

# VESYS Zusammenfassung

---

## Inhalt

1	Einführung .....	4
1.1	Definition "Verteiltes System" .....	4
1.2	Klassifikation .....	4
1.2.1	Client-Server System.....	4
1.2.2	Verteilte Applikation .....	4
1.2.3	P2P Netzwerk.....	4
1.3	Interaktionsmodelle .....	4
1.3.1	Kooperation .....	4
1.3.2	Kommunikation .....	4
1.4	Kommunikationsstile .....	4
1.4.1	RPC (Remote Procedure Calls).....	4
1.4.2	Nachrichtenbasierte Systeme .....	5
2	Java Networking .....	6
2.1	IP Socket Verbindungen.....	6
2.2	Class InetAddress.....	6
2.2.1	Statische Factory Methoden.....	6
2.3	Instanzmethoden.....	6
2.4	Class NetworkInterface.....	6
2.5	Stream Socket .....	6
2.6	Class Socket .....	6
2.7	class ServerSocket .....	7
2.8	ObjectStreams .....	7
2.9	Class DatagramSocket.....	7
3	Internet.....	8
3.1	HTTP (HyperText Transfer Protocol) .....	8
3.1.1	Request.....	8
3.1.2	Response .....	8
3.2	Common Header Lines .....	8
3.2.1	MIME (Multipurpose Internet Mail Extensions).....	9
3.3	HTTP Status Codes.....	9
3.3.1	1xx: Information .....	9
3.3.2	2xx: Successful .....	9

3.3.3	3xx: Redirection .....	9
3.3.4	4xx: Client Error .....	9
3.3.5	5xx: Server Error .....	10
3.4	GET vs POST .....	10
4	Webservices.....	12
4.1	XML-RPC (Remote Procedure Call) .....	12
4.1.1	Primitive Datentypen.....	12
4.1.2	Structs.....	12
4.1.3	Arrays.....	12
4.1.4	XML-RPC Request .....	12
4.1.5	XML-RPC Response.....	13
4.1.6	Fault Result .....	13
4.1.7	Apache XML-RPC Sample Server .....	13
4.1.8	Apache XML-RPC Client .....	14
4.2	SOAP .....	14
4.2.1	WSDL .....	14
4.2.2	JAX-WS (Java API for XML Web Services) .....	15
4.2.3	JAX-WS HTTP Request .....	19
4.2.4	JAX-WS HTTP Response .....	19
4.2.5	SOAP Message Structure .....	20
4.3	Vergleich von SOAP und XML-RPC.....	20
4.3.1	Allgemein.....	20
4.3.2	Features .....	20
5	REST (Representational State Transfer) .....	21
5.1	Prinzipien .....	21
5.2	REST in 5 Schritten .....	21
5.3	REST vs SOAP .....	22
5.4	REST Client Seite .....	22
5.5	REST Server Seite .....	22
5.6	JSR 311: Java API for RESTful Web Services (JAX-RS).....	22
5.6.1	HTTP Methoden.....	22
5.6.2	JAX-RS Injection .....	22
5.6.3	Content Negotiaton .....	23
5.6.4	JAX-RS Data Binding.....	24
5.6.5	JAX-RS Antworten .....	24
5.6.6	JAX-RS Cache Control.....	25

5.6.7	JAX-RS Concurrency .....	25
5.7	JAX-RS Deployment .....	25
5.7.1	Server-Implementationen .....	25
6	Glossar .....	26
6.1	CRLF (Carriage Return [and] Line Feed).....	26
7	Code Beispiele .....	26
7.1	StringBuilder .....	26
7.2	Netzwerkinterfaces ausgeben .....	26
7.3	Example WSDL 2.0 code .....	26

# 1 Einführung

## 1.1 Definition “Verteiltes System”

Ein verteiltes System ist eine Menge von unabhängigen Computern die von ihren Usern als ein einziges zusammenhängendes System wahrgenommen wird.

## 1.2 Klassifikation

Es gibt verschiedene Varianten von verteilten Systemen:

### 1.2.1 Client-Server System

Mehrere Clients benutzen denselben Server (z.B. Webserver)

### 1.2.2 Verteilte Applikation

Ein Algorithmus läuft auf mehreren „Slaves“, zentral koordiniert, mobiler Code (z.B. SETI@Home)

### 1.2.3 P2P Netzwerk

Gleichwertige autonome Knoten (z.B. Kazaa, Gnutella usw.)

## 1.3 Interaktionsmodelle

Da die verschiedenen Knoten in verteilten Systemen zum Teil auf dieselben Daten zugreifen müssen und sich auch teilweise Daten hin und herschicken müssen gibt es zwei Arten von Interaktionen.

### 1.3.1 Kooperation

Verschiedene verteilte Prozesse arbeiten auf demselben freigegebenen Repository. → Symmetrisch

### 1.3.2 Kommunikation

Ein Prozess sendet Daten als Nachrichten an anderen Prozess. → Asymmetrisch (sender/receiver)

Kommunikation kann wiederum in zwei Teilbereiche zerlegt werden.

#### 1.3.2.1 Synchrone Kommunikation

- Sender wartet bis Kommunikation beendet
- Sender & Empfänger synchronisieren Kommunikation
- Beispiel: HTTP

#### 1.3.2.2 Asynchrone Kommunikation

- Sender & Empfänger laufen unabhängig
- Sender wartet nicht auf Antwort
- Beispiel: Email, SMS

## 1.4 Kommunikationsstile

### 1.4.1 RPC (Remote Procedure Calls)

- Ähnlich wie lokale Prozeduraufrufe, Operationen beschrieben durch Name, Parameter und Rückgabewert
- Typischerweise synchron (Client wartet bis Kommunikation beendet)

Prozedural:

- Server stellt (i.d.R. zustandslose [stateless]) Operationen zur Verfügung
- Technologien: SOAP (JAX-RPC), XML-RPC

OO-RPC:

- Server hostet Set von Objekten
- Verteilte Objekte haben typischerweise einen eigenen Zustand
- Technologien: RMI, Corba

#### 1.4.2 Nachrichtenbasierte Systeme

- Informationsaustausch durch Nachrichten, typischerweise asynchron
- Sender & Empfänger getrennt durch Message Queue Service
- Sender & Empfänger müssen sich nicht kennen
- Technologien: SOAP (JAX-WS), JMS, Akka

## 2 Java Networking

### 2.1 IP Socket Verbindungen

Auf dem Server muss ein Service auf einem speziellen Port auf eingehende Requests warten. Wenn der Client sich zu einem Server Service verbindet, muss er seine eigene Adresse und Port bekannt geben um eine Antwort zu erhalten.

Hostname / IP# identifiziert Host – genutzt von IP (L3)

Port# identifiziert Applikation auf dem Host – genutzt von TCP/UDP (L4)

### 2.2 Class InetAddress

#### 2.2.1 Statische Factory Methoden

- `getByName(String name)`
- `getByAddress(4/16 bytes)`
- `getAllByName(String host)`
- `getLocalHost()`

### 2.3 Instanzmethoden

- `byte[] getAddress()`
- `String.getHostAddress()`
- `String.getHostName()`
- `String.getCanonicalHostName()`
- `boolean isReachable(int timeout)`
- `boolean isMulticastAddress()`

### 2.4 Class NetworkInterface

Erlaubt Zugriff auf die lokalen Netzwerkinterfaces.

### 2.5 Stream Socket

- Permanent und zuverlässige Verbindung zwischen 2 Maschinen (TCP)
- Socket erstellt In- & Outputstream
- Nach Übertragung schliessen eine oder beide Seiten die Verbindung
- Lowest-Level Form der Kommunikation aus Entwicklersicht
  - o Programmierer ist verantwortlich den Flow der Bytes zwischen den Maschinen zu managen
  - o Higher-Level Techniken:
    - Nachrichtenübermittlungssysteme (SOAP, JMS)
    - Erweiterungen zu Web Servern (Servlets, JSP, ASP, ...)
    - Verteilte Objekte (RMI)

### 2.6 Class Socket

Für das Aufbauen von Verbindungen zu einem offenen ServerSocket.

Methoden:

- `InetAddress.getInetAddress()`
- `InetAddress.getLocalAddress()`
- `int.getPort()`

- `int getLocalPort()`
- `InputStream getInputStream()`
- `OutputStream getOutputStream()`
- `void close()`
- `void shutdownOutput()`
- `void shutdownInput()`

## 2.7 `class ServerSocket`

- Läuft auf einem Server und wartet auf eingehende Verbindungen
- Gebunden an IP Adresse und Port
- Nur ein Prozess pro Port!

Methoden:

- `Socket accept()`            `// blockiert bis eine Verbindung angefordert wird`

## 2.8 `ObjectStreams`

- Klasse muss `java.io.Serializable` interface implementieren
  - o Alternativ kann sie auch `Externalizable` implementieren

## 2.9 `Class DatagramSocket`

- UDP

Methoden:

- `connect(addr, port)`    `// erlaubt Senden/Empfangen nur für angegebenen Host`
- `disconnect()`           `// löst Restriktion auf`
- `getInetAddress()`       `// gibt Restriktions-Host zurück (oder Null)`
- `getPort()`
- `isConnected()`           `// gibt zurück ob restringiert`
- `close()`                 `// gibt an, dass Socket nicht mehr in Verwendung`
- `send(DatagramPacket)`
- `receive(DatagramPacket)`

## 3 Internet

### 3.1 HTTP (HyperText Transfer Protocol)

[http://www.tutorialspoint.com/http/http\\_messages.htm](http://www.tutorialspoint.com/http/http_messages.htm)

- Stateless
- Basiert auf TCP/IP

#### 3.1.1 Request

```
POST /test/demo_form.asp HTTP/1.1 // Request (method path version)
Host: w3schools.com // Header Lines (header: value <CRLF>)
Content-Type: text/plain // "
Content-Length: x // "
// <CRLF>
name1=value1&name2=value2 // Message Body (hier spez. POST Query)
```

#### 3.1.2 Response

```
HTTP/1.1 200 OK // Response (version status )
Date: Fri, 31 Dec 1999 23:59:59 GMT // Header Lines
Content-Type: text/html
Content-Length: 1354
Host: w3schools.com

<html> // Response Body
<body>
<h1>Header</h1>
(more file contents)
.
</body>
</html>
```

### 3.2 Common Header Lines

Header-Line	Description
<b>Allow</b>	Liste von unterstützten http Methoden (Allow: GET,HEAD)
<b>Content-Encoding</b>	In erster Linie genutzt um komprimierte Dokumente mit ihrem ursprünglichen Media Type zu versenden. (Content-Encoding: x-gzip)
<b>Content-Length</b>	Gibt die Länge des Message Bodys an
<b>Content-Type</b>	Siehe MIME (Content-Type: text/plain)
<b>Date</b>	(Date: Tue, 15 Nov 1994 08:12:31 GMT)
<b>Expires</b>	Gibt Datum & Zeit ab wann das Objekt nicht mehr gültig ist
<b>From</b>	E-Mail Adresse der zuständigen Person (From: hans@müller.ch)
<b>If-Modified-Since</b>	Konditionelle GET Requests. Falls nicht erfüllt wird bloss 304 (not modified) zurückgegeben ohne Body.
<b>Location</b>	Verwendet mit Redirects (Status 3xx) um den neuen Ort der Ressource anzugeben.
<b>Pragma</b>	(Pragma = "Pragma" ":" 1#pragma-directive) (pragma-directive = "no-cache"   extension-pragma) (extension-pragma = token [ "=" word ])
<b>Referer</b>	Gibt an von woher die URL aufgerufen wurde.
<b>User-Agent</b>	Information über den User-Agent der den Request verursacht hat.



### 3.2.1 MIME (Multipurpose Internet Mail Extensions)

Format :        media-type = type / subtype { « ; » parameter }

z.B.            `text-html; charset=ISO-8859-1`

Liste gültiger MIME Types: <http://www.iana.org/assignments/media-types>

## 3.3 HTTP Status Codes

### 3.3.1 1xx: Information

Message:	Description:
<b>100 Continue</b>	Only a part of the request has been received by the server, but as long as it has not been rejected, the client should continue with the request
<b>101 Switching Protocols</b>	The server switches protocol

### 3.3.2 2xx: Successful

Message:	Description:
<b>200 OK</b>	The request is OK
<b>201 Created</b>	The request is complete, and a new resource is created
<b>202 Accepted</b>	The request is accepted for processing, but the processing is not complete
<b>203 Non-authoritative Information</b>	
<b>204 No Content</b>	
<b>205 Reset Content</b>	
<b>206 Partial Content</b>	

### 3.3.3 3xx: Redirection

Message:	Description:
<b>300 Multiple Choices</b>	A link list. The user can select a link and go to that location. Maximum five addresses
<b>301 Moved Permanently</b>	The requested page has moved to a new url
<b>302 Found</b>	The requested page has moved temporarily to a new url
<b>303 See Other</b>	The requested page can be found under a different url
<b>304 Not Modified</b>	
<b>305 Use Proxy</b>	
<b>306 Unused</b>	This code was used in a previous version. It is no longer used, but the code is reserved
<b>307 Temporary Redirect</b>	The requested page has moved temporarily to a new url

### 3.3.4 4xx: Client Error

Message:	Description:
<b>400 Bad Request</b>	The server did not understand the request
<b>401 Unauthorized</b>	The requested page needs a username and a password
<b>402 Payment Required</b>	<i>You can not use this code yet</i>
<b>403 Forbidden</b>	Access is forbidden to the requested page
<b>404 Not Found</b>	The server can not find the requested page
<b>405 Method Not Allowed</b>	The method specified in the request is not allowed
<b>406 Not Acceptable</b>	The server can only generate a response that is not accepted by the client
<b>407 Proxy Authentication Required</b>	You must authenticate with a proxy server before this

	request can be served
<b>408 Request Timeout</b>	The request took longer than the server was prepared to wait
<b>409 Conflict</b>	The request could not be completed because of a conflict
<b>410 Gone</b>	The requested page is no longer available
<b>411 Length Required</b>	The "Content-Length" is not defined. The server will not accept the request without it
<b>412 Precondition Failed</b>	The precondition given in the request evaluated to false by the server
<b>413 Request Entity Too Large</b>	The server will not accept the request, because the request entity is too large
<b>414 Request-url Too Long</b>	The server will not accept the request, because the url is too long. Occurs when you convert a "post" request to a "get" request with a long query information
<b>415 Unsupported Media Type</b>	The server will not accept the request, because the media type is not supported
<b>416</b>	
<b>417 Expectation Failed</b>	

### 3.3.5 5xx: Server Error

Message:	Description:
<b>500 Internal Server Error</b>	The request was not completed. The server met an unexpected condition
<b>501 Not Implemented</b>	The request was not completed. The server did not support the functionality required
<b>502 Bad Gateway</b>	The request was not completed. The server received an invalid response from the upstream server
<b>503 Service Unavailable</b>	The request was not completed. The server is temporarily overloading or down
<b>504 Gateway Timeout</b>	The gateway has timed out
<b>505 HTTP Version Not Supported</b>	The server does not support the "http protocol" version

## 3.4 GET vs POST

	GET	POST
<b>BACK button/Reload</b>	Harmless	Data will be re-submitted (the browser should alert the user that the data are about to be re-submitted)
<b>Bookmarked</b>	Can be bookmarked	Cannot be bookmarked
<b>Cached</b>	Can be cached	Not cached
<b>Encoding type</b>	application/x-www-form-urlencoded	application/x-www-form-urlencoded or multipart/form-data. Use multipart encoding for binary data
<b>History</b>	Parameters remain in browser	Parameters are not saved in

	history	browser history
<b>Restrictions on data length</b>	Yes, when sending data, the GET method adds the data to the URL; and the length of a URL is limited (maximum URL length is 2048 characters)	No restrictions
<b>Restrictions on data type</b>	Only ASCII characters allowed	No restrictions. Binary data is also allowed
<b>Security</b>	<p>GET is less secure compared to POST because data sent is part of the URL</p> <p>Never use GET when sending passwords or other sensitive information!</p>	POST is a little safer than GET because the parameters are not stored in browser history or in web server logs
<b>Visibility</b>	Data is visible to everyone in the URL	Data is not displayed in the URL

## 4 Webservices

### 4.1 XML-RPC (Remote Procedure Call)

XML-RPC verwendet ebenfalls XML um RPCs zu kodieren. Verglichen mit der Architektur von SOAP hat es eine simplere Architektur. XML-RPC ist eher bescheiden in seinen Zielen. Es versucht nicht die Lösung für jedes Problem zu sein, stattdessen versucht es eine einfach und effektiv zu sein in dem was es kann – das Anfragen und Erhalten von Informationen.

#### 4.1.1 Primitive Datentypen

- int, i4 signed 32bit Integer
- string ASCII string (no latin1)
- boolean either 0 or 1
- double double-precision floating point number
- dateTime.iso8601 z.B. 20050717T14:08:14
- base64 raw binary data, base64 encoded

```
<i4 >13 </i4 >  
<boolean >0 </ boolean >
```

#### 4.1.2 Structs

- Struct enthält Members mit Name und Wert.
- können rekursiv sein (Structs die Structs enthalten)

```
<struct >  
  <member >  
    <name >from </ name >  
    <value ><i4 > -5 </i4 ></ value >  
  </ member >  
  <member >  
    <name >to </ name >  
    <value ><i4 >5 </i4 ></ value >  
  </ member >  
</ struct >
```

#### 4.1.3 Arrays

Element-Typen können gemischt werden

```
<array >  
  <data >  
    <value ><i4 >-5 </i4 ></ value >  
    <value >< string >44 </ string ></ value >  
    <value >< boolean >1 </ boolean ></ value >  
  </data >  
</array >
```

#### 4.1.4 XML-RPC Request

```
<? xml version = " 1.0 " encoding ="UTF -8"?>  
<methodCall >  
  <methodName > Echo . getEcho </ methodName >
```

```
<params >
  <param >
    <value >World </ value >
  </param >
</ params >
</ methodCall >
```

#### 4.1.5 XML-RPC Response

##### 4.1.5.1 Single Result

```
<? xml version = " 1.0 " encoding = "UTF -8"?>
<methodResponse >
  <params >
    <param >
      <value > Hello World , welcome to XML -RPC </ value >
    </param >
  </ params >
</ methodResponse >
```

Als Resultat kann nur ein Wert zurückkommen, dieser kann jedoch auch ein Struct oder ein Array sein.

##### 4.1.6 Fault Result

```
<? xml version = " 1.0 " encoding = "UTF -8"?>
<methodResponse >
  <fault >
    <value >
      <struct >
        <member >
          <name > faultCode </ name >
          <value ><i4 >0 </i4 ></ value >
        </ member >
        <member >
          <name > faultString </ name >
          <value >No such handler : Echo .foo </ value >
        </ member >
      </ struct >
    </value >
  </fault >
</ methodResponse >
```

##### 4.1.7 Apache XML-RPC Sample Server

```
import org . apache . xmlrpc . server . * ;
import org . apache . xmlrpc . webserver . WebServer ;
3
public class HelloServer {
  public static void main ( String [] args ) throws Exception {
    PropertyHandlerMapping phm = new PropertyHandlerMapping ();
    phm . addHandler ( " Echo ", ch . fhnw . ds . xmlrpc . echo . EchoImpl .
class );
    WebServer server = new WebServer ( 80 ) ;
    XmlRpcServer xmlRpcServer = server . getXmlRpcServer ();
    xmlRpcServer . setHandlerMapping ( phm );
    server . start ();
    System . out . println ( " Server started at port 80 " );
  }
}
```

```
}
```

#### 4.1.7.1 Handler Class Server

```
public class EchoImpl {
    public String getEcho ( String name ) {
        return "[XML - RPC ] Hello " + name + ", welcome to XML - RPC ";
    }
}
```

Nur Instanzmethoden der Handlerklasse sind zugreifbar. Keine void Methoden. Public Default Constructor zwingend.

#### 4.1.8 Apache XML-RPC Client

```
import java . util . * ;
import org . apache . xmlrpc . * ;
public class HelloClient {
    public static void main ( String [] args ) throws Exception {
        XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl () ;
        config . setServerURL ( new URL ( " http :// localhost / xmlrpc " ) ) ;
        XmlRpcClient client = new XmlRpcClient () ;
        client . setConfig ( config ) ;
        List params = new ArrayList () ;
        params . add ( args [0] ) ;
        Object result = client . execute ( " Echo . getEcho " , params ) ;
        System . out . println ( " The result is : " + result . toString () ) ;
    }
}
```

## 4.2 SOAP

- Einfaches, schlankes Protokoll um strukturierte und typisierte Informationen im Web auszutauschen
- Protokolle: HTTP, SMTP, raw TCP/IP, ...
- Format: XML
- Message Typ: one-way oder Request / Response
- Body: kann zusätzliche Meta Infos enthalten
- Design Ziel: auf bereits existierenden und akzeptierten Standards aufbauen

### 4.2.1 WSDL

WSDL ist eine XML basierte Interface Beschreibungssprache die genutzt wird um die Funktionalität von Webservices zu beschreiben. Eine WSDL Beschreibung eines Webservices enthält:

- Wie der Service aufgerufen werden kann
- Welche Parameter er erwartet
- Welche Datenstruktur er zurückgibt

WSDL 1.1 Term	WSDL 2.0 Term	Description
<b>Service</b>	<b>Service</b>	Contains a set of system functions that have been exposed to the Web-based protocols.
<b>Port</b>	<b>Endpoint</b>	Defines the address or connection point to a Web service. It is typically

		represented by a simple HTTP URL string.
<b>Binding</b>	<b>Binding</b>	Specifies the interface and defines the SOAP binding style ( <a href="#">RPC</a> /Document) and transport (SOAP Protocol). The binding section also defines the operations.
<b>PortType</b>	<b>Interface</b>	Defines a Web service, the operations that can be performed, and the messages that are used to perform the operation.
<b>Operation</b>	<b>Operation</b>	Defines the SOAP actions and the way the message is encoded, for example, "literal." An operation is like a method or function call in a traditional programming language.
<b>Message</b>	<b>n/a</b>	Typically, a message corresponds to an operation. The message contains the information needed to perform the operation. Each message is made up of one or more logical parts. Each part is associated with a message-typing attribute. The message name attribute provides a unique name among all messages. The part name attribute provides a unique name among all the parts of the enclosing message. Parts are a description of the logical content of a message. In RPC binding, a binding may reference the name of a part in order to specify binding-specific information about the part. A part may represent a parameter in the message; the bindings define the actual meaning of the part. Messages were removed in WSDL 2.0, in which XML schema types for defining bodies of inputs, outputs and faults are referred to simply and directly.
<b>Types</b>	<b>Types</b>	Describes the data. The <a href="#">XML Schema</a> language (also known as XSD) is used (inline or referenced) for this purpose.

#### 4.2.2 JAX-WS (Java API for XML Web Services)

JAX-WS ist eine Java API um Webservices zu erstellen. Es ist Teil der Java EE (Enterprise Edition) Plattform von Sun Microsystems. Wie auch andere Java EE APIs nutzt JAX-WS Annotationen (@WebService, @WebMethod usw). Basiert auf SOAP. Nur WSDL 1.1 unterstützt.

##### 4.2.2.1 Anleitung

###### 4.2.2.1.1 Interface erstellen

Ein Interface heisst SEI (Service Endpoint Interface)

```
package ch.fhnw.ds.jaxws.server;
import javax.jws.WebService;
@WebService
public interface HelloService {
    String sayHello(@WebParam(name = "name") String name);
}
```

###### 4.2.2.1.2 Interface implementieren

Die Implementation heisst SIB (Service Implementation Bean)

```
package ch.fhnw.ds.jaxws.server;
import java.util.Date;
@WebService
public class HelloServiceImpl implements HelloService {
    @Override
    public String sayHello(@WebParam(name = "name") String name){
        return "Hello " + name + " from SOAP at " + new Date();
    }
}
```

```
}
```

#### 4.2.2.1.3 Java Objekte für XML Requests & Responses generieren

```
% wsgen -cp bin -keep -s src -d bin  
ch.fhnw.ds.jaxws.server.HelloServiceImpl
```

--cp <path> classpath

--keep keep generated files

--s <path> path where to place generated source files

--d <path> path where to place generated output files

--SEI specify a SIB (service implementation bean)

##### 4.2.2.1.3.1 SayHello

```
package ch.fhnw.ds.jaxws.server.jaxws;  
import ...;  
@XmlElement(name = "sayHello",  
            namespace = "http://server.jaxws.ds.fhnw.ch/")  
@XmlAccessorType(XmlAccessType.FIELD)  
@XmlType(name = "sayHello",  
        namespace = "http://server.jaxws.ds.fhnw.ch/")  
public class SayHello {  
  
    @XmlElement(name = "name", namespace = "")  
    private String name;  
    public String getName() { return this.name; }  
    public void setName(String name) { this.name = name; }  
}
```

##### 4.2.2.1.3.2 SayHelloResponse

```
package ch.fhnw.ds.jaxws.server.jaxws;  
import ...;  
@XmlElement(name = "sayHelloResponse",  
            namespace = "http://server.jaxws.ds.fhnw.ch/")  
@XmlAccessorType(XmlAccessType.FIELD)  
@XmlType(name = "sayHelloResponse",  
        namespace = "http://server.jaxws.ds.fhnw.ch/")  
public class SayHelloResponse {  
  
    @XmlElement(name = "return", namespace = "")  
    private String _return;  
    public String getReturn() { return this._return; }  
    public void setReturn(String _return) {  
        this._return = _return;  
    }  
}
```

##### 4.2.2.1.4 Publish the Service

```
package ch.fhnw.ds.jaxws.server;  
import javax.xml.ws.Endpoint;  
  
public class HelloServicePublisher {
```



```
public static void main(String[] args){
    Endpoint.publish(
        "http://127.0.0.1:9876/hs", // publication URI
        new HelloServiceImpl()); // SIB instance
    System.out.println("service published");
}
}
```

#### 4.2.2.2 Generierte Webservice Definition anschauen

http://localhost:9876/hs?wsdl

```
<?xml version="1.0" encoding="UTF-8"?> <definitions
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://server.jaxws.ds.fhnw.ch/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://server.jaxws.ds.fhnw.ch/"
  name="HelloServiceImplService">
<types>
  <xsd:schema>
    <xsd:import namespace="http://server.jaxws.ds.fhnw.ch/"
schemaLocation="http://localhost:9876/hs?xsd=1">
    </xsd:import>
  </xsd:schema>
</types>
<message name="sayHello">
<part name="parameters"
element="tns:sayHello">
</part>
</message>
<message name="sayHelloResponse">
<part name="parameters"
element="tns:sayHelloResponse">
</part>
</message>
<portType name="HelloServiceImpl">
<operation name="sayHello">
<input message="tns:sayHello"></input>
<output message="tns:sayHelloResponse"></output>
</operation>
</portType>
<binding name="HelloServiceImplPortBinding"
type="tns:HelloServiceImpl">
<soap:binding
transport="http://schemas.xmlsoap.org/soap/http"
style="document">
</soap:binding>
<operation name="sayHello">
<soap:operation soapAction=""></soap:operation>
<input>
<soap:body use="literal"></soap:body>
</input>
<output>
<soap:body use="literal"></soap:body>
```

```
</output>
</operation>
</binding>
<service name="HelloServiceImplService">
  <port name="HelloServiceImplPort"
    binding="tns:HelloServiceImplPortBinding">
    <soap:address location="http://localhost:9876/hs">
    </soap:address>
  </port>
</service>
</definitions>
```

Referenzierte Schema Definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:tns="http://server.jaxws.ds.fhnw.ch/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  version="1.0"
  targetNamespace="http://server.jaxws.ds.fhnw.ch/">
  <xs:element name="sayHello" type="tns:sayHello"></xs:element> <xs:element
    name="sayHelloResponse" type="tns:sayHelloResponse"></xs:element>
  <xs:complexType name="sayHello"> <xs:sequence> <xs:element name="name"
    type="xs:string" minOccurs="0"></xs:element> </xs:sequence>
  </xs:complexType> <xs:complexType name="sayHelloResponse"> <xs:sequence>
    <xs:element name="return" type="xs:string" minOccurs="0"></xs:element>
  </xs:sequence>
  </xs:complexType>
</xs:schema>
```

#### 4.2.2.3 Client Proxy generieren

```
% wsimport -keep -p ch.fhnw.ds.jaxws.client.jaxws -d bin -s src
http://localhost:9876/hs?wsdl
```

- keep keep generated files
- p <package> specify (overwrite) target package
- s <path> path where to place generated source files
- d <path> path where to place generated output files
- <WSDL> Web Service Definition
- > HelloServiceImpl generated interface
- > HelloServiceImplService factory class

#### 4.2.2.4 Client Applikation schreiben

```
package ch.fhnw.imvs.client;
import ch.fhnw.imvs.client.jaxws>HelloServiceImpl;
import ch.fhnw.imvs.client.jaxws>HelloServiceImplService;

public class Client {
    public static void main(String[] args) {
        HelloServiceImplService service =
```

```
        new HelloServiceImplService();
        HelloServiceImpl port =
            service.getHelloServiceImplPort();
        String result = port.sayHello("Dominik");
        System.out.println(result);
    }
}
```

#### 4.2.3 JAX-WS HTTP Request

```
POST /hs HTTP/1.1                                HTTP Request
Accept: text/xml, multipart/related
User-Agent: JAX-WS RI 2.2.4-b01
Host: 127.0.0.1:9877
Connection: keep-alive
Content-Length: 209
Content-Type: text/xml; charset=utf-8            SOAP HTTP Binding
SOAPAction:
"http://server.jaxws.ds.fhnw.ch/HelloServiceImpl/sayHelloRequest"

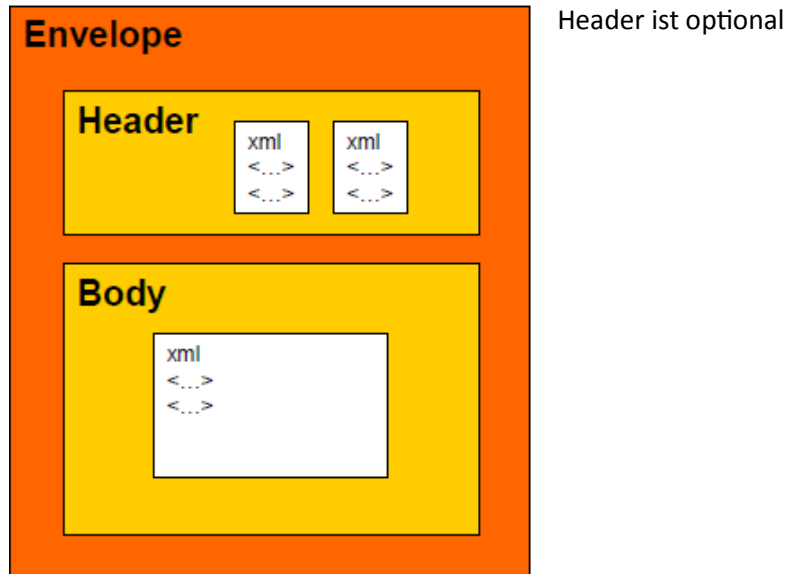
<?xml version="1.0" ?>                          SOAP Payload
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:sayHello xmlns:ns2="http://server.jaxws.ds.fhnw.ch/">
      <name>Dominik</name>
    </ns2:sayHello>
  </S:Body>
</S:Envelope>
```

#### 4.2.4 JAX-WS HTTP Response

```
HTTP/1.1 200 OK                                  HTTP Response
Transfer-encoding: chunked
Content-type: text/xml; charset=utf-8
Date: Mon, 18 Mar 2013 00:06:44 GMT
Content-Length: 277

<?xml version="1.0" ?>                          SOAP Payload
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
<ns2:sayHelloResponse
xmlns:ns2="http://server.jaxws.ds.fhnw.ch/">
<return>
Hello Dominik from SOAP at Mon Mar 18 01:06:44 CET 2013
</return>
</ns2:sayHelloResponse>
</S:Body>
</S:Envelope>
```

#### 4.2.5 SOAP Message Structure



### 4.3 Vergleich von SOAP und XML-RPC

Quelle: <http://weblog.masukomi.org/2006/11/21/xml-rpc-vs-soap>

#### 4.3.1 Allgemein

- SOAP produziert mehr Overhead
- XML-RPC ermöglicht beinahe alles zu erledigen mit möglichst wenig Aufwand
- SOAP bietet viel mehr Optionen und Möglichkeiten

#### 4.3.2 Features

Feature	XML-RPC	SOAP
<b>Structs</b>	Yes	Yes
<b>Arrays</b>	Yes	Yes
<b>Named structs &amp; arrays</b>	No	Yes
<b>Short learning curve</b>	Yes	No
<b>Developer specified character set</b>	No	Yes
<b>Developer defined data types</b>	No	Yes
<b>Can specify recipient</b>	No	Yes
<b>Require client understanding</b>	No	Yes
<b>Message specific processing instructions</b>	No	yes

## 5 REST (Representational State Transfer)

REST ist eine relativ neue Variante um verteilte Systeme aufzubauen. Der Begriff wurde erstmals im Jahr 2000 in einer Doktorarbeit verwendet.

### 5.1 Prinzipien

- Jede Ressource ist adressierbar über eine URI
- Als Requesttypen werden die bestehenden http Requests verwendet:
  - o GET, HEAD → READ
    - Informationen über eine Repräsentation erhalten
    - Keine Nebeneffekte, möglicherweise gecached
    - Kann Query Parameter enthalten
  - o POST → CREATE
    - Neue Sub-Ressource erstellen ohne bekannte ID
  - o PUT → CREATE & UPDATE
    - Existierende Ressource updaten oder
    - Neue Ressource mit bekannter ID erstellen
  - o DELETE
    - Ressource(n) löschen
  - o OPTIONS
    - Erlaubte Operationen zurückgeben
- Repräsentationsorientiert, d.h. es werden mehrere Repräsentationen einer Ressource erlaubt
  - o text/html
  - o text/plain
  - o application/json
  - o application/xml
- Dinge verknüpfen
  - o Referenzen zu anderen Ressourcen können in Repräsentationen genutzt werden

```
<order>
<
date>16.03.2013</
date>
<amount>23</amount>
<product ref="http://example.com/products/4711" />
<customer ref="http://example.com/customers/1234" />
</order>
```

- HTTP Protokoll
  - o Standard Antwortcodes (201 Created, 404 Not Found, 405 Method not Allowed)
  - o Caching

### 5.2 REST in 5 Schritten

1. Allem eine ID geben (in URI Form)
2. Dinge verknüpfen (Ressource-Referenzen)
3. Standardmethoden verwenden (http POST/GET/PUT/DELETE)
4. Verschiedene Repräsentationen
  - a. Inhaltverhandlung
  - b. URL basierte Repräsentationen
5. Zustandslose Kommunikation

### 5.3 REST vs SOAP

REST	SOAP
<ul style="list-style-type: none"> <li>–Resource Oriented</li> <li>–Messages represented in different formats</li> <li>–HTTP used as protocol</li> <li>–HTTP verbs are used for access and manipulation</li> <li>–HTTP error codes are used as error messages</li> <li>–No formal interface description language (=&gt; WADL)</li> <li>–GET requests can be cached in a proxy</li> </ul>	<ul style="list-style-type: none"> <li>–Service oriented</li> <li>–Messages represented in XML</li> <li>–Can be bound to different protocols</li> <li>–Access and Manipulation is service specific</li> <li>–Fault Elements in SOAP body describes errors</li> <li>–WSDL is used as interface description language</li> <li>–All requests are POST requests, no caching!</li> </ul>

### 5.4 REST Client Seite

Für einen REST Client wird eine HTTP Client Library benötigt.

- java.net.HttpURLConnection
- Jakarta Commons HttpClient Library (http Components Project)
- Jersey Client Library
- RESTEasy Client Library
- cURL (Command Line Tool um http Requests zu übermitteln)

### 5.5 REST Server Seite

### 5.6 JSR 311: Java API for RESTful Web Services (JAX-RS)

Es gibt verschiedene Implementationen von JAX-RS:

- Jersey: Reference Implementation
- JBoss RESTEasy JAX-RS
- CXF
- Wink
- Restlet

```
@Singleton /* eine Instanz handelt alle Requests, sonst wird eine neue
Instanz erstellt für jeden Request */
@Path("/polls")
public class ... {
    @GET
    @Path("/{id}")
    public String getPoll(@PathParam("id") String key) { ... }
}
```

#### 5.6.1 HTTP Methoden

- Mit den Annotationen @GET, @POST, @PUT, @DELETE, @HEAD wird die http Methode angegeben
  - Der Name der Java Methode kann frei gewählt werden
  - Eigene http Methoden können definiert werden (WebDAV) indem neue Annotationen verwendet werden
- ```
@HttpMethod("LOCK")
```

#### 5.6.2 JAX-RS Injection

| Typ | Beschreibung |
|-----|--------------|
|-----|--------------|

|                    |                                                           |
|--------------------|-----------------------------------------------------------|
| <b>PathParam</b>   | Werte aus den URI Template Parametern zu extrahieren      |
| <b>MatrixParam</b> | Exakte Matrix Parameter ( /images/cars;color=blue/2012/ ) |
| <b>QueryParam</b>  | Query Parameter aus der URI ( /images/cars?color=blue )   |
| <b>FormParam</b>   | Werte aus geposteten Formulardaten                        |
| <b>HeaderParam</b> | Request Header Daten                                      |
| <b>CookieParam</b> | Werte aus HTTP cookies                                    |

```
@Path("/hello")
public class HelloResource {
    //hello/meier-3?size=20

    @GET
    @Path("{name}-{id}")
    public String getHello(
        @PathParam("name") String name,
        @PathParam("id") int id,
        @QueryParam("size") int size,
        @HeaderParam("Referer") String referer,
        @CookieParam("customerId") Cookie custId) {
        ...
    }
}
```

#### 5.6.2.1 Automatische Typkonvertierung

Strings werden automatisch zu ihren primitiven Typen konvertiert (int, short, float, double, byte, char, Boolean).

#### 5.6.2.2 Default-Werte

Es können Default Werte definiert werden für den Fall, dass der Parameter nicht mit dem Request mitgegeben wird.

```
@DefaultValue(10) @QueryParam("size") int size,
```

#### 5.6.2.3 Kontext

Javax.ws.rs.core.Context erlaubt es verschiedene Helfer Objekte einzufügen.

|                          |                                                                                                                              |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------|
| @Context UriInfo         | getBaseUri, getPath, getAbsolutePath, getQueryParameters                                                                     |
| @Context HttpHeaders     | getRequestHeaders, getRequestHeader, getMediaType, getLanguage, get Cookies, getAcceptableLanguages, getAcceptableMediaTypes |
| @Context Request         | getMethod, evaluatePrecondition                                                                                              |
| @Context SecurityContext |                                                                                                                              |
| @Context Providers       |                                                                                                                              |

#### 5.6.3 Content Negotiation

@Produces: deklariert den Typ des Rückgabewertes (default: alle Rückgabewerte unterstützt)

```
@Path("/polls") // sets the path for this service
public class DoodleResource {
    @GET @Path("{id}")
    @Produces({"text/plain", "text/html"})
    public String getPollAsText(@PathParam("id") String id){
        ...
    }
}
```

```
@GET @Path("/{id}")
@Produces("application/xml")
public Poll getPoll (@PathParam("id") String id){
    ...
}
```

@Consumes: deklariert akzeptierte Typen (PUT/POST)

```
@Path("/polls") // sets the path for this service
public class DoodleResource {
    @PUT @Path("/{id}")
    @Consumes("application/x-www-form-urlencoded")
    @Produces("text/html")
    public String setPollForm(@PathParam("id") String id,
                             @FormParam("name") String name ){
        ...
    }
    @PUT @Path("/{id}")
    @Consumes("application/xml")
    public void setPoll(@PathParam("id") String id, Poll p){
        ...
    }
}
```

## 5.6.4 JAX-RS Data Binding

### 5.6.4.1 @Provider

- Provider Interface implementieren um proprietäre Medientypen zu unterstützen
- Klasse mit @Provider markieren und
  - o MessageBodyReader<T>
    - readFrom: inputStream => T
  - o MessageBodyWriter<T>
    - writeTo:

## 5.6.5 JAX-RS Antworten

### 5.6.5.1 Standardantworten

- 200 OK (wenn kein Wert zurückgegeben wird)
- 204 No Content (für void Methoden)
- 404 Not Found (wenn kein Ressourcen-Handler gefunden wurde)
- 406 Not Acceptable (wenn eine angeforderte Repräsentation nicht angeboten wird)
- 405 Method not allowed (wenn HTTP Methode nicht unterstützt wird)

### 5.6.5.2 WebApplicationException

Errorstatus kann definiert werden.

### 5.6.5.3 Exception Mapping

Es kann ein Mapper definiert werden, welcher Application Exceptions zu Antworten mappt

### 5.6.5.4 Komplexe Antworten

Das Builder Pattern wird benutzt um Antworten zu erstellen. ResponseBuilder wird durch statische Methoden als Antwort (status(), ok(), noContent(), notModified()...) erstellt.



### 5.6.6 JAX-RS Cache Control

```
Date lastMod= ...; // get the timestamp
CacheControl cc = new CacheControl();
cc.setMaxAge(1000); // HTTP max-age field, in seconds
ResponseBuilder builder = request.evaluatePreconditions(lastMod);
if(builder != null){
    builder.cacheControl(cc);
    return builder.build(); // returns 304 Not Modified
}
builder = Response.ok(response, type);

builder.cacheControl(cc);
builder.lastModified(lastMod);
return builder.build();
```

#### 5.6.6.1 Cache-Control Header

- Private: caching in zwischengeschalteten Proxies ist nicht erlaubt
- Public: Antwort kann von jeder Zwischenstelle gecached werden
- No-cache: Antwort soll nicht gecached werden. Kann gespeichert werden, aber muss reevaluiert werden
- No-store: Antwort wird nicht auf der Disk gespeichert
- No-transform: Antwort wird nicht in einer veränderten Form gespeichert (z.b. komprimiert)
- Max-age: definiert wie lange ein Cache gültig ist (in Sekunden)

### 5.6.7 JAX-RS Concurrency

Da alle Aktionen auf der Client Seite ausgeführt werden, müssen wir Versionenkonflikte verhindern. Versionierung wird am einfachsten mithilfe des ETag oder des Änderungsdatums realisiert.

```
Date lastMod = ...; // get the timestamp
EntityTag tag = new EntityTag(...); // compute hash code
ResponseBuilder builder =
request.evaluatePreconditions(
lastMod, tag
);
if(builder != null){
    return builder.build();
// returns 412 Precondition Failed
}
... perform the update ...
return Response.noContent().build();
```

## 5.7 JAX-RS Deployment

### 5.7.1 Server-Implementationen

- Jersey
- JBoss
- CXF
- Wink
- Restlet

## 6 Glossar

### 6.1 CRLF (Carriage Return [and] Line Feed)

In der Informatik ist CRLF ein Zeilenumbruch, auch bekannt als end-of-line (EOL). CRLF ist ein Sonderzeichen. Darauf folgender Text wird auf einer neuen Zeile angezeigt.

## 7 Code Beispiele

### 7.1 StringBuilder

```
public String decorateTheString(String orgStr){
    StringBuilder builder = new StringBuilder();
    builder.append(orgStr);
    builder.deleteCharAt(orgStr.length()-1);
    builder.insert(0,builder.hashCode());
    return builder.toString();
}
```

### 7.2 Netzwerkinterfaces ausgeben

```
public static void main(String[] args) throws SocketException {
    Enumeration<NetworkInterface> interfaces =
        NetworkInterface.getNetworkInterfaces();
    while(interfaces.hasMoreElements()){
        NetworkInterface intf = interfaces.nextElement();
        System.out.print(intf.getName());
        System.out.println(" ["+intf.getDisplayName()+"]");
        Enumeration<InetAddress> adr = intf.getInetAddresses();
        while(adr.hasMoreElements()){
            System.out.println("\t" + adr.nextElement());
        }
        byte[] hardwareAddress = intf.getHardwareAddress();
    }
}
```

### 7.3 Example WSDL 2.0 code

```
<?xml version="1.0" encoding="UTF-8"?>
<description xmlns="http://www.w3.org/ns/wsd1"
    xmlns:tns="http://www.tmsws.com/wsd120sample"
    xmlns:whhttp="http://schemas.xmlsoap.org/wsd1/http/"
    xmlns:wsoap="http://schemas.xmlsoap.org/wsd1/soap/"
    targetNamespace="http://www.tmsws.com/wsd120sample">

<!-- Abstract type -->
    <types>
        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns="http://www.tmsws.com/wsd120sample"
            targetNamespace="http://www.example.com/wsd120sample">

            <xs:element name="request">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="header" maxOccurs="unbounded">
```

```
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:string">
              <xs:attribute name="name" type="xs:string"
use="required"/>
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="body" type="xs:anyType" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="method" type="xs:string"
use="required"/>
    <xs:attribute name="uri" type="xs:anyURI" use="required"/>
  </xs:complexType>
</xs:element>

  <xs:element name="response">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="header" maxOccurs="unbounded">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute name="name" use="required"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="body" type="xs:anyType" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="status-code" type="xs:anySimpleType"
use="required"/>
      <xs:attribute name="response-phrase" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
</types>

<!-- Abstract interfaces -->
<interface name="RESTfulInterface">
  <fault name="ClientError" element="tns:response"/>
  <fault name="ServerError" element="tns:response"/>
  <fault name="Redirection" element="tns:response"/>
  <operation name="Get" pattern="http://www.w3.org/ns/wsd1/in-out">
    <input messageLabel="In" element="tns:request"/>
    <output messageLabel="Out" element="tns:response"/>
  </operation>
  <operation name="Post" pattern="http://www.w3.org/ns/wsd1/in-out">
    <input messageLabel="In" element="tns:request"/>
    <output messageLabel="Out" element="tns:response"/>
  </operation>
  <operation name="Put" pattern="http://www.w3.org/ns/wsd1/in-out">
    <input messageLabel="In" element="tns:request"/>
    <output messageLabel="Out" element="tns:response"/>
  </operation>
</interface>
```

```
</operation>
<operation name="Delete" pattern="http://www.w3.org/ns/wsd1/in-out">
  <input messageLabel="In" element="tns:request"/>
  <output messageLabel="Out" element="tns:response"/>
</operation>
</interface>

<!-- Concrete Binding Over HTTP -->
<binding name="RESTfulInterfaceHttpBinding"
interface="tns:RESTfulInterface"
  type="http://www.w3.org/ns/wsd1/http">
  <operation ref="tns:Get" whttp:method="GET"/>
  <operation ref="tns:Post" whttp:method="POST"
    whttp:inputSerialization="application/x-www-form-
urlencoded"/>
  <operation ref="tns:Put" whttp:method="PUT"
    whttp:inputSerialization="application/x-www-form-
urlencoded"/>
  <operation ref="tns:Delete" whttp:method="DELETE"/>
</binding>

<!-- Concrete Binding with SOAP-->
<binding name="RESTfulInterfaceSoapBinding"
interface="tns:RESTfulInterface"
  type="http://www.w3.org/ns/wsd1/soap"
  wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"
  wsoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-
response">
  <operation ref="tns:Get" />
  <operation ref="tns:Post" />
  <operation ref="tns:Put" />
  <operation ref="tns:Delete" />
</binding>

<!-- Web Service offering endpoints for both bindings-->
<service name="RESTfulService" interface="tns:RESTfulInterface">
  <endpoint name="RESTfulServiceHttpEndpoint"
    binding="tns:RESTfulInterfaceHttpBinding"
    address="http://www.example.com/rest/" />
  <endpoint name="RESTfulServiceSoapEndpoint"
    binding="tns:RESTfulInterfaceSoapBinding"
    address="http://www.example.com/soap/" />
</service>
</description>
```