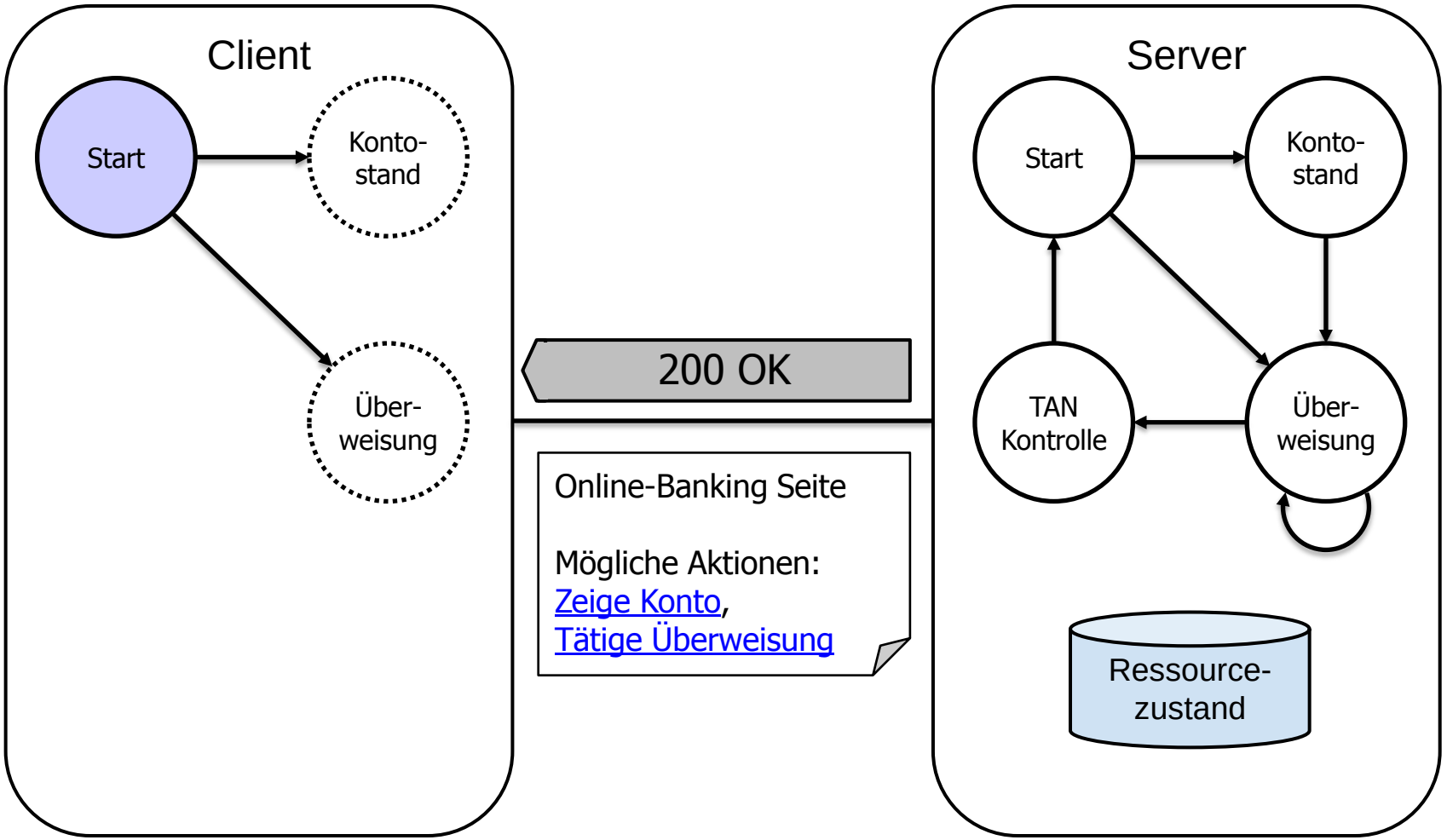
REST: Widersprüchliche Praxis

- Bei vielen Implementierungen wird der Client-Zustand auch auf dem Server gehalten oder von ihm direkt verändert
  - „URL Rewriting“ kodiert spezielle Informationen (z.B. eine Client-spezifische Session-ID) in die Requests
  - Vom Server definierte „Cookies“ müssen vom Client mitgesendet werden und verändern die Interpretation des Requests
- Dann funktioniert eine Kopie einer URI (bookmark) später meistens nicht, weil dem Server der Kontext dazu fehlt
  - auch back button im Browser ist problematisch: führt zu einer früheren Zustandskopie, ohne dass der Server dies mitbekommt – Client meint fälschlicherweise, in einem gewissen Zustand zu sein, der tatsächliche Zustand wird aber auf dem Server gehalten
- Widerspricht den REST Prinzipien

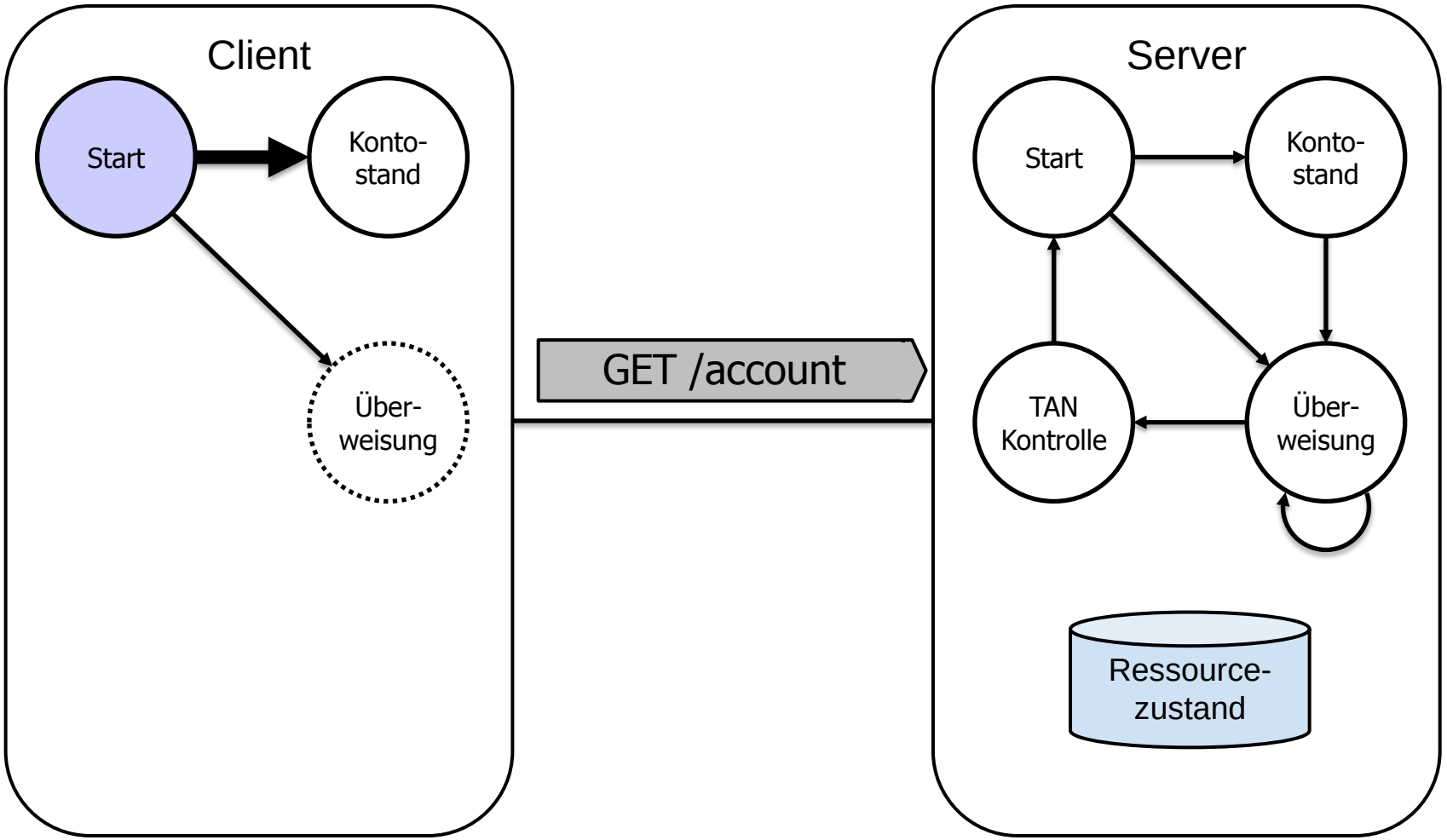
Beispiel: Bankanwendung mit REST



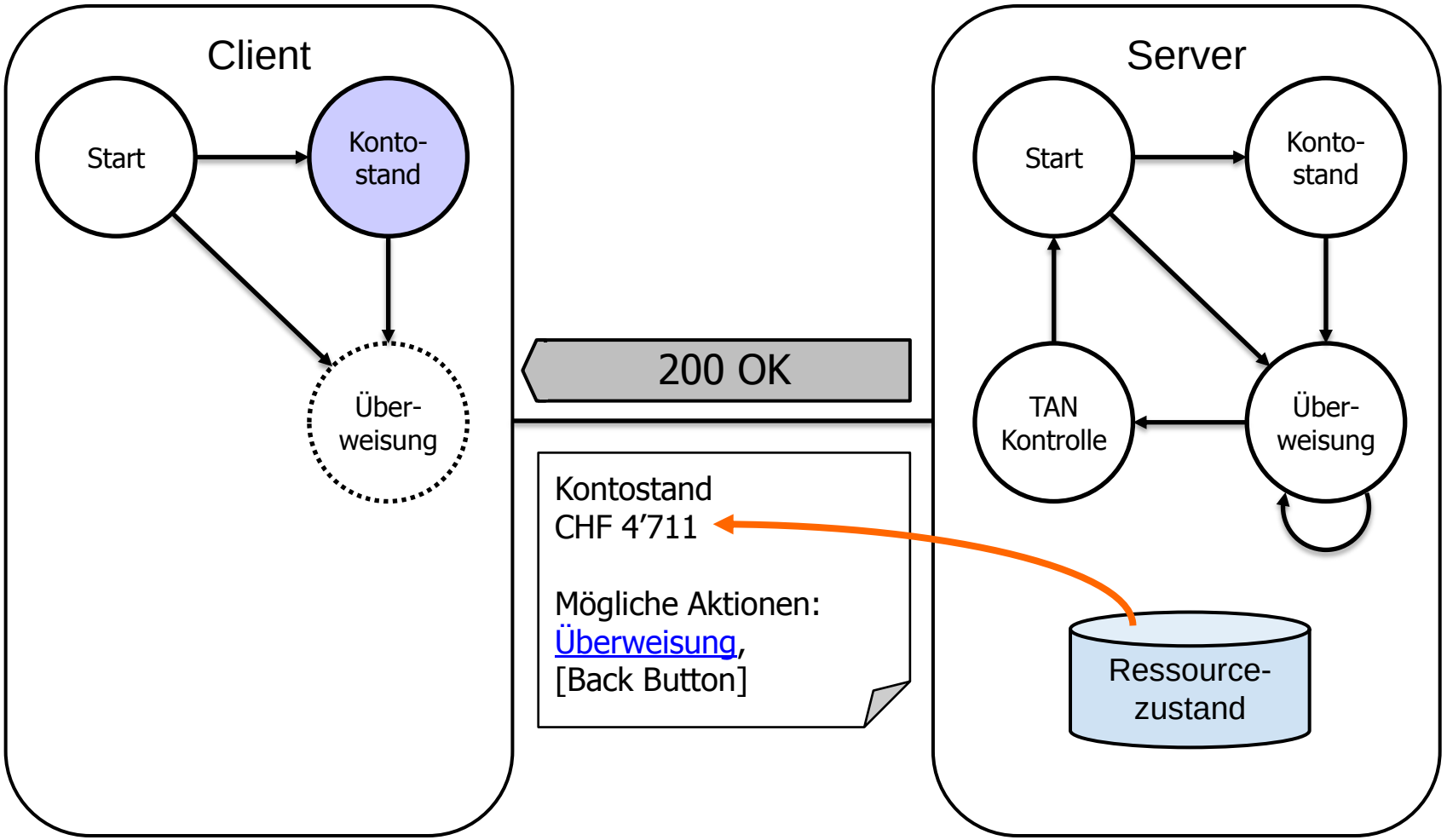
Beispiel: Bankanwendung mit REST



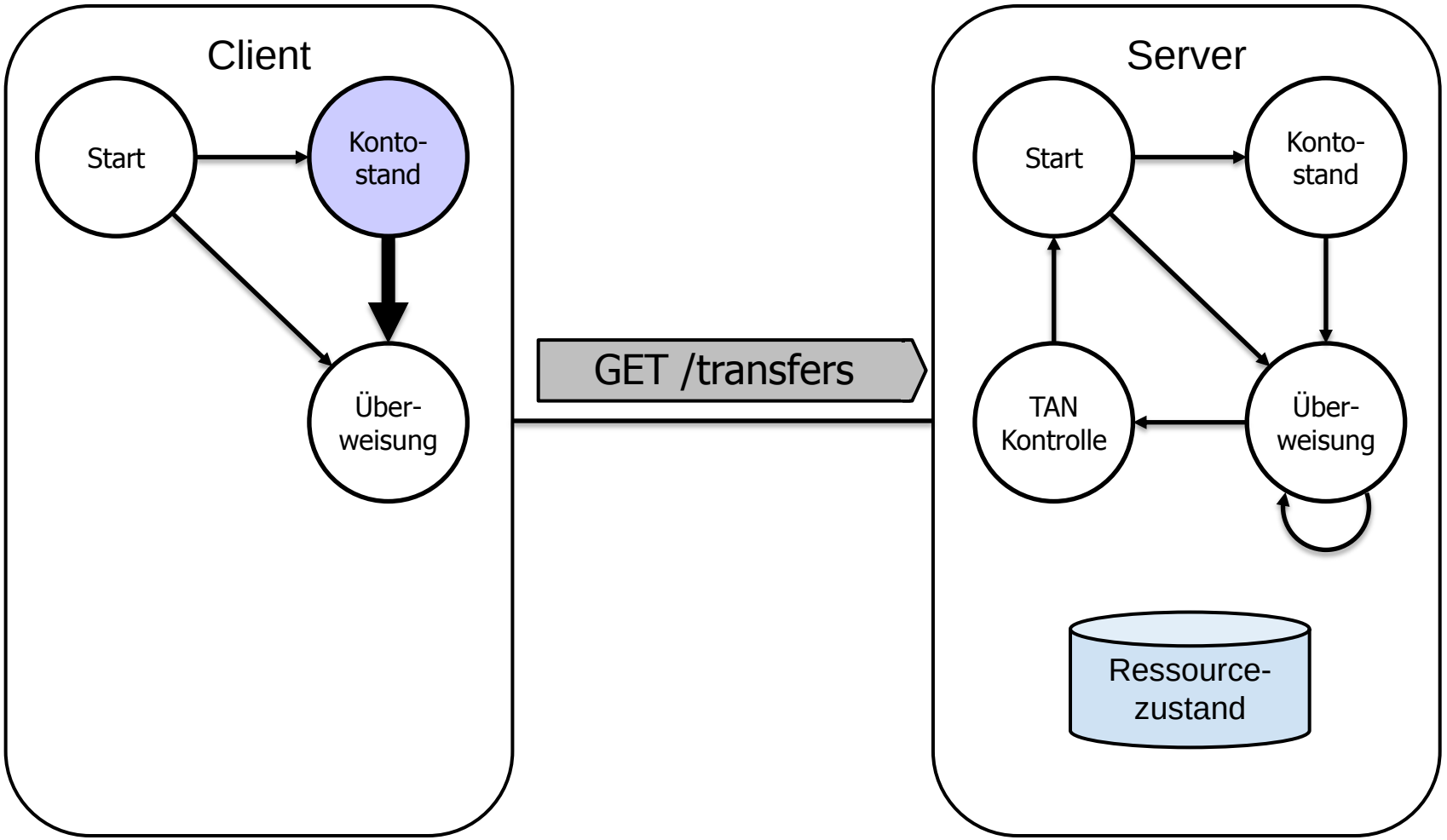
Beispiel: Bankanwendung mit REST



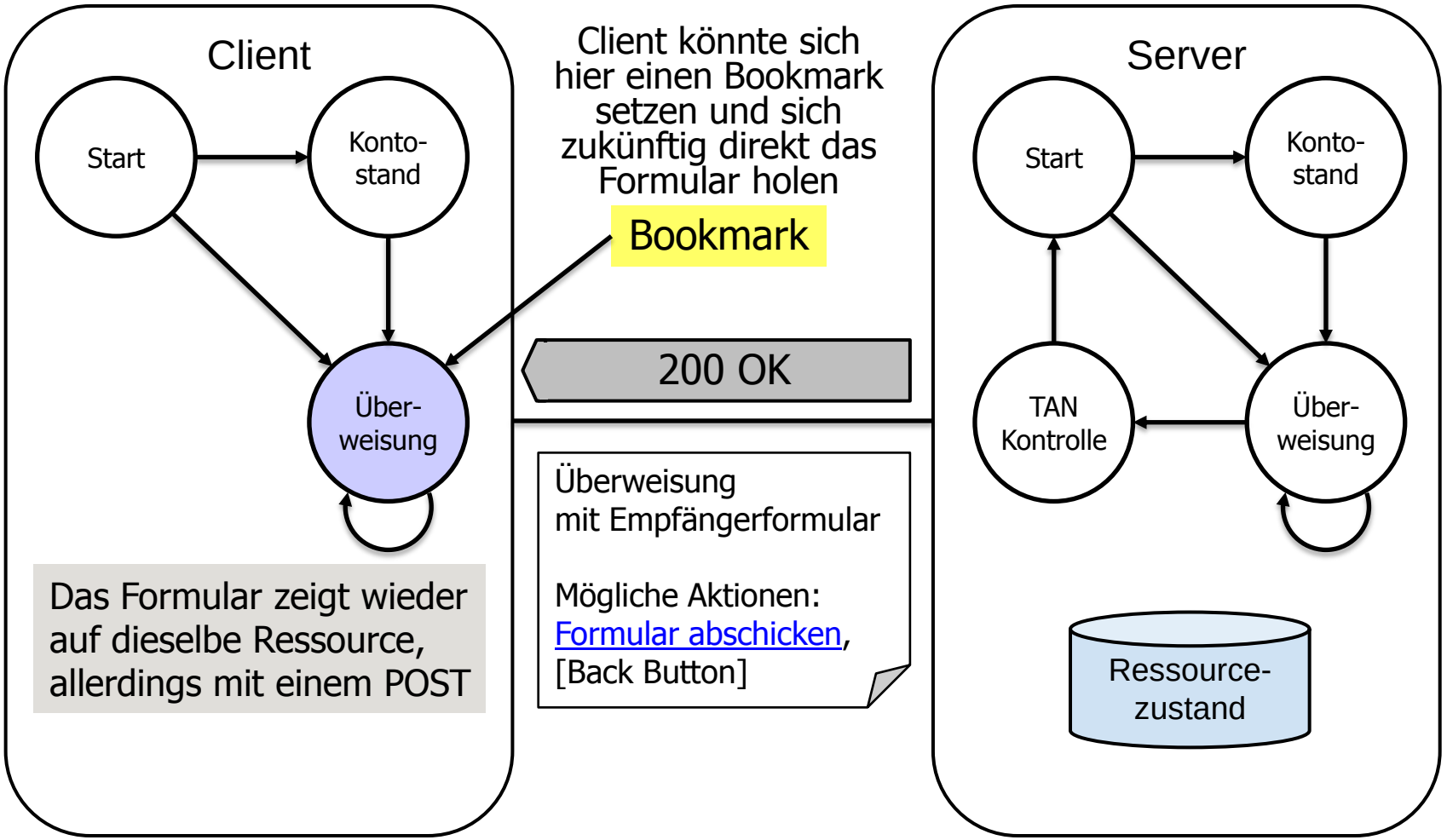
Beispiel: Bankanwendung mit REST



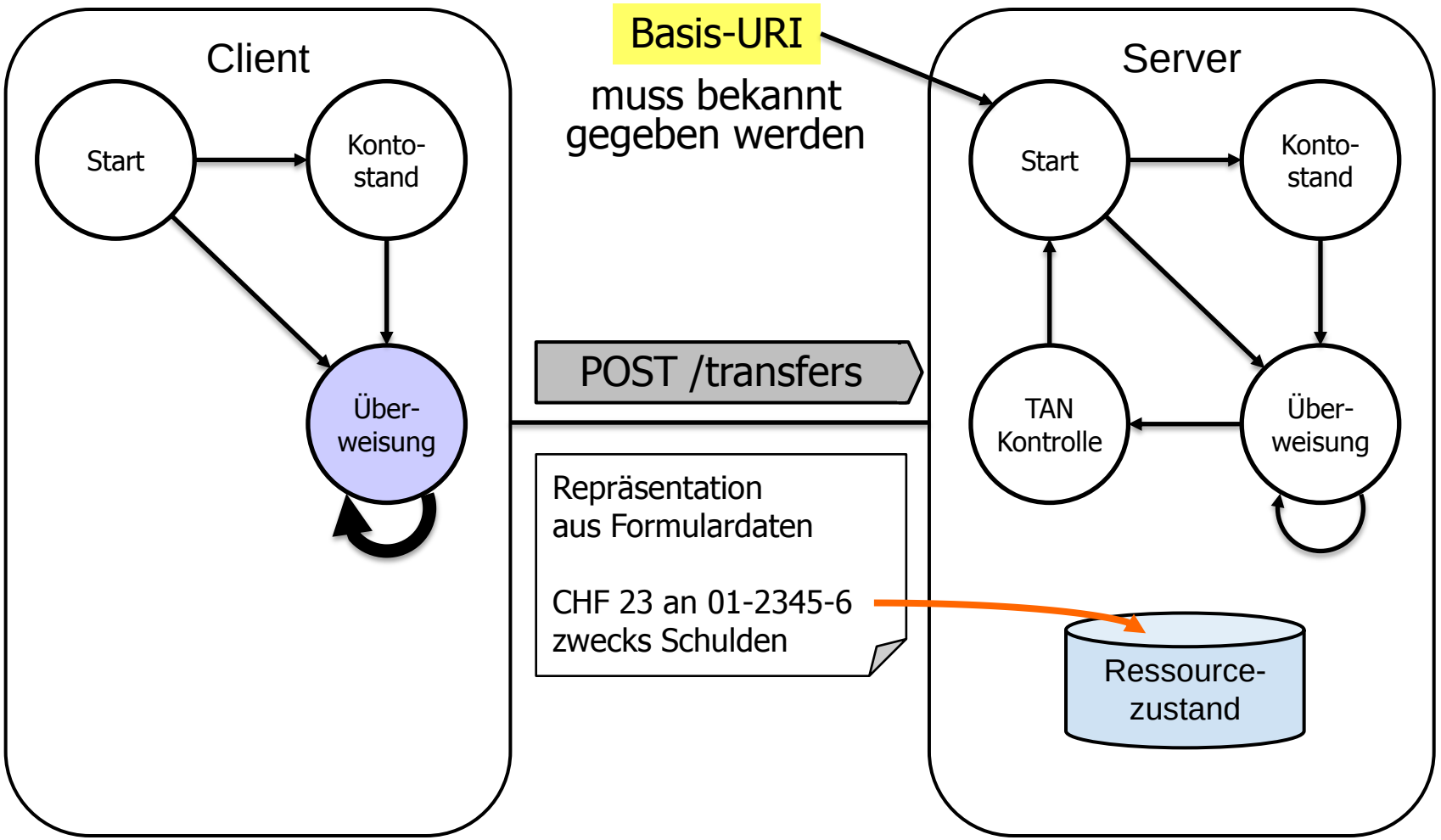
Beispiel: Bankanwendung mit REST



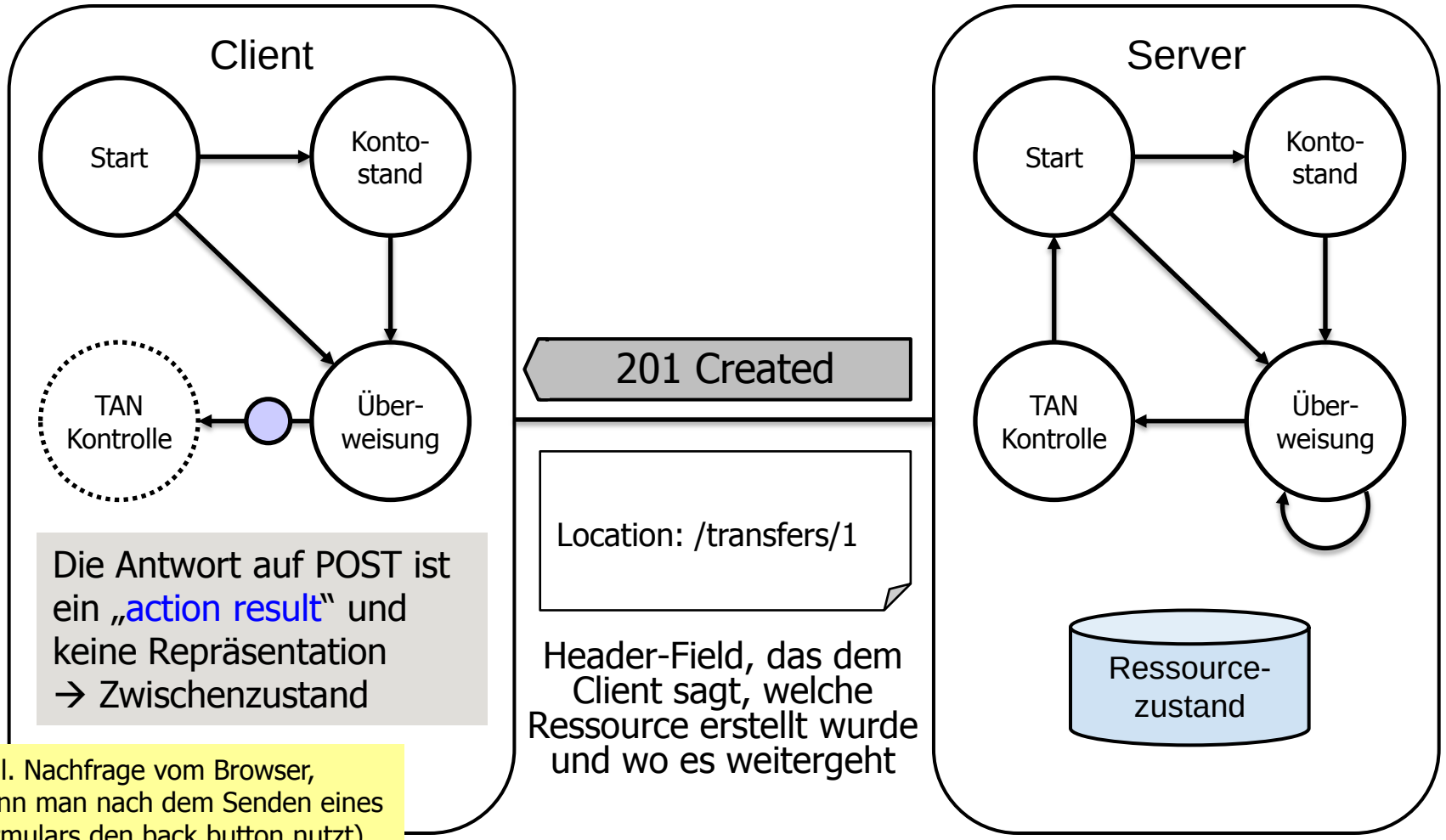
Beispiel: Bankanwendung mit REST



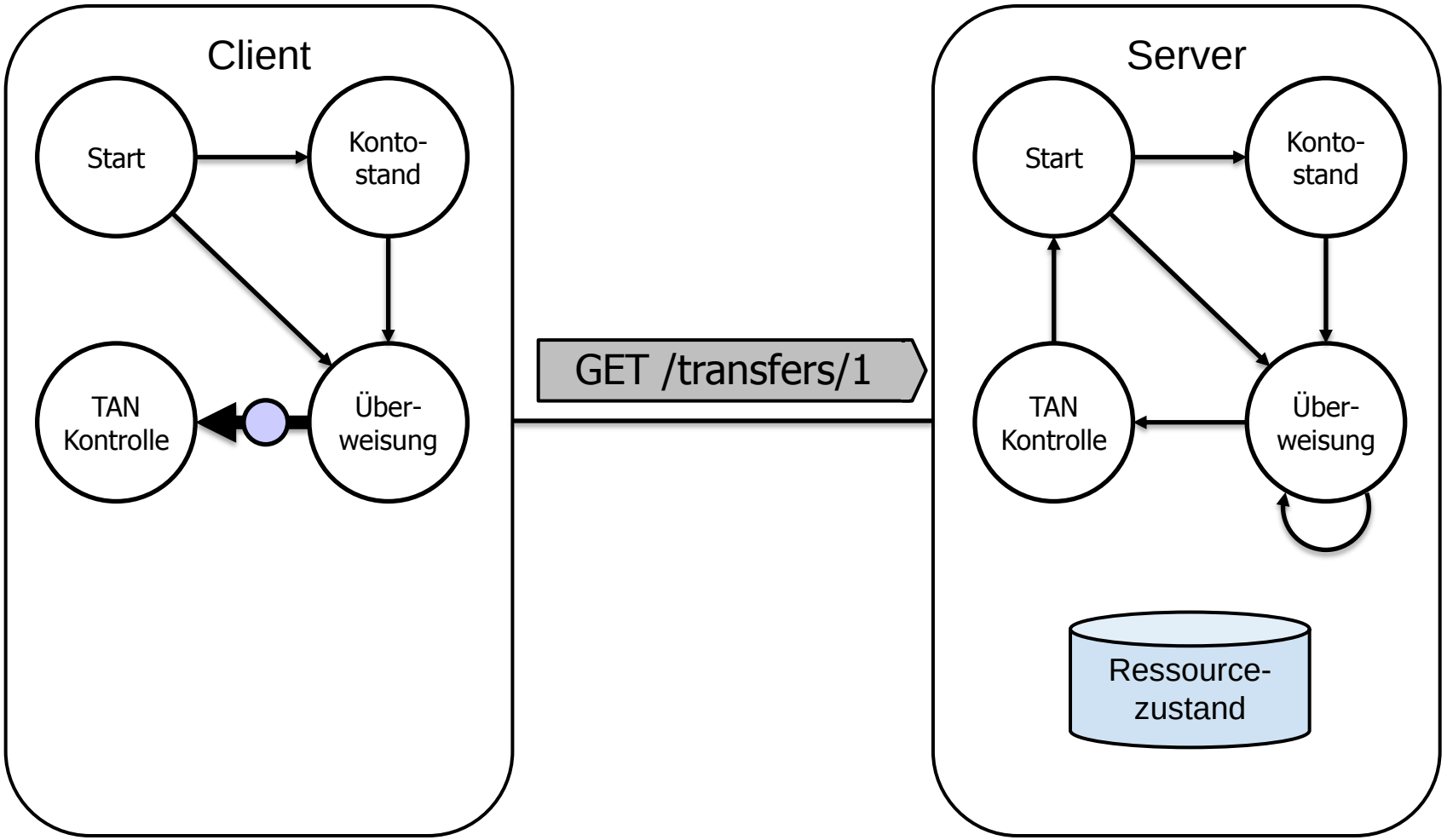
Beispiel: Bankanwendung mit REST



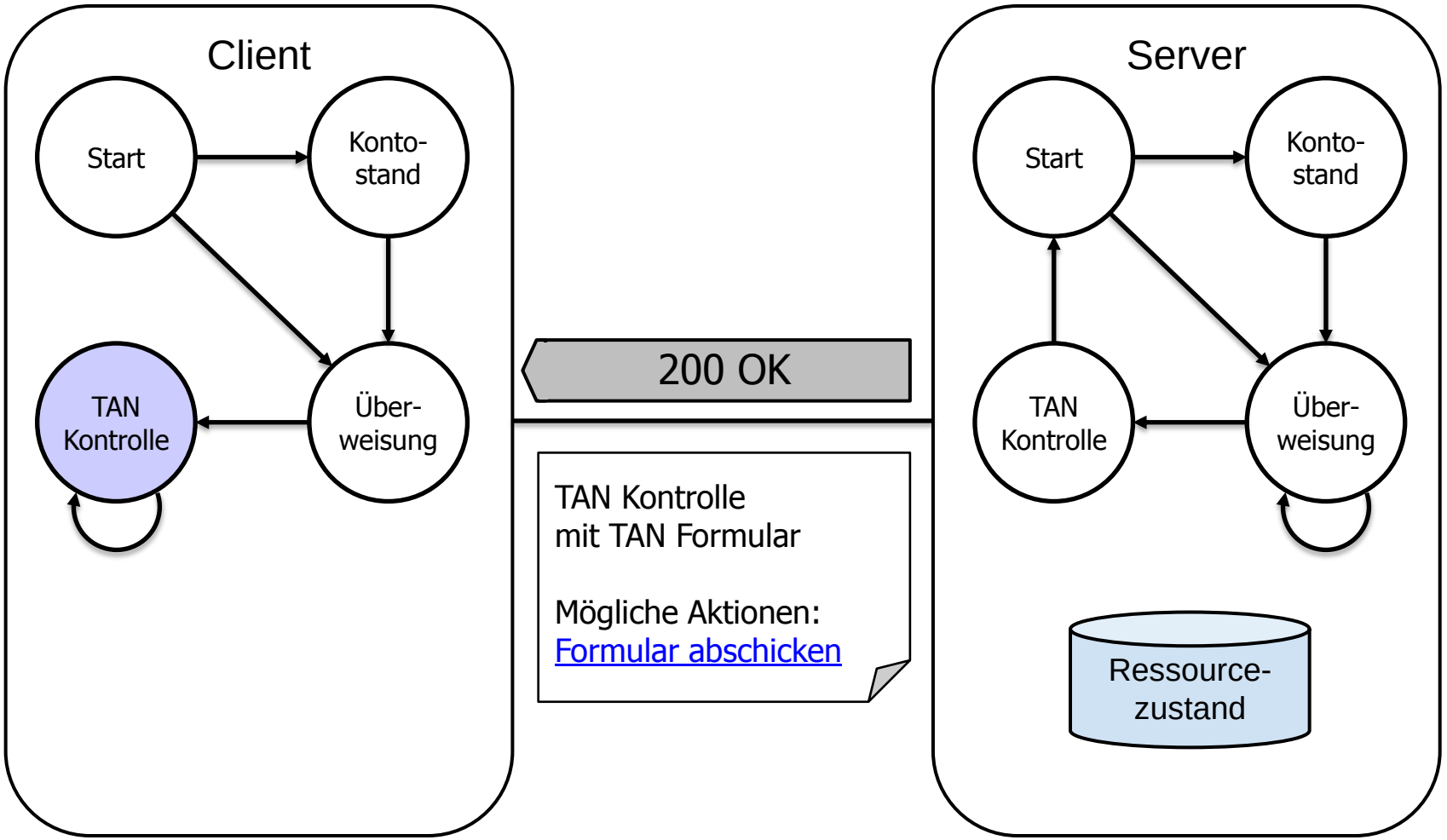
Beispiel: Bankanwendung mit REST



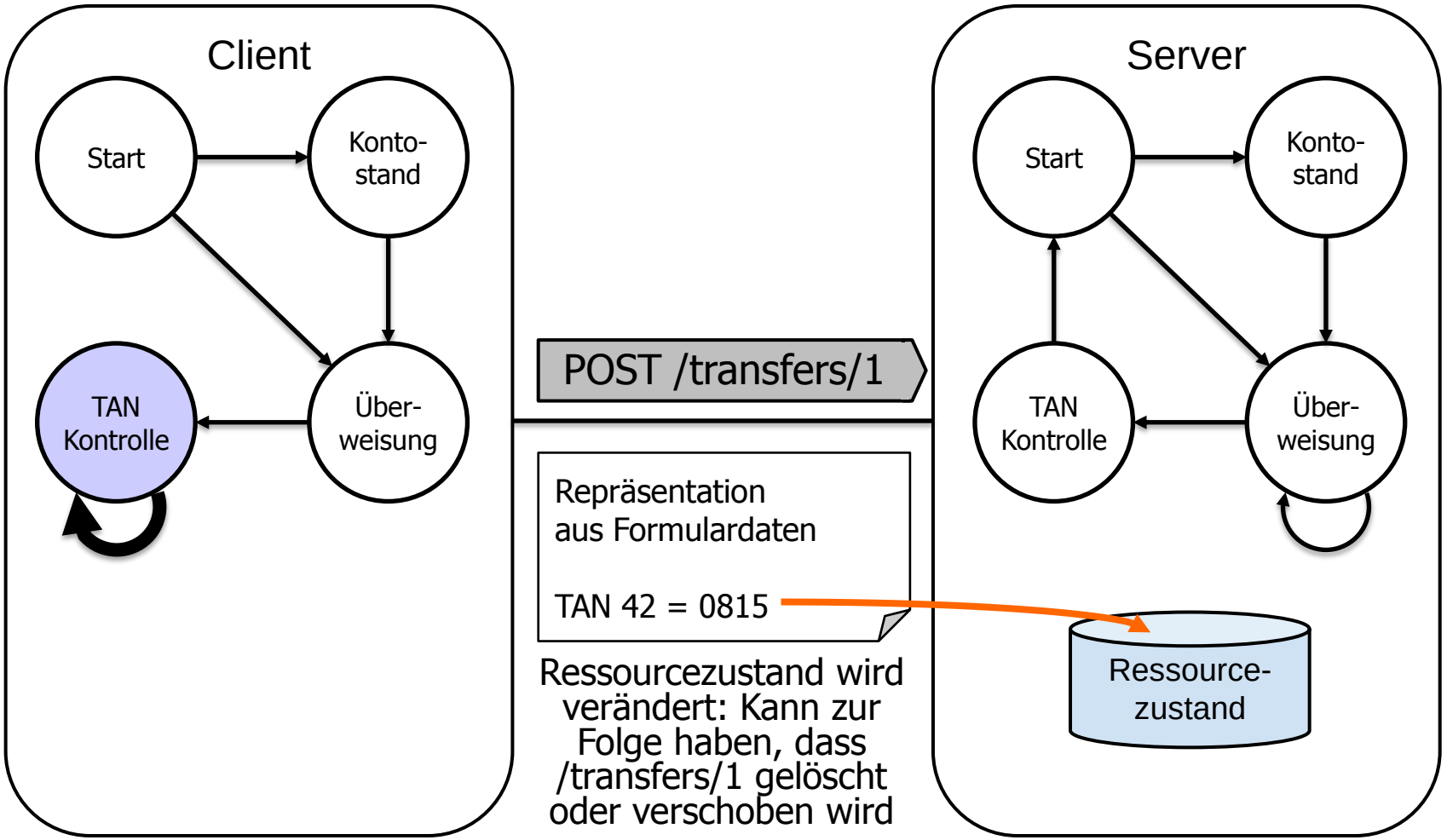
Beispiel: Bankanwendung mit REST



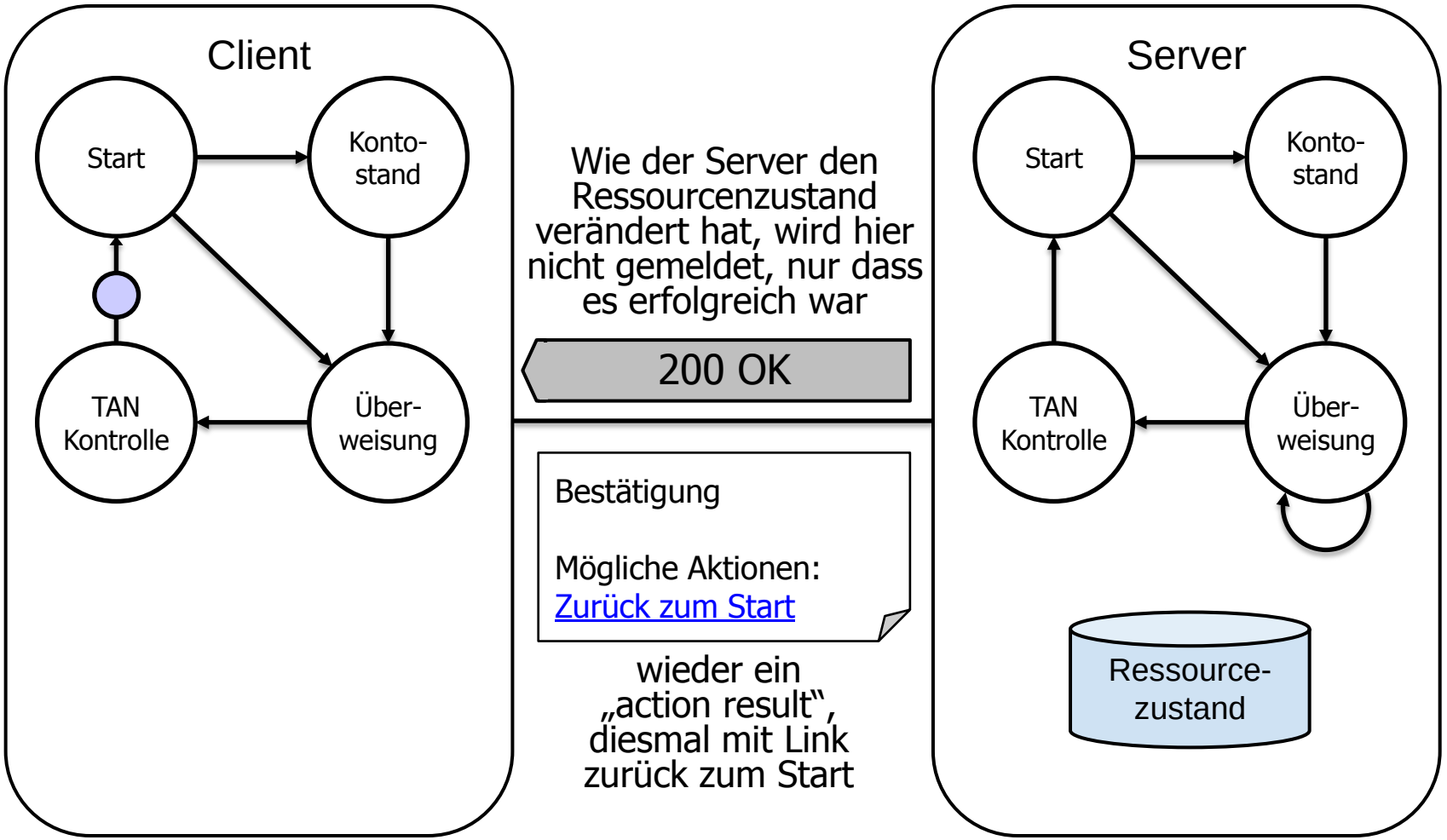
Beispiel: Bankanwendung mit REST



Beispiel: Bankanwendung mit REST



Beispiel: Bankanwendung mit REST



Beispiel: Bankanwendung mit REST

- Der Client hat die Anwendung sukzessive durch die Bekanntgabe von Links erlernt
- Der Server kann die Anwendungslogik unabhängig vom Client verändern → beim nächsten Mal werden einfach andere Links übertragen, die z.B. den Ablauf verändern oder zu neuen Ressourcen führen
- Bookmarks funktionieren dann ggf. nicht mehr
  - Server antwortet mit 404 Not Found
  - Client muss (oder besser: kann) wieder mit Basis-URI beginnen, um die Änderungen zu erfahren

Maschine statt Mensch hinter Client?
Siehe z.B. http://www.hydra-cg.com/
Community Group bei der W3C