

Software Engineering 1

Übung: XML inklusive XML Schema und Netzwerkkommunikation

Kristof Böhmer
Fakultät für Informatik
Universität Wien

Universität Wien

29

Fakultät für Informatik
Institut für Informatik und
Formale Sprachen (IFS)



1.1 XML: Grundlagen

1.2 XML: Definition

1.3 XML: Schema

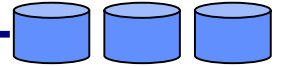
1.4 Netzwerkkommunikation: Einführung

1.5 Netzwerkkommunikation: REST

1.6 Netzwerkkommunikation: REST Implementierung

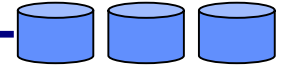
1.7 Netzwerkkommunikation: Message Bus, Sockets, SOAP

1.1 XML: Grundlagen – Motivation 1



- ❑ **Webservices:** Standardformat wird benötigt um Daten einfacher austauschen zu können.
 - Einfache Strings sind nicht immer ausreichend.
- ❑ **Beispiel:** Die Informationen über einen Uni Kurs auslesen
 - GET Anfrage an <http://www.example.com/courses>
 - Soll zurückgeben:
 - ◆ Name
 - ◆ Universität
 - ◆ Fakultät

1.1 XML: Grundlagen – Motivation 2

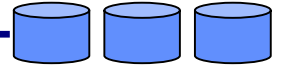


- ❑ **Möglichkeit:** Die Informationen als reinen String zu codieren ist technisch machbar, aber ineffizient.
 - Codierung beispielsweise über Trennzeichen.
 - Nachteil: Trennzeichen darf nicht im Text vorkommen.
 - Nicht zwingend eindeutig, welche Information was bedeutet.

```
"Software Engineering|Universität Wien|Fakultät für Informatik"
```

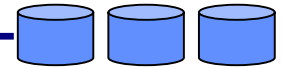
- ❑ **Alternative:** XML
- ❑ **Warum wird in der Lehrveranstaltung XML eingesetzt:** Weit verbreiteter Standard, sehr gute Toolunterstützung, XML Schemata erlauben detaillierte Format Definitionen, Namespaces und strikte Validierung möglich.

1.1 XML: Grundlagen – Einsatzmöglichkeiten und Alternativen



- ❑ **Einsatzmöglichkeiten:** XML kann nicht nur für Webservices eingesetzt werden, sondern auch:
 - Zum Speichern von Daten und auch das Analysieren der Daten durch auf XML ausgerichtete Tools ist möglich.
- ❑ **Alternativen:** XML ist nicht der einzige Weg, um strukturierte Daten zu übertragen.
 - Eine Alternative wäre beispielsweise JSON (JavaScript Objekt Notation).
 - ◆ Kompaktere Syntax als XML.
 - ◆ Aber keine "offizielle" Möglichkeit um zu definieren, wie eine Nachricht bzw. Daten aufgebaut sein müssen (Vgl. mit XML Schema).

1.1 XML: Grundlagen – Benötigte Software



- ❑ **Benötigte Software:** Keine spezielle Software notwendig – XML kann in einem beliebigem Texteditor erstellt werden.
 - Alternative: Spezielle Software, die das Designen von XML Dokumenten vereinfacht, wie beispielsweise der XML Editor in Eclipse.

Node	Content
?? xml	version="1.0" encoding="UTF-8"
DOCTYPE	hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://www.hib...
hibernate-configuration	(session-factory, security?)
session-factory	(property*, mapping*, (class-cache collection-cache)*, event*, listener*)
property	
property	
property	
property	
property	
property	
property	
property	
--	<property name="hibernate.hbm2ddl.auto">update</property>
mapping	
mapping	



1.1 XML: Grundlagen

1.2 XML: Definition

1.3 XML: Schema

1.4 Netzwerkkommunikation: Einführung

1.5 Netzwerkkommunikation: REST

1.6 Netzwerkkommunikation: REST Implementierung

1.7 Netzwerkkommunikation: Message Bus, Sockets, SOAP

1.2 XML: Definition



- ❑ **XML Definition:** Ist eine Abkürzung für eXtensible Markup Language
 - Kann für das Speichern und Übertragen von Daten verwendet werden.
 - ◆ Beispielsweise für Netzwerkschnittstellen.
 - Können mit jedem beliebigem Texteditor geöffnet und bearbeitet werden.
 - Ist eine W3C Rekommandation.
- ❑ **Beispiel eines XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<kurs>
  <name>Software Engineering</name>
  <universitaet>Universität Wien</universitaet>
  <fakultaet>Fakultät für Informatik</fakultaet>
</kurs>
```


1.2 XML: Definition – Eigenschaften von XML Dokumenten 1



□ XML: Ist als Baumstruktur aufgebaut

- Verschachtelungen sind über mehrere Ebenen möglich.
- Es existiert immer genau ein Root Tag (in diesem Beispiel: <kurs>).
- Öffnendes und schließendes Tag müssen denselben Namen haben.
- Reihenfolge öffnender und schließender Tags muss beibehalten werden.

□ Beispiel eines XML

```
<?xml version="1.0" encoding="UTF-8" />
<kurs>
  <name>Software Engineering</name>
  <universitaet>Universität Wien</universitaet>
  <fakultaet>Fakultät für Informatik</fakultaet>
</kurs>
```

Diagram illustrating the XML structure with annotations:

- Öffnendes Tag** (Opening Tag) points to `<kurs>`.
- Schließendes Tag** (Closing Tag) points to `</kurs>`.

1.2 XML: Definition – Eigenschaften von XML Dokumenten 2



□ XML: Ist als Baumstruktur aufgebaut

- Verschachtelungen sind über mehrere Ebenen möglich.
- Es existiert immer genau ein Root Tag (in diesem Beispiel: <kurs>).
- Öffnendes und schließendes Tag müssen denselben Namen haben.
- Reihenfolge öffnender und schließender Tags muss beibehalten werden.

□ Beispiel eines XML

```
<?xml version="1.0" encoding="UTF-8"?>
<kurs>
  <name>Software Engineering</name>
  <universitaet>Universität Wien</universitaet>
  <fakultaet>Fakultät für Informatik</fakultaet>
</kurs>
```

Öffnendes Tag

Schließendes Tag

1.2 XML: Definition – Eigenschaften von XML Dokumenten 3



□ XML: Ist als Baumstruktur aufgebaut

- Verschachtelungen sind über mehrere Ebenen möglich.
- Es existiert immer genau ein Root Tag (in diesem Beispiel: <kurs>).
- Öffnendes und schließendes Tag müssen denselben Namen haben.
- Reihenfolge öffnender und schließender Tags muss beibehalten werden.

□ Beispiel eines XML

```
<?xml version="1.0" encoding="UTF-8"?>
<kurs>
  <name>Software Engineering</name>
  <universitaet>Universität Wien</universitaet>
  <fakultaet>Fakultät für Informatik</fakultaet>
</kurs>
```

Diagramm: Ein blauer Pfeil zeigt von einem Kasten mit der Aufschrift "Inhalt eines Tags" auf den Text "Software Engineering" im XML-Code.

1.2 XML: Definition – Eigenschaften von XML Dokumenten 4



❑ XML: Ist als Baumstruktur aufgebaut

- Verschachtelungen sind über mehrere Ebenen möglich.
- Es existiert immer genau ein Root Tag (in diesem Beispiel: <kurs>).
- Öffnendes und schließendes Tag müssen denselben Namen haben.
- Reihenfolge öffnender und schließender Tags muss beibehalten werden.

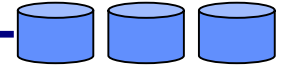
❑ Beispiel eines fehlerhaften XML

```
<?xml version="1.0" encoding="UTF-8"?>
<kurs>
  <name>Software Engineering</name>
  <universitaet>Universität Wien</uni>
  <fakultaet>Fakultät für Informatik</kurs>
</fakultaet>
```

- Command Line Tool: xmllint
- Online: https://www.w3schools.com/xml/xml_validator.asp

XML-Verarbeitungsfehler: Nicht übereinstimmendes Tag. Erwartet: </universitaet>.
Adresse: https://www.w3schools.com/xml/xml_validator.asp
Zeile Nr. 4, Spalte 35:
 <universitaet>Universität Wien</uni>

1.2 XML: Definition – Attribute



- ❑ **Attribute:** Jedes XML Tag kann eines oder mehrere Attribute haben.
 - Attribute werden immer im Anschluss an den Namen im XML Tag definiert.
 - Syntax: `attributname = "wert"`
 - Jeder Attributname kann pro Tag nur 1x auftreten.

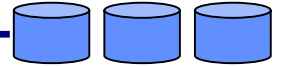
❑ **Gültiger Einsatz von Attributen:**

```
<kurs semester="4">Software Engineering</kurs>
```

❑ **Ungültiger Einsatz von Attributen:**

```
<semester="4" kurs semester="4">Software  
Engineering</kurs>
```

```
<kurs semester="4" semester="6">Software  
Engineering</kurs>
```



XML: Definition

NAMENSRÄUME

1.2 XML: Definition – Namensräume (Namespaces) 1



- ❑ **Namensräume:** Werden XML Dokumente verschiedener Projekte / Entwickler gemeinsam verwendet, kann es passieren, dass Tags öfter vorkommen und verschiedene Bedeutungen haben, siehe:

```
<?xml version="1.0" encoding="UTF-8"?>
<kurs>
  <name>Software Engineering</name>
  <universitaet>Universität Wien</universitaet>
</kurs>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kurs>
  <raumschiff>USS Enterprise</raumschiff>
  <richtung>14 mark 68</richtung>
</kurs>
```

1.2 XML: Definition – Namensräume (Namespaces) 2



□ Beispiel für Namensräume:

- **Ausgangslage:** Zwei getrennte XML Dokumente wie in der vorhergehen Folie skizziert.
- **Aufgabe:** Man möchte diese beiden getrennten Dokumente zu einem zusammenfügen. Was ist das Problem das dabei auftritt?
- **Problem:** Es befinden sich nun zwei Tags mit der Bezeichnung Kurs in einem XML Dokument. Allerdings haben diese eine komplett unterschiedliche Bedeutung.
- **Lösung:** Für jeden Präfix einen Namensraum definieren und diesen Präfix bei jedem Element anführen, wodurch diese unterscheidbar werden.
 - ◆ Der Präfix kann entweder über das **xmlns Attribut** bei den jeweiligen Elementen oder direkt für alle Elemente im Root Element definiert werden.

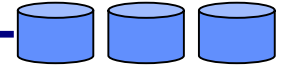
1.2 XML: Definition – Namensräume (Namespaces) 3



- ❑ **Namensräume:** Zusammengefügte XML Dokumente ohne der Verwendung von Namespaces.

```
<?xml version="1.0" encoding="UTF-8"?>
<kurse>
  <kurs>
    <name>Software Engineering</name>
    <universitaet>Universität Wien</universitaet>
  </kurs>
  <kurs>
    <raumschiff>USS Enterprise</raumschiff>
    <richtung>14 mark 68</richtung>
  </kurs>
</kurse>
```

1.2 XML: Definition – Namensräume (Namespaces) 4



- ❑ **Namensräume:** Zusammengefügte XML Dokumente unter Verwendung von Namespaces – Definition beim jeweiligen Element.

```
<?xml version="1.0" encoding="UTF-8"?>
<kurse>
  <u:kurs xmlns:u="https://www.univie.ac.at/kurse">
    <u:name>Software Engineering</u:name>
    <u:universitaet>Universität Wien</u:universitaet>
  </u:kurs>
  <s:kurs xmlns:s="https://www.univie.ac.at/startrek">
    <s:raumschiff>USS Enterprise</s:raumschiff>
    <s:richtung>14 mark 68</s:richtung>
  </s:kurs>
</kurse>
```

1.2 XML: Definition – Namensräume (Namespaces) 5



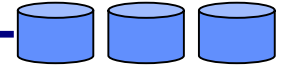
- ❑ **Namensräume:** Zusammengefügte XML Dokumente unter Verwendung von Namespaces – Definition beim jeweiligen Element.

```
<?xml version="1.0" encoding="UTF-8" ?>
<kurse>
  <u:kurs xmlns:u="https://www.univie.ac.at/kurse">
    <u:name>Software Engineering</u:name>
    <u:universitaet>Universität Wien</u:universitaet>
  </u:kurs>
  <s:kurs xmlns:s="https://www.univie.ac.at/startrek">
    <s:raumschiff>USS Enterprise</s:raumschiff>
    <s:richtung>14 mark 68</s:richtung>
  </s:kurs>
</kurse>
```

Definition des Namensraums für einen Präfix

Präfix eines Elements

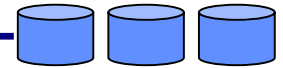
1.2 XML: Definition – Namensräume (Namespaces) 6



- ❑ **Namensräume:** Zusammengefügte XML Dokumente unter Verwendung von Namespaces – Definition im Root Element.

```
<?xml version="1.0" encoding="UTF-8"?>
<kurse xmlns:u="https://www.univie.ac.at/kurse"
      xmlns:s="https://www.univie.ac.at/startrek">
  <u:kurs>
    <u:name>Software Engineering</u:name>
    <u:universitaet>Universität Wien</u:universitaet>
  </u:kurs>
  <s:kurs>
    <s:raumschiff>USS Enterprise</s:raumschiff>
    <s:richtung>14 mark 68</s:richtung>
  </s:kurs>
</kurse>
```

1.2 XML: Definition – Namensräume (Namespaces) 7



- **Namensräume:** Zusammengefügte XML Dokumente unter Verwendung von Namespaces – Definition im Root

```
<?xml version="1.0" encoding="UTF-8"?>
<kurse xmlns:u="https://www.univie.ac.at/kurse"
      xmlns:s="https://www.univie.ac.at/startrek">
  <u:kurs>
    <u:name>Software Engineering</u:name>
    <u:universitaet>Universität Wien</u:universitaet>
  </u:kurs>
  <s:kurs>
    <s:raumschiff>USS Enterprise</s:raumschiff>
    <s:richtung>14 mark 68</s:richtung>
  </s:kurs>
</kurse>
```

Definition des Namenraums für einen Präfix

Präfix eines Elements

1.2 XML: Definition – Namensräume (Namespaces) 8



- ❑ **Namensräume:** Die URI (z.B. <https://www.univie.ac.at/kurse>) verleiht dem Namensraum einen einzigartigen Namen.
 - Sie muss nicht auf eine existierende Ressource verweisen.
 - Manchmal verweist sie auf eine Webseite, die den Namespace näher beschreibt, dies ist allerdings nicht verpflichtend.
- ❑ **Achtung:** Es ist nicht ausreichend nur den Präfix vor einem Element anzuführen (u:kurs) sondern es muss vorher immer ein xmlns Attribut (Namensraum) dafür definiert werden.



XML: Definition

XML PARSING IN JAVA



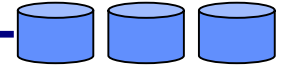
❑ JAXB – Java Architektur für XML Binding

- Konvertiert XML in Java Objekte
- Unterstützt alle XML Features
- Binding Java zu XML ("marshalling") und umgekehrt ("unmarshalling")
- Binding von XML Schemata
- *Wird im Rahmen der Übung und im Spring Framework eingesetzt.*

❑ SAX / DOM

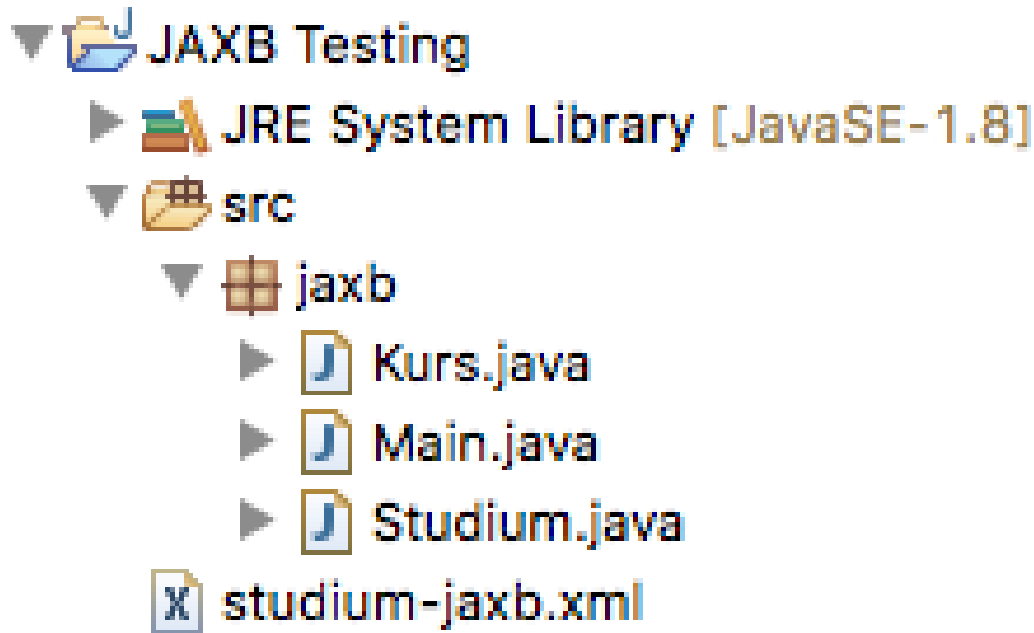
- Lower Level API zum Zugriff auf XML Daten.
- Ermöglicht das Iterieren über Elemente der XML Dokumente ohne diese auf Java Objekte zu mappen.
- Relativ aufwändig zu verwenden, dafür hat es nur einen geringen Bedarf an Rechenleitung und Arbeitsspeicher.

1.2 XML: Definition – XML Parsing in Java 2



□ JAXB – Beispiel

- Einfaches Eclipse Projekt mit 3 Klassen:
 - ◆ Kurs.java beinhaltet Infos über einen Kurs an der Uni.
 - ◆ Studium.java beinhaltet Infos über ein Studium.
 - ◆ Main.java beinhaltet die main(String[] args) Funktion.



1.2 XML: Definition – XML Parsing in Java 3



□ JAXB – Beispiel

- Gewünschtes Endergebnis: Parsen von XML in Objekte ("unmarshalling") und von Objekten in XML ("marshalling").

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<studium>
  <kursListe>
    <kurs>
      <kursName>Software Engineering 1</kursName>
      <leiter>Kristof Böhmer</leiter>
      <ects>6</ects>
    </kurs>
    <kurs>
      <kursName>Programmierung 1</kursName>
      <leiter>Helmut Wanek</leiter>
      <ects>6</ects>
    </kurs>
  </kursListe>
  <bezeichnung>Bachelor Informatik</bezeichnung>
  <ort>Universität Wien</ort>
</studium>
```

1.2 XML: Definition – XML Parsing in Java 4



□ JAXB – Beispiel: Kurs.java

```
1 package jaxb;
2
3 import javax.xml.bind.annotation.XmlAccessType;
4 import javax.xml.bind.annotation.XmlAccessorType;
5 import javax.xml.bind.annotation.XmlRootElement;
6 import javax.xml.bind.annotation.XmlType;
7
8 @XmlAccessorType(XmlAccessType.FIELD)
9 @XmlRootElement()
10 @XmlType(propOrder = { "kursName", "leiter", "ects" })
11 public class Kurs {
12
13     private String kursName;
14     private String leiter;
15     private int ects;
16
17     public String getName() {
18         return this.kursName;
19     }
20
21     public void setName(String _name) {
22         this.kursName = _name;
23     }
24
25     public String getLeiter() {
26         return leiter;
27     }
28
29     public void setLeiter(String _leiter) {
30         this.leiter = _leiter;
31     }
32
33     public int getECTS() {
34         return ects;
35     }
36
37     public void setECTS(int _ects) {
38         this.ects = _ects;
39     }
40 }
```

1.2 XML: Definition – XML Parsing in Java 5



□ JAXB – Beispiel: Studium.java

```
1 package jaxb;
2
3 import java.util.ArrayList;
4
5 import javax.xml.bind.annotation.XmlAccessType;
6 import javax.xml.bind.annotation.XmlElement;
7 import javax.xml.bind.annotation.XmlElementWrapper;
8 import javax.xml.bind.annotation.XmlRootElement;
9 import javax.xml.bind.annotation.XmlAccessorType;
10
11 @XmlAccessorType(XmlAccessType.FIELD)
12 @XmlRootElement()
13 public class Studium {
14
15     @XmlElementWrapper(name = "kursListe")
16     @XmlElement(name = "kurs")
17     private ArrayList<Kurs> kursListe;
18     private String bezeichnung;
19     private String ort;
20
21     public void setKurse(ArrayList<Kurs> kursListe) {
22         this.kursListe = kursListe;
23     }
24
25     public ArrayList<Kurs> getKurse() {
26         return kursListe;
27     }
28
29     public String getBezeichnung() {
30         return bezeichnung;
31     }
32
33     public void setBezeichnung(String _bezeichnung) {
34         this.bezeichnung = _bezeichnung;
35     }
36
37     public String getLocation() {
38         return ort;
39     }
40
41     public void setLocation(String _ort) {
42         this.ort = _ort;
43     }
44 }
```


1.2 XML: Definition – XML Parsing in Java 6



□ JAXB – Beispiel: Main.java

- Kümmert sich um (un)marshalling.
- Schritt 1: Importieren der nötigen Libraries.
- Schritt 2: Erstellen der Kurse und des Studiums.

```
1 package jaxb;
2
3 import java.io.File;
4 import java.io.FileReader;
5 import java.io.IOException;
6 import java.util.ArrayList;
7
8 import javax.xml.bind.JAXBContext;
9 import javax.xml.bind.JAXBException;
10 import javax.xml.bind.Marshaller;
11 import javax.xml.bind.Unmarshaller;
12
13 public class Main {
14
15     private static final String STUDIUM_XML = "./studium-jaxb.xml";
16
17     public static void main(String[] args) throws JAXBException, IOException {
18
19         ArrayList<Kurs> liste = new ArrayList<Kurs>();
20
21         // create books
22         Kurs kurs1 = new Kurs();
23         kurs1.setName("Software Engineering 1");
24         kurs1.setLeiter("Kristof Böhmer");
25         kurs1.setECTS(6);
26         liste.add(kurs1);
27
28         Kurs kurs2 = new Kurs();
29         kurs2.setName("Programmierung 1");
30         kurs2.setLeiter("Helmut Wanek");
31         kurs2.setECTS(6);
32         liste.add(kurs2);
33
34         Studium studium = new Studium();
35         studium.setBezeichnung("Bachelor Informatik");
36         studium.setLocation("Universität Wien");
37         studium.setKurse(liste);
38     }
39 }
```

1.2 XML: Definition – XML Parsing in Java 7



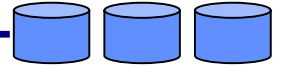
□ JAXB – Beispiel: Main.java

○ Schritt 3: Marshalling

- ◆ Wandelt Objekte in XML um.
- ◆ Ausgabe in System.out (Zeile 45) und in eine Datei (Zeile 48).

```
39      // create JAXB context and instantiate marshaller
40      JAXBContext context = JAXBContext.newInstance(Studium.class);
41      Marshaller m = context.createMarshaller();
42      m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
43
44      // Write to System.out
45      m.marshal(studium, System.out);
46
47      // Write to File
48      m.marshal(studium, new File(STUDIUM_XML));
49
50      // get variables from our xml file, created before
51      System.out.println();
52      System.out.println("Output from our XML File: ");
```

1.2 XML: Definition – XML Parsing in Java 8



❑ JAXB – Beispiel: Main.java

○ Schritt 4: Unmarshalling

◆ Wandelt XML in Objekte um

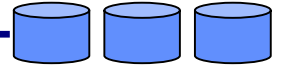
```
50 // get variables from our xml file, created before
51 System.out.println();
52 System.out.println("Output from our XML File: ");
53 Unmarshaller um = context.createUnmarshaller();
54 Studium studium2 = (Studium) um.unmarshal(new FileReader(STUDIUM_XML));
55 ArrayList<Kurs> list = studium2.getKurse();
56 for (Kurs kurs : list) {
57     System.out.println("Kurs: " + kurs.getName() + " geleitet von " + kurs.getLeiter());
58 }
59 }
60 }
```

1.2 XML: Definition – XML Parsing in Java 9



❑ JAXB – Beispiel: Ergebnis in Datei `studium-jaxb.xml`

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<studium>
  <kursListe>
    <kurs>
      <kursName>Software Engineering 1</kursName>
      <leiter>Kristof Böhmer</leiter>
      <ects>6</ects>
    </kurs>
    <kurs>
      <kursName>Programmierung 1</kursName>
      <leiter>Helmut Wanek</leiter>
      <ects>6</ects>
    </kurs>
  </kursListe>
  <bezeichnung>Bachelor Informatik</bezeichnung>
  <ort>Universität Wien</ort>
</studium>
```



1.1 XML: Grundlagen

1.2 XML: Definition

1.3 XML: Schema

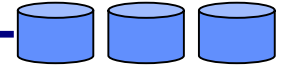
1.4 Netzwerkkommunikation: Einführung

1.5 Netzwerkkommunikation: REST

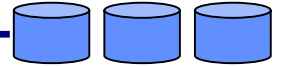
1.6 Netzwerkkommunikation: REST Implementierung

1.7 Netzwerkkommunikation: Message Bus, Sockets, SOAP

1.3 XML: Schema – Definition



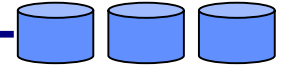
- ❑ **XML Schema** (auch **XSD** – XML Schema Definition): Die Struktur von XML Dokumenten kann beliebig sein, daher wird ein Schema benötigt das deren Struktur zu beschreibt.
 - Das heißt es wird definiert, welche Tags in einem Dokument wie oft vorkommen müssen.
 - Das XML Schema ist eigentlich auch nur ein XML Dokument.
- ❑ **Vorgänger: DTD** (Document Type Definition)
 - Nicht so umfangreich wie XML Schema.
- ❑ **Alternative:** Relax NG (ist im Vergleich bequemer zu definieren)
 - Aber: XML Schema hat eine gute (bessere) Toolunterstützung in Java und ist auch ein offizieller W3C Standard.
- ❑ **Tutorials, Schema Standard und Quellen:**
https://www.w3schools.com/xml/schema_howto.asp



XML: Schema

BEISPIEL

1.3 XML: Schema – Beispiel 1



□ Folgendes XML Dokument ist gegeben:

```
<?xml version="1.0" encoding="UTF-8"?>
<kurs xmlns="https://www.univie.ac.at/kurse">
  <name>Software Engineering</name>
  <universitaet>Universität Wien</universitaet>
</kurs>
```

- **Ziel von XSD:** Beschreibt die Struktur dieses XML Dokuments, damit ein potentieller Empfänger weiß, was ihn erwartet. Hierdurch könnten Daten auch vor der eigentlichen Verarbeitung validiert werden.

1.3 XML: Schema – Beispiel 2



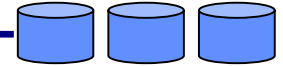
□ XML Schema zu einem gegebenem XML Dokument:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="https://www.univie.ac.at/kurse"
            xmlns="https://www.univie.ac.at/kurse"
            elementFormDefault="qualified">

  <xs:element name="kurs">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="universitaet" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>


</xs:schema>
```

1.3 XML: Schema – Beispiel 3



□ XML Schema zu einem gegebenem XML Dokument:

XML Namespace



```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="https://www.univie.ac.at/kurse"
  xmlns="https://www.univie.ac.at/kurse"
  elementFormDefault="qualified">

  <xs:element name="kurs">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="universitaet" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

1.3 XML: Schema – Beispiel 4



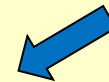
□ XML Schema zu einem gegebenem XML Dokument:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="https://www.univie.ac.at/kurse"
            xmlns="https://www.univie.ac.at/kurse"
            elementFormDefault="qualified">

  <xs:element name="kurs">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="universitaet" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

XML Namespace
der erstellten
Elemente



1.3 XML: Schema – Beispiel 5



□ XML Schema zu einem gegebenem XML Dokument:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="https://www.univie.ac.at/kurse"
  xmlns="https://www.univie.ac.at/kurse"
  elementFormDefault="qualified">

  <xs:element name="kurs">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="universitaet" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```



Default
Namespace

1.3 XML: Schema – Beispiel 6



□ XML Schema zu einem gegebenem XML Dokument:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="https://www.univie.ac.at/kurse"
  xmlns="https://www.univie.ac.at/kurse"
  elementFormDefault="qualified">

  <xs:element name="kurs">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="universitaet" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Elemente müssen den
Namespace beinhalten

1.3 XML: Schema – Beispiel 7



□ XML Schema zu einem gegebenem XML Dokument:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="https://www.univie.ac.at/kurse"
  xmlns="https://www.univie.ac.at/kurse"
  elementFormDefault="qualified">

  <xs:element name="kurs"> ← Definiert ein Element
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="universitaet" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

1.3 XML: Schema – Beispiel 8



□ XML Schema zu einem gegebenem XML Dokument:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="https://www.univie.ac.at/kurse"
  xmlns="https://www.univie.ac.at/kurse"
  elementFormDefault="qualified">

  <xs:element name="kurs">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="universitaet" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Beinhaltet mehrere Elemente

1.3 XML: Schema – Beispiel 9



□ XML Schema zu einem gegebenem XML Dokument:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="https://www.univie.ac.at/kurse"
  xmlns="https://www.univie.ac.at/kurse"
  elementFormDefault="qualified">

  <xs:element name="kurs">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="universitaet" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Eine fix definierte Liste
verschiedener Elemente

1.3 XML: Schema – Beispiel 10



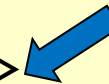
□ XML Schema zu einem gegebenem XML Dokument:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="https://www.univie.ac.at/kurse"
  xmlns="https://www.univie.ac.at/kurse"
  elementFormDefault="qualified">

  <xs:element name="kurs">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="universitaet" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

**Simple Types – einfache
Elemente, in diesem Fall
vom Typ string**



1.3 XML: Schema – Beispiel 11



- **Das XSD File kann in einem XML File entsprechend verlinkt sein:**

```
<?xml version="1.0" encoding="UTF-8"?>
<kurs xmlns = "https://www.univie.ac.at/kurse"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
      xsi:schemaLocation =
"https://www.univie.ac.at/kurse kurs.xsd">
  <name>Software Engineering</name>
  <universitaet>Universität Wien</universitaet>
</kurs>
```

1.3 XML: Schema – Beispiel 12



- **Das XSD File kann in einem XML File entsprechend verlinkt sein:**

```
<?xml version="1.0" encoding="UTF-8"?>
<kurs xmlns = "https://www.univie.ac.at/kurse"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
      xsi:schemaLocation =
"https://www.univie.ac.at/kurse kurs.xsd">
  <name>Software Engineering</name>
  <universitaet>Universität Wien</universitaet>
</kurs>
```



**Namespace des XML
Schemas**

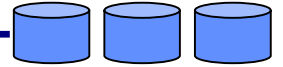
1.3 XML: Schema – Beispiel 13



- **Das XSD File kann in einem XML File entsprechend verlinkt sein:**

```
<?xml version="1.0" encoding="UTF-8"?>
<kurs xmlns = "https://www.univie.ac.at/kurse"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
      xsi:schemaLocation =
"https://www.univie.ac.at/kurse kurs.xsd">
  <name>Software Engineering</name>
  <universitaet>Universität Wien</universitaet>
</kurs>
```

2 Werte: Namespace +
Speicherort des XSD



XML: Schema

BESTANDTEILE

1.3 XML: Schema – XSD Simple Elements 1



- ❑ **XSD Simple Elements:** Können nur Text, aber keine anderen Elemente beinhalten.

```
<xs:element name="xxx" type="yyy"/>
```

- ❑ **Der Text kann dabei mehrere Typen haben:**

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

- ❑ **Beispiel**

```
<name>Schmidt</name>  
<alter>42</alter>  
<geburt>1975-03-27</geburt>
```

```
<xs:element name="name" type="xs:string"/>  
<xs:element name="age" type="xs:integer"/>  
<xs:element name="geburt" type="xs:date"/>
```

1.3 XML: Schema – XSD Simple Elements 2



- **XSD Simple Elements:** Können Standardwerte und Festwerte haben.
 - **Standardwerte:** Werden genommen, wenn kein anderer Wert gegeben ist.

```
<xs:element name="farbe" type="xs:string" default="red"/>
```

- **Festwerte:** Sind fixiert und können nicht geändert/überschrieben werden.

```
<xs:element name="farbe" type="xs:string" fixed="red"/>
```


1.3 XML: Schema – XSD Attribute



- ❑ **XSD Attribute:** Werden ähnlich wie Simple Elements definiert.
 - Die verwendbaren Typen sind ident zu denen der Simple Elements.

```
<xs:attribute name="xxx" type="yyy"/>
```

- Auch hier sind Standard- und Festwerte definierbar.

```
<xs:attribute name="lang" type="xs:string" fixed="DE"/>
```

```
<xs:attribute name="lang" type="xs:string" default="DE"/>
```

- **Required:** Attribute sind standardmäßig freiwillig. Wird ein Attribut unbedingt benötigt, so muss dies entsprechend deklariert werden.

```
<xs:attribute name="lang" type="xs:string" use="required"/>
```

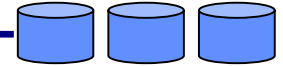
- ❑ Attribute können nicht zu Simple Elements hinzugefügt werden, dazu werden Complex Elements benötigt. Eine Erklärung dazu folgt später.

1.3 XML: Schema – XSD Restrictions 1



- ❑ **XSD Restrictions:** Ermöglichen die Definition von eigenen Typen, die bestimmte Restriktionen aufweisen.
 - Beispiel: "Alter" ist ein Integer, der zwischen 0 und 120 liegen muss.

```
<xs:element name="alter">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```



❑ XSD Restrictions: Beispiele

- In diesem Beispiel muss es sich um einen Audi oder BMW handeln.

```
<xs:element name="car">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Audi"/>
      <xs:enumeration value="BMW"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

- Selbstdefinierte Datentypen können auch wiederverwendet werden.

```
<xs:element name="auto" type="autoMarke"/>

<xs:simpleType name="autoMarke">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Audi"/>
    <xs:enumeration value="BMW"/>
  </xs:restriction>
</xs:simpleType>
```



❑ XSD Restrictions: Regular Expressions

- Es können auch beliebige Reguläre Ausdrücke eingesetzt werden.

```
<xs:element name="letter">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-z]" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```



❑ XSD Restrictions: Whitespaces

- Whitespaces (Leerzeichen, Tabulatoren etc.) können verschieden behandelt werden.
- Generell gibt es drei Möglichkeiten:
 - ◆ **Preserve:** Alles bleibt so, wie es im XML definiert ist.
 - ◆ **Replace:** Alle Whitespaces werden durch Leerzeichen ersetzt.
 - ◆ **Collapse:** Alle Whitespaces werden gelöscht.

```
<xs:element name="address">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="replace"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

1.3 XML: Schema – XSD Restrictions 5



- ❑ **XSD Restrictions:** Die Länge eines Wertes kann eingeschränkt werden.

- Exakte Länge

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:length value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

- Minimum / Maximum

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="5"/>
      <xs:maxLength value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

1.3 XML: Schema – XSD Complex Elements 1



- ❑ **XSD Complex Elements:** Können leer sein oder auch andere Elemente und/oder Attribute enthalten.

- Beispiel für ein leeres komplexes Element.

```
<product pid="1345"/>
```

- Beispiel für ein komplexes Element, das andere Elemente enthält.

```
<angestellter>  
  <name>John Smith</name>  
</angestellter>
```

- Beispiel für ein komplexes Element, das nur Text enthält.

```
<product pid="1345">Eis</product>
```

- Beispiel für ein komplexes Element, das sowohl Text als auch andere Elemente enthält.

```
<angestellter>  
  Der Angestellte heißt <name>John Smith</name>  
</angestellter>
```

1.3 XML: Schema – XSD Complex Elements 2



- ❑ **XSD Complex Elements:** Können direkt im Element definiert werden.

```
<xs:element name="angestellter">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="vorname" type="xs:string"/>
      <xs:element name="nachname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<angestellter>
  <vorname>John</vorname>
  <nachname>Smith</nachname>
</angestellter>
```


1.3 XML: Schema – XSD Complex Elements 3



- ❑ **XSD Complex Elements:** Können wiederverwendbar definiert werden.

```
<xs:element name="angestellter" type="personinfo"/>
<xs:element name="student" type="personinfo"/>
<xs:element name="mitglied" type="personinfo"/>

<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

```
<angestellter>
  <vorname>John</vorname>
  <nachname>Smith</nachname>
</angestellter>
<student>
  <vorname>Jane</vorname>
  <nachname>Doe</nachname>
</student>
```

1.3 XML: Schema – XSD Complex Elements 4



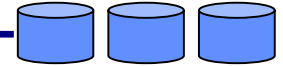
- ❑ **XSD Complex Elements:** Können auch aufeinander aufbauen.

```
<xs:element name="angestellter" type="fullpersoninfo"/>

<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="vorname" type="xs:string"/>
    <xs:element name="nachname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="fullpersoninfo">
  <xs:complexContent>
    <xs:extension base="personinfo">
      <xs:sequence>
        <xs:element name="address" type="xs:string"/>
        <xs:element name="city" type="xs:string"/>
        <xs:element name="country" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

1.3 XML: Schema – XSD Complex Elements 5



- ❑ **XSD Complex Elements:** Beispiel für Elemente, die ausschließlich Text beinhalten.

```
<schuhgroesse land="oesterreich">44</schuhgroesse>
```

- Es muss eine Extension für einen bestehenden Datentyp, oder eine Restriction auf einen bestehenden Datentyp definiert werden.
- Beispiel für ein mögliches Schema:

```
<xs:element name="schuhgroesse">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="land" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

1.3 XML: Schema – XSD Complex Elements 6



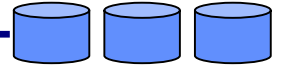
- ❑ **XSD Complex Elements:** Beispiel für gemischte Elemente, die Text, andere Elemente, und Attribute enthalten.

```
<brief>
  Lieber Herr <name>John Smith</name>
  Ihre Bestellung Nr. <orderid>12345</orderid> ist vom Paketdienst in
  der Nähe Ihrer Wohnung liegen gelassen worden. Viel Spaß beim Suchen.
</brief>
```

- Beispiel für ein mögliches Schema:

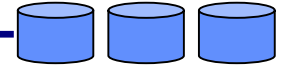
```
<xs:element name="letter">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="orderid" type="xs:positiveInteger"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

1.3 XML: Schema – XSD Indicators 1



- ❑ **XSD Indicators:** Definieren, wie Elemente im Dokument zu verwenden sind.
 - **Order Indicators**
 - ◆ All
 - ◆ Choice
 - ◆ Sequence
 - **Occurrence Indicators**
 - ◆ maxOccurs
 - ◆ minOccurs
 - **Group Indicators**
 - ◆ group name
 - ◆ attributeGroup name

1.3 XML: Schema – XSD Indicators 2



- ❑ **A11 Indicator:** Die Kind-Elemente können in beliebiger Reihenfolge auftreten, aber jedes Element darf genau 1x auftreten.
 - Beispiel für ein mögliches Schema:

```
<xs:element name="person">
  <xs:complexType>
    <xs:all>
      <xs:element name="vorname" type="xs:string"/>
      <xs:element name="nachname" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

1.3 XML: Schema – XSD Indicators 3



- ❑ **Choice Indicator:** Genau eines der Kind-Elemente kann auftreten.
 - Beispiel für ein mögliches Schema:

```
<xs:element name="person">
  <xs:complexType>
    <xs:choice>
      <xs:element name="angestellter" type="angestellter"/>
      <xs:element name="mitglied" type="mitglied"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

1.3 XML: Schema – XSD Indicators 4



- ❑ **Sequence Indicator:** Die Kind-Elemente müssen in der vorgegebenen Reihenfolge auftreten.
 - Beispiel für ein mögliches Schema:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="vorname" type="xs:string"/>
      <xs:element name="nachname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```


1.3 XML: Schema – XSD Indicators 5



- ❑ **maxOccurs, minOccurs Indicator:** Die Elemente mit diesen Indikatoren können mindestens `minOccurs`, und maximal `maxOccurs` vorkommen.
 - Es kann auch nur `minOccurs` oder `maxOccurs` definiert sein.
 - Beispiel für ein mögliches Schema:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="kind" type="xs:string"
        maxOccurs="10" minOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

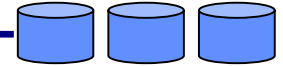
1.3 XML: Schema – XSD Indicators 6



- ❑ **maxOccurs, minOccurs Indicator:** Die Elemente mit diesen Indikatoren können mindestens `minOccurs`, und maximal `maxOccurs` vorkommen.
 - Beispiel eines gültigen XML zum vorhergehenden Schema.
 - ◆ Effekt des Schemas: mindestens ein Kind und maximal 10 Kinder dürfen bzw. müssen definiert werden.

```
<person>
  <name>John Smith</name>
  <kind>Jane Smith</kind>
  <kind>Jacob Smith</kind>
</person>
```

1.3 XML: Schema – XSD Indicators 7



- ❑ **group Indicator:** Gruppen können verwendet werden, um in anderen Definitionen wiederverwendet zu werden.
 - Innerhalb einer Gruppe muss ein `all`, `sequence` oder `choice` Element definiert sein.
 - Beispiel für ein mögliches Schema:

```
<xs:group name="personengruppe">
  <xs:sequence>
    <xs:element name="vorname" type="xs:string"/>
    <xs:element name="nachname" type="xs:string"/>
    <xs:element name="geburtsdatum" type="xs:date"/>
  </xs:sequence>
</xs:group>

<xs:element name="person" type="personinfo"/>

<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:group ref="personengruppe"/>
    <xs:element name="land" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

1.3 XML: Schema – XSD Indicators 8

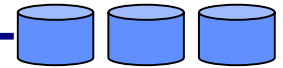


- ❑ **attributeGroup Indicator:** Ähnlich wie Gruppen für Elemente, können auch Attribute gruppiert werden.
 - Hier muss allerdings keine Reihenfolge (über `sequence`, `all` oder `choice`) angegeben werden.
 - Beispiel für ein mögliches Schema:

```
<xs:attributeGroup name="personattrgroup">
  <xs:attribute name="vorname" type="xs:string"/>
  <xs:attribute name="nachname" type="xs:string"/>
  <xs:attribute name="geburtsdatum" type="xs:date"/>
</xs:attributeGroup>

<xs:element name="person">
  <xs:complexType>
    <xs:attributeGroup ref="personattrgroup"/>
  </xs:complexType>
</xs:element>
```

1.3 XML: Schema – XSD <any>



- ❑ **XSD <any>**: Erlaubt es, ein beliebiges Element hinzuzufügen, welches nicht im Schema definiert sein muss.
 - Beispiel für ein mögliches Schema:

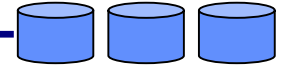
```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="vorname" type="xs:string"/>
      <xs:element name="nachname" type="xs:string"/>
      <xs:any minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

- Beispiel für ein mögliches XML:

```
<person>
  <vorname>John</vorname>
  <nachname>Smith</nachname>
  <hobby>Landhockey</hobby>
  <lieblingssessen>Schnitzel</lieblingssessen>
</person>
```

Kann in einem anderen
XSD File definiert sein

1.3 XML: Schema – XSD <anyAttribute>



- ❑ **XSD <anyAttribute>**: Ähnlich wie <any>, allerdings für Attribute.
 - Beispiel für ein mögliches Schema:

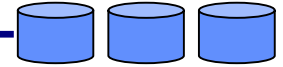
```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="vorname" type="xs:string"/>
      <xs:element name="nachname" type="xs:string"/>
    </xs:sequence>
    <xs:anyAttribute/>
  </xs:complexType>
</xs:element>
```

- Beispiel für ein mögliches XML:

```
<person augenfarbe="blau">
  <vorname>John</vorname>
  <nachname>Smith</nachname>
</person>
```

Kann in einem anderen
XSD File definiert sein

1.3 XML: Schema – XSD Element Ersetzung



□ **XSD Element Ersetzung:** Beispielsweise für zweisprachige Elemente.

- User sollen wählen können, ob sie deutsche oder englische Tagbezeichnungen verwenden wollen.

```
<xs:element name="firstname" type="xs:string"/>
<xs:element name="vorname" substitutionGroup="firstname"/>

<xs:complexType name="custinfo">
  <xs:sequence>
    <xs:element ref="name"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="customer" type="custinfo"/>
<xs:element name="kunde" substitutionGroup="customer"/>
```

○ Zugehörige XML Nachricht:

```
<customer>
  <firstname>John Smith</firstname>
</customer>
```

```
<kunde>
  <vorname>John Smith</vorname>
</kunde>
```

1.3 XML: XML & XSD Beispiel

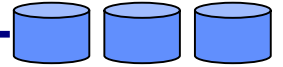


□ XML Schema für eine einfache Nachricht

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="kurs">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="universitaet" type="xs:string"/>
        <xs:element name="fakultaet" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

□ Zugehörige XML Nachricht

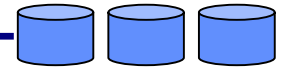
```
<?xml version="1.0" encoding="UTF-8"?>
<kurs>
  <name>Software Engineering</name>
  <universitaet>Universität Wien</universitaet>
  <fakultaet>Fakultät für Informatik</fakultaet>
</kurs>
```

XML: Schema

VALIDIERUNG

1.3 XML: Schema – Validierung 1



- ❑ **Validierung eines XML Schemas:** Über ein Command Line Tool wie beispielsweise xmllint.

```
$ xmllint --noout --schema schema.xsd file.xml
```



1.3 XML: Schema – Validierung 2



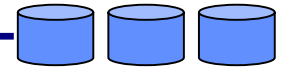
- ❑ **Validierung eines XML Schemas:** Über ein Command Line Tool wie beispielsweise xmllint.

```
$ xmllint --noout --schema schema.xsd file.xml
```



Keine Ausgabe
des result
trees

1.3 XML: Schema – Validierung 3



- ❑ **Validierung eines XML Schemas:** Über ein Command Line Tool wie beispielsweise xmllint.

```
$ xmllint --noout --schema schema.xsd file.xml
```



Validierung
gegen ein
XML Schema

1.3 XML: Schema – Validierung 4



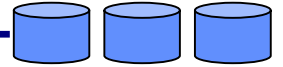
- ❑ **Validierung eines XML Schemas:** Über ein Command Line Tool wie beispielsweise xmllint.

```
$ xmllint --noout --schema schema.xsd file.xml
```



Schema Datei

1.3 XML: Schema – Validierung 5



- ❑ **Validierung eines XML Schemas:** Über ein Command Line Tool wie beispielsweise xmllint.

```
$ xmllint --noout --schema schema.xsd file.xml
```



XML Datei

1.3 XML: Schema – Validierung 6



- ❑ **Validierung eines XML Schemas:** Über ein Command Line Tool wie beispielsweise xmllint.

```
$ xmllint --noout --schema schema.xsd file.xml
```

- ❑ **Alternative:**

http://www.utilities-online.info/xsdvalidation/#.Wec_S4ppH0N

XML

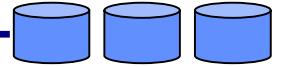
XSD Schema

Check XML Well Formed

Check XSD Validity

Validate XML against XSD

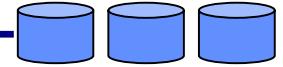
Result



XML: Schema

VALIDIERUNG IN JAVA

1.3 XML: Schema – Validierung in Java 1



❑ Schema Validierung in Java

- Ziel: XML File mit Schema validieren.
- Wir arbeiten mit dem zuvor erstellten Studium.xml File

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<studium>
  <kursListe>
    <kurs>
      <kursName>Software Engineering 1</kursName>
      <leiter>Kristof Böhmer</leiter>
    </kurs>
    <kurs>
      <kursName>Programmierung 1</kursName>
      <leiter>Helmut Wanek</leiter>
      <ects>6</ects>
    </kurs>
  </kursListe>
  <bezeichnung>Bachelor Informatik</bezeichnung>
  <ort>Universität Wien</ort>
</studium>
```

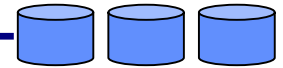
1.3 XML: Schema – Validierung in Java 2



❑ Schema Validierung in Java: Zugehöriges XSD File

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="studium">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="kursListe"/>
        <xs:element name="bezeichnung" type="xs:string"/>
        <xs:element name="ort" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="kursListe">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="kurs" minOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="kurs">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="kursName" type="xs:string"/>
        <xs:element name="leiter" type="xs:string"/>
        <xs:element name="ects" type="xs:integer"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

1.3 XML: Schema – Validierung mittels JAXB 1



□ Schema Validierung mittels JAXB

- Hinzufügen fehlender Abhängigkeiten (Dependencies)
- Hinzufügen der Schema Datei (Zeile 58)

```
3 import java.io.File;  
4 import java.io.FileReader;  
5 import java.io.IOException;  
6 import java.util.ArrayList;  
7  
8 import javax.xml.XMLConstants;  
9 import javax.xml.bind.JAXBContext;  
10 import javax.xml.bind.JAXBException;  
11 import javax.xml.bind.Marshaller;  
12 import javax.xml.bind.Unmarshaller;  
13 import javax.xml.bind.ValidationEvent;  
14 import javax.xml.bind.ValidationEventHandler;  
15 import javax.xml.validation.Schema;  
16 import javax.xml.validation.SchemaFactory;  
17  
18 import org.xml.sax.SAXException;
```

```
57 SchemaFactory sf = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);  
58 Schema schema = sf.newSchema(new File("studium.xsd"));  
59 Unmarshaller um = context.createUnmarshaller();  
60 um.setEventHandler(new StudiumValidationEventHandler());  
61 um.setSchema(schema);  
62 Studium studium2 = (Studium) um.unmarshal(new FileReader(STUDIUM_XML));
```

1.3 XML: Schema – Validierung mittels JAXB 2

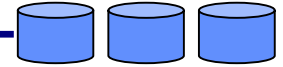


❑ Schema Validierung mittels JAXB

- Erstellen des Schema Validators
- Dieser „schweigt“, wenn das Schema erfolgreich validiert

```
70 class StudiumValidationEventHandler implements ValidationEventHandler {
71     @Override
72     public boolean handleEvent(ValidationEvent event) {
73         System.out.println("\nEVENT");
74         System.out.println("SEVERITY: " + event.getSeverity());
75         System.out.println("MESSAGE: " + event.getMessage());
76         System.out.println("LINKED EXCEPTION: " + event.getLinkedException());
77         System.out.println("LOCATOR");
78         System.out.println("    LINE NUMBER: " + event.getLocator().getLineNumber());
79         System.out.println("    COLUMN NUMBER: " + event.getLocator().getColumnNumber());
80         System.out.println("    OFFSET: " + event.getLocator().getOffset());
81         System.out.println("    OBJECT: " + event.getLocator().getObject());
82         System.out.println("    NODE: " + event.getLocator().getNode());
83         System.out.println("    URL: " + event.getLocator().getURL());
84         return true;
85     }
86 }
```

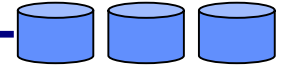
1.3 XML: Schema – Validierung mittels JAXB 3



❑ Schema Validierung mittels JAXB

- Erstellen des Schema Validators
- Dieser „schweigt“, wenn das Schema erfolgreich validiert.
- Validiert das Schema nicht erfolgreich, wird eine entsprechende Fehlermeldung ausgegeben.
- Beispielsweise wurde hier „bezeichnung“ durch „bezeichnung1“ ersetzt.

```
EVENT
SEVERITY: 2
MESSAGE: cvc-complex-type.2.4.a: Ungültiger Content wurde beginnend mit Element
"bezeichnung1" gefunden. "{bezeichnung}" wird erwartet.
LINKED EXCEPTION: org.xml.sax.SAXParseException; lineNumber: 15; columnNumber: 19;
cvc-complex-type.2.4.a: Ungültiger Content wurde beginnend mit Element
"bezeichnung1" gefunden. "{bezeichnung}" wird erwartet.
LOCATOR
  LINE NUMBER: 15
  COLUMN NUMBER: 19
  OFFSET: -1
  OBJECT: null
  NODE: null
  URL: null
```



1.1 XML: Grundlagen

1.2 XML: Definition

1.3 XML: Schema

1.4 Netzwerkkommunikation: Einführung

1.5 Netzwerkkommunikation: REST

1.6 Netzwerkkommunikation: REST Implementierung

1.7 Netzwerkkommunikation: Message Bus, Sockets, SOAP

1.4 Netzwurkkommunikation: Einführung



❑ Was ist Netzwurkkommunikation?

- Austausch von Information mehrerer Computer untereinander.

❑ Wofür braucht man es?

- Webseiten
- Serverbasierte Applikationen
- Web Services
- Onlinespiele

❑ Wie kann sie realisiert werden?

- Eindeutige Schnittstellen müssen definiert werden.
- Alle Beteiligten müssen wissen welche Daten zu erwarten sind.

❑ Mögliche Ansätze

- REST (für die Übung zu verwenden)
- Message Bus, Sockets, SOAP (zur Vollständigkeit vorhanden)



1.1 XML: Grundlagen

1.2 XML: Definition

1.3 XML: Schema

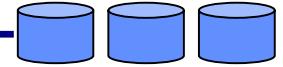
1.4 Netzwerkkommunikation: Einführung

1.5 Netzwerkkommunikation: REST

1.6 Netzwerkkommunikation: REST Implementierung

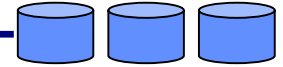
1.7 Netzwerkkommunikation: Message Bus, Sockets, SOAP

1.5 Netzwerkkommunikation: REST – Allgemein



- ❑ **REST (REpresentational State Transfer):** Kommunikation von Web Services über HTTP.
 - Verwendet Standard HTTP Methoden (GET, POST, PUT, DELETE).
 - Zustandslos
 - ◆ Jede Nachricht enthält alle notwendigen Informationen um verstanden zu werden.
 - ◆ Weder Client noch Server muss Zustände zwischen zwei Nachrichten speichern.
- ❑ **Vorteile**
 - REST Services können von jedem Tool das mit HTTP arbeiten kann aufgerufen werden. Dies umfasst nahezu jede aktuelle Programmiersprache.
 - Services können gut mit wechselnden Aufwand skalieren – Load Balancer können leicht eingebunden werden.
 - **Zustandslosigkeit:** Jeder Request ist unabhängig, folglich kein "Aufräumen" nach dem Disconnect eines Clients notwendig.
- ❑ **Nachteile**
 - **Zustandslosigkeit:** Problematisch wenn Zustände benötigt werden.
 - ◆ Beispielsweise: Anwender loggt sich einmal ein; diese Information muss in diesem Fall bei jeder Nachricht mitgeschickt werden.

1.5 Netzwerkkommunikation: REST – Zugriff über Ressourcen 1

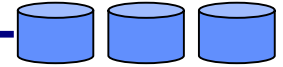


□ Zugriff über Ressourcen

- Beispiel: Firma, die eine Anzahl an Servicemitarbeitern hat, und Produkte, die sie verkaufen will.
 - ◆ Mitarbeiter und Kunden sind Ressourcen.
 - ◆ Alle Funktionalitäten die sich um diese Ressourcen drehen sollten ihnen zugeordnet sein.

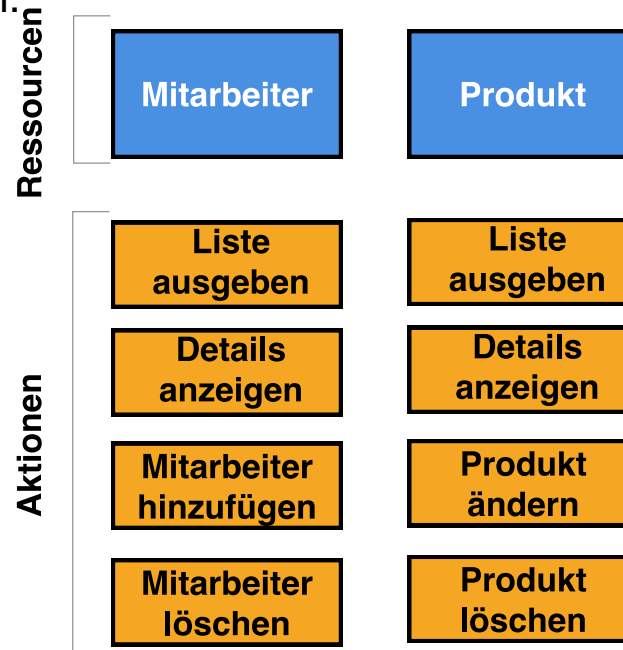


1.5 Netzwerkkommunikation: REST – Zugriff über Ressourcen 2



□ Zugriff über Ressourcen

- Beispiel: Firma, die eine Anzahl an Servicemitarbeitern hat, und Produkte, die sie verkaufen will.
 - ◆ Mitarbeiter und Kunden sind Ressourcen.
 - ◆ Alle Funktionalitäten die sich um diese Ressourcen drehen sollten ihnen zugeordnet sein.



1.5 Netzwerkkommunikation: REST – Zugriff über Ressourcen 3



□ Zugriff über Ressourcen – Mitarbeiter

- GET: <http://www.example.com/mitarbeiter>
 - ◆ Gibt eine Liste aller Mitarbeiter aus.
- POST: <http://www.example.com/mitarbeiter>
 - ◆ Fügt einen neuen Mitarbeiter hinzu.
- GET: <http://www.example.com/mitarbeiter/1>
 - ◆ Gibt Details über Mitarbeiter 1 aus (Name, Aufgabenbereich, ...).
- DELETE: <http://www.example.com/mitarbeiter/1>
 - ◆ Löscht Mitarbeiter 1.

Mitarbeiter

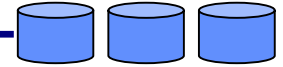
**Liste
ausgeben**

**Details
anzeigen**

**Mitarbeiter
hinzufügen**

**Mitarbeiter
löschen**

1.5 Netzwerkkommunikation: REST – Zugriff über Ressourcen 4



□ Zugriff über Ressourcen – Produkt

- GET: <http://www.example.com/products>
 - ◆ Gibt eine Liste aller Produkte aus.
- GET: <http://www.example.com/products/1>
 - ◆ Gibt Details über Produkt 1 aus.
- PUT: <http://www.example.com/products/1>
 - ◆ Ändert Details zu Produkt 1.
- DELETE: <http://www.example.com/products/1>
 - ◆ Löscht Produkt 1.

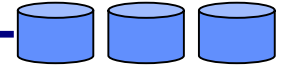
Produkt

**Liste
ausgeben**

**Details
anzeigen**

**Produkt
ändern**

**Produkt
löschen**



□ REST Services Best Practices

- Bezeichnungen der REST Interfaces sollten sprechend sein.
- Struktur der REST Interfaces sollte konsistent sein.
- Die HTTP Methoden sollten widerspiegeln, was in dem Service passiert.
- GET für Abfragen, POST fürs Erstellen, Delete für Löschen, PUT für Ändern.
- Rückgabe der Services sollte aussagekräftig sein.
 - ◆ Entsprechende Ausgaben / Erfolgsmeldungen
 - ◆ Entsprechende HTTP Error Codes wenn z.B. die übergebenen Parameter nicht passen
- SSL (HTTPS) sollte in echten Projekten immer eingesetzt werden.
- Query Parameters verwenden, um detailliertere Einschränkungen vorzunehmen (z.B. um Daten zu filtern).

1.5 Netzwerkkommunikation: REST – HATEOAS 1



- ❑ **HATEOAS (Hypermedia as the Engine of Application State):** Ist eine Möglichkeit zur Dokumentation einer REST Architektur.
 - Idee: Information um durch die Schnittstellen der REST Architektur zu navigieren wird mit den Antworten zurückgegeben.
 - Dokumentation ist damit teilweise direkt Teil der REST Architektur.

REST Antwort ohne HATEOAS

```
<clients>
  <client>
    <name>Alice</name>
  </client>
  <client>
    <name>Bob</name>
  </client>
</clients>
```

REST Antwort mit HATEOAS

```
<clients>
  <client>
    <name>Alice</name>
    <links>
      <rel>self</rel>
      <href>http://localhost:9090/customers/1</href>
    </links>
  </client>
  <client>
    <name>Bob</name>
    <links>
      <rel>self</rel>
      <href>http://localhost:9090/customers/2</href>
    </links>
  </client>
</clients>
```

1.5 Netzwerkkommunikation: REST – HATEOAS 2



- ❑ **HATEOAS (Hypermedia as the Engine of Application State):** Ist eine Möglichkeit zur Dokumentation einer REST Architektur.
 - Idee: Information um durch die Schnittstellen der REST Architektur zu navigieren wird mit den Antworten zurückgegeben.
 - Dokumentation ist damit teilweise direkt Teil der REST Architektur.

REST Antwort ohne HATEOAS

```
<clients>
  <client>
    <name>Alice</name>
  </client>
  <client>
    <name>Bob</name>
  </client>
</clients>
```

REST Antwort mit HATEOAS

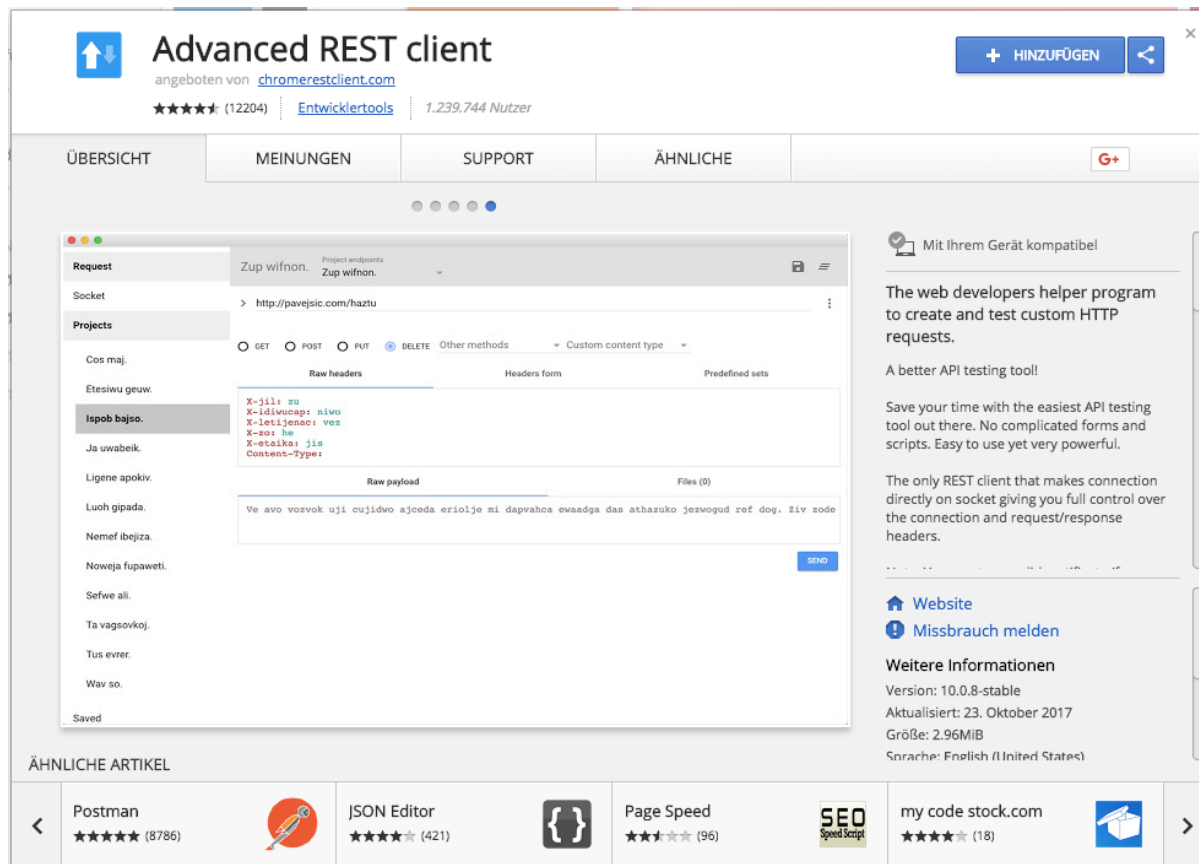
```
<clients>
  <client>
    <name>Alice</name>
    <links>
      <rel>self</rel>
      <href>http://localhost:9090/customers/1</href>
    </links>
  </client>
  <client>
    <name>Bob</name>
    <links>
      <rel>self</rel>
      <href>http://localhost:9090/customers/2</href>
    </links>
  </client>
</clients>
```

Link zu den REST
Services für die
einzelnen Kunden

1.5 Netzwerkkommunikation: REST – Services testen 1



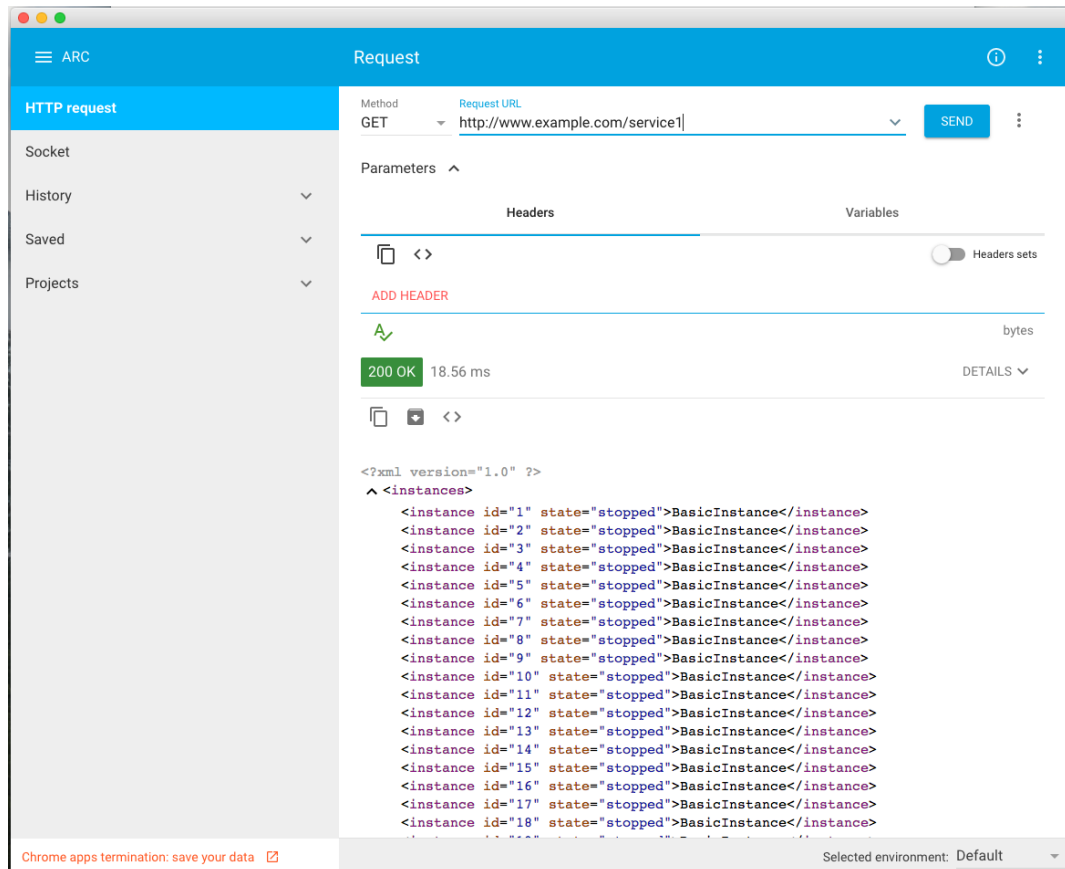
- ❑ **REST Services testen:** Beispielsweise über den **Advanced REST Client** (Google Chrome Plugin) oder Postman (auch Server Mocks).



1.5 Netzwerkkommunikation: REST – Services testen 2



- ❑ **REST Services testen:** Beispielsweise über den Advanced REST Client (Google Chrome Plugin) oder **Postman** (auch Server Mocks).





1.1 XML: Grundlagen

1.2 XML: Definition

1.3 XML: Schema

1.4 Netzwurkkommunikation: Einführung

1.5 Netzwurkkommunikation: REST

1.6 Netzwurkkommunikation: REST Implementierung

1.7 Netzwurkkommunikation: Message Bus, Sockets, SOAP



□ Einfaches Hello World Rest Service mit Java und Eclipse

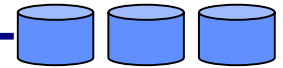
○ Erste Schritte

- ◆ Schritt 1: Eclipse installieren
- ◆ Schritt 2: Gradle für Eclipse installieren (entfällt in neueren Eclipse Versionen)
- ◆ Schritt 3: Neues Eclipse Projekt erstellen
- ◆ Schritt 4: Gradle Build File erstellen (oder aus dem bereitgestellten Basis-Projekt auf Moodle übernehmen)
- ◆ Schritt 5: Einfaches REST Service mit Spring Framework erstellen, das "Hello World" ausgibt

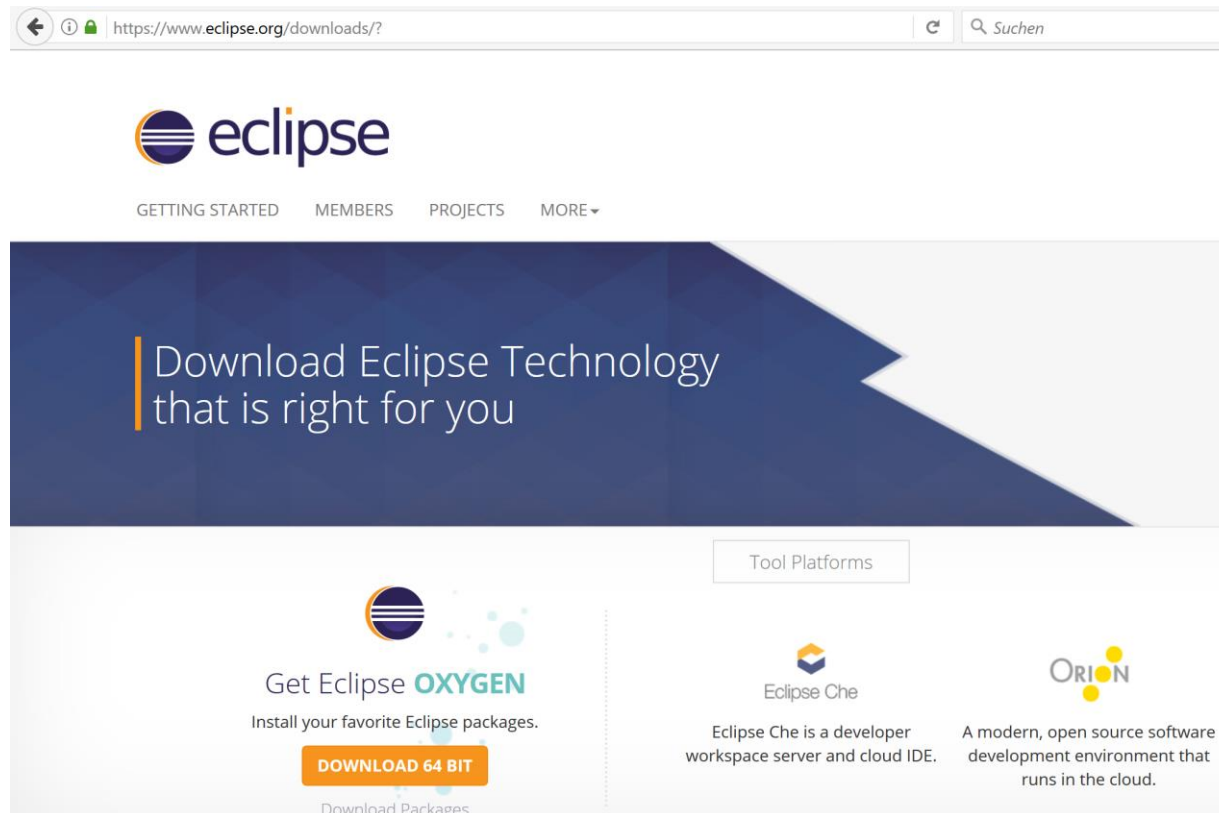
○ Gewünschtes Ergebnis:

- ◆ Aufruf <http://localhost:8080/greeting> im Web-Browser
- ◆ Rückgabe: {"id":1,"content":"Hello, World!"}

1.6 Netzwerkkommunikation: REST Implementierung – Schritt 1



- ❑ **Schritt 1:** Eclipse installieren – Version ist in der Angabe vorgegeben
 - <https://www.eclipse.org/downloads>
 - Installer ausführen

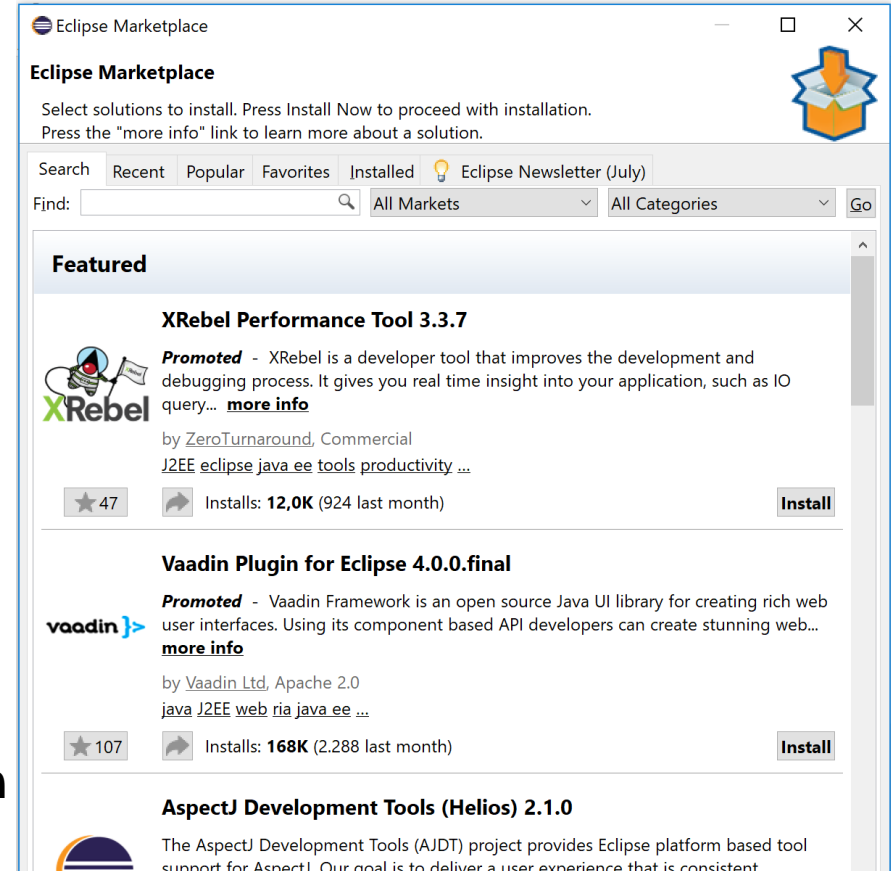


1.6 Netzwerkkommunikation: REST Implementierung – Schritt 2



❑ Schritt 2: Gradle für Eclipse installieren

- Gradle: Build Automation System
- Projektkonfigurationen können beschrieben und ausgeführt werden.
- Verwendet eine auf Groovy basierende Domain Specific Language (DSL).
- Für Eclipse: Buildship
 - ◆ Installation über den Eclipse Marketplace
 - ◆ Help → Eclipse Marketplace
- **In neueren Eclipse Versionen nicht mehr notwendig.**



1.6 Netzwerkkommunikation: REST Implementierung – Schritt 2



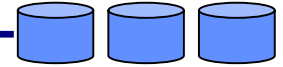
❑ Schritt 2: Gradle für Eclipse installieren – Eclipse Marketplace

1. Suche nach "Buildship"

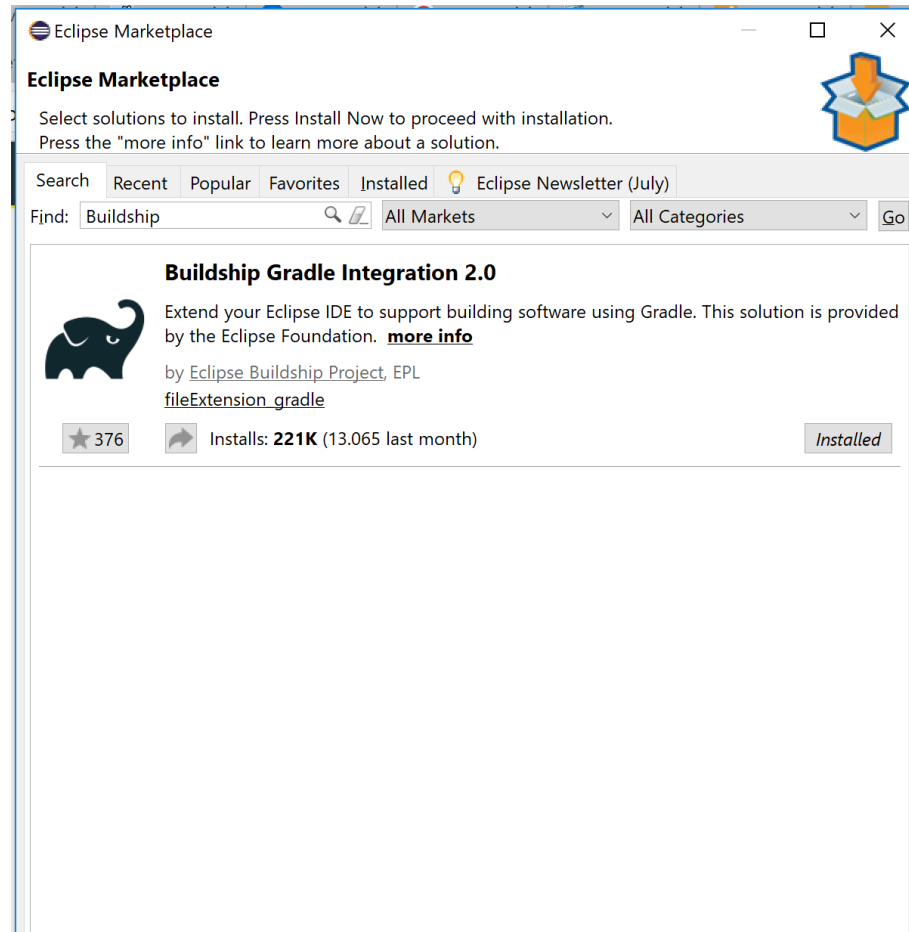
2. Install Button drücken

The screenshot shows the Eclipse Marketplace interface. At the top, there's a search bar with the text 'Buildship' entered. Below the search bar, the results for 'Buildship' are displayed. The first result is 'Buildship Gradle Integration 2.0', which is highlighted. To the right of this result, the 'Install' button is visible and highlighted with a red box. Red arrows from the text '1. Suche nach "Buildship"' and '2. Install Button drücken' point to the search bar and the 'Install' button respectively.

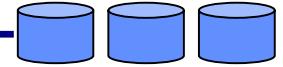
1.6 Netzwerkkommunikation: REST Implementierung – Schritt 2



❑ Schritt 2: Gradle für Eclipse installieren – Eclipse Marketplace



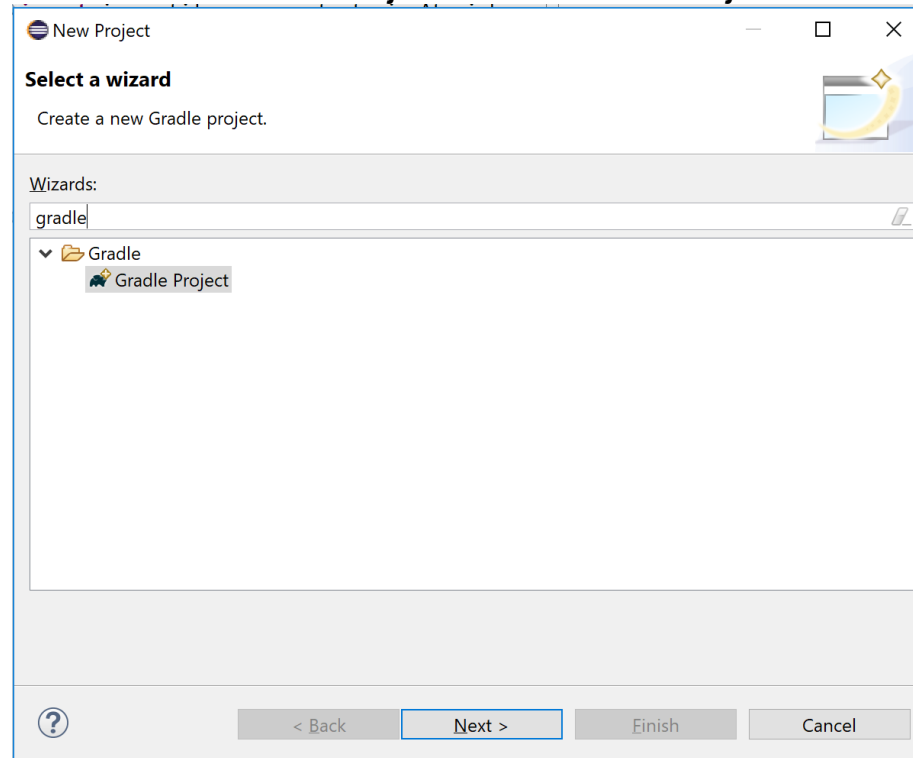
1.6 Netzwerkkommunikation: REST Implementierung – Schritt 3



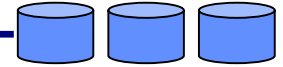
❑ Schritt 3: Neues Eclipse Projekt erstellen

- Nachdem wir Gradle für Eclipse installiert haben, können wir nun ein Gradle Projekte erstellen.

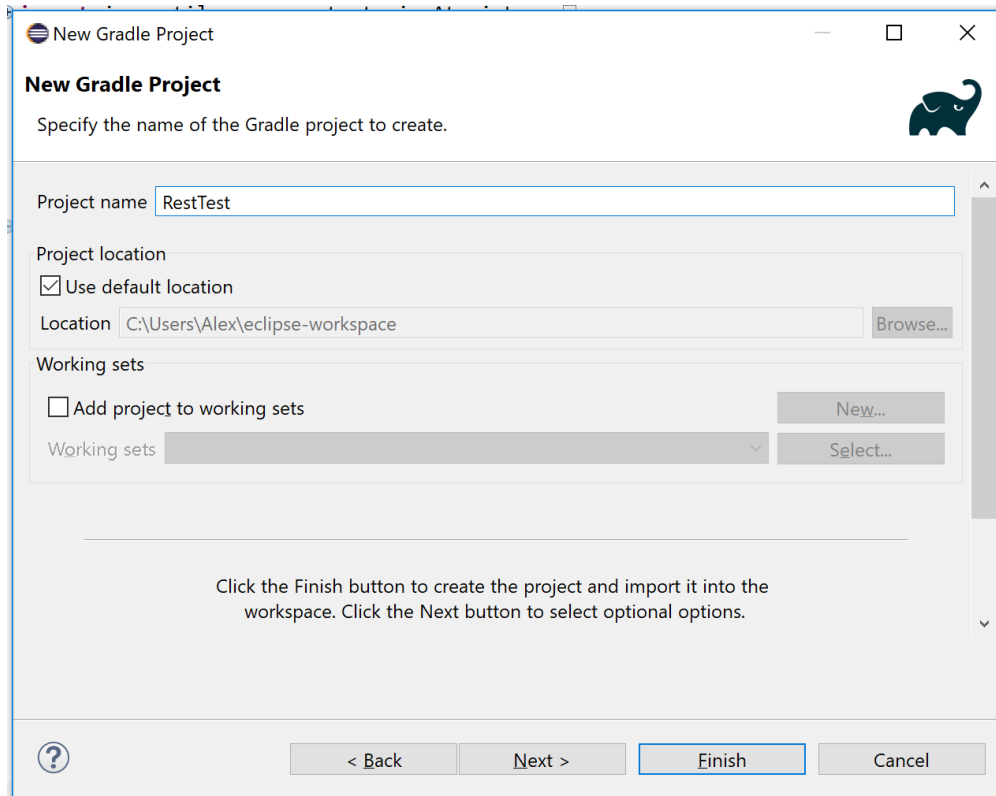
◆ Datei → Neu → Projekt → Gradle Projekt



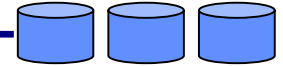
1.6 Netzwerkkommunikation: REST Implementierung – Schritt 3



- ❑ **Schritt 3: Neues Eclipse Projekt erstellen**
 - Projektname eingeben: RestTest
 - Restliche Einstellungen können beibehalten werden

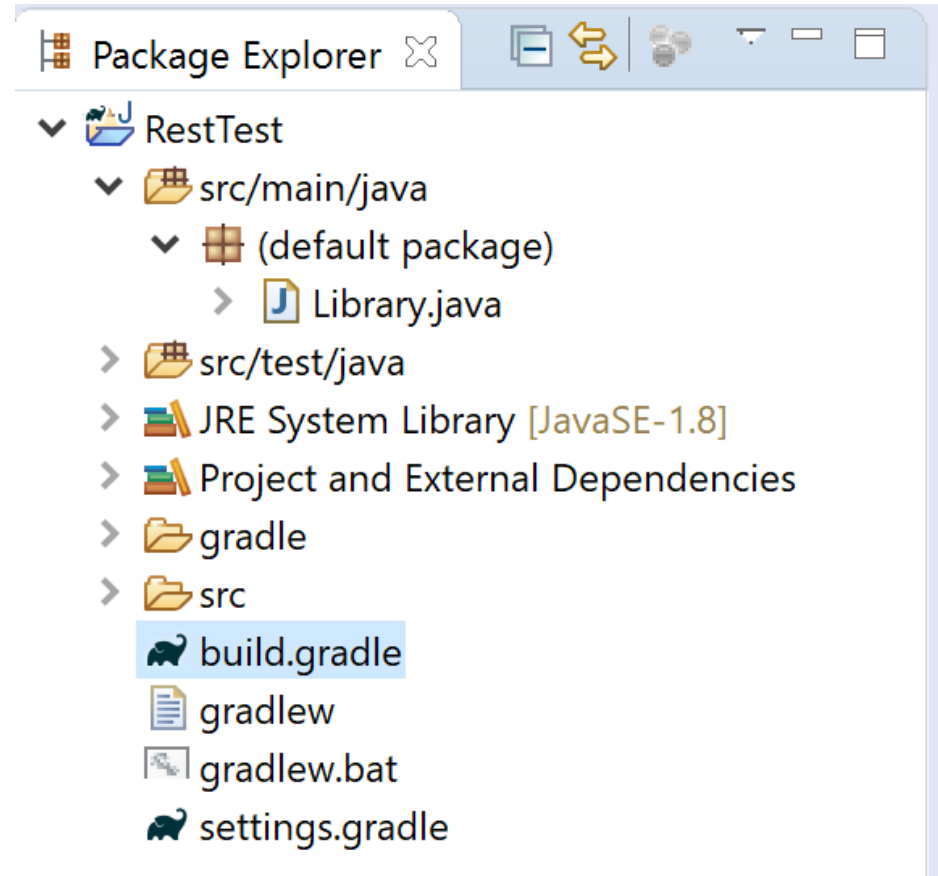


1.6 Netzwerkkommunikation: REST Implementierung – Schritt 3



❑ Schritt 3: Neues Eclipse Projekt erstellen

- Im Package Explorer sehen wir nun das neu erstellte Projekt
- Im default package wurde bereits eine Datei `Library.java` erstellt – diese kann gelöscht werden
- In `build.gradle` werden wir unsere Libraries eintragen



1.6 Netzwerkkommunikation: REST Implementierung – Schritt 4



❑ Schritt 4: Gradle Build File erstellen

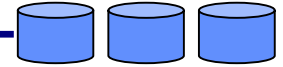
```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:2.1.6.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'
repositories {
    mavenCentral()
}

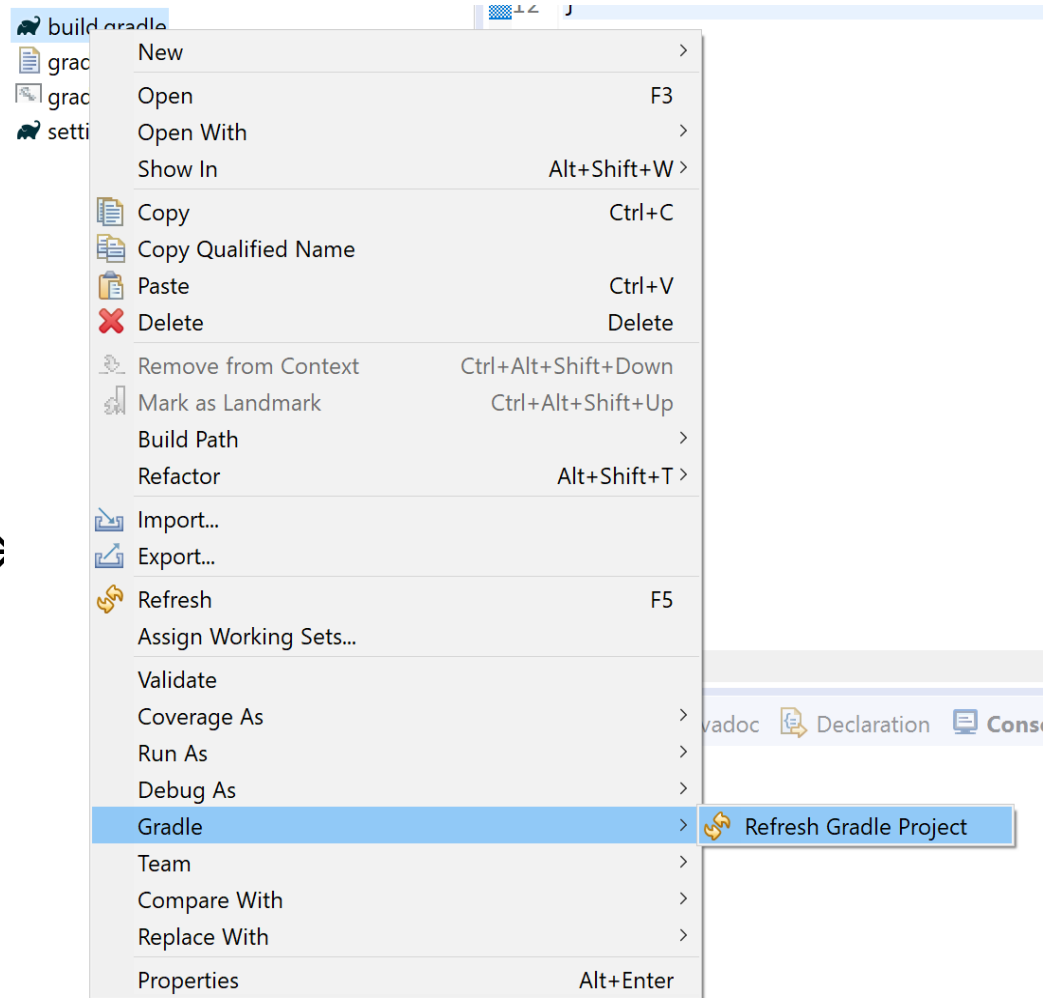
sourceCompatibility = 1.11
targetCompatibility = 1.11
dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:2.1.6.RELEASE")
    compile("org.springframework.boot:spring-boot-starter-webflux:2.1.6.RELEASE")
    compile("com.sun.xml.bind:jaxb-impl:2.3.1")
    compile("javax.xml.bind:jaxb-api:2.3.1")
}
```

Nützen Sie die Beispielprojekte auf Moodle, diese beinhalten eine vollständige & passende Gradle Konfiguration für die Übung.

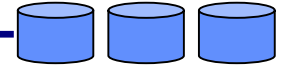
1.6 Netzwerkkommunikation: REST Implementierung – Schritt 4



- ❑ **Schritt 4: Gradle Build File erstellen**
 - Das Gradle Build File muss erst verarbeitet werden
 - Lädt benötigte Libraries in das Projekt
 - Rechtsklick auf build.gradle → Gradle → Refresh Gradle Project

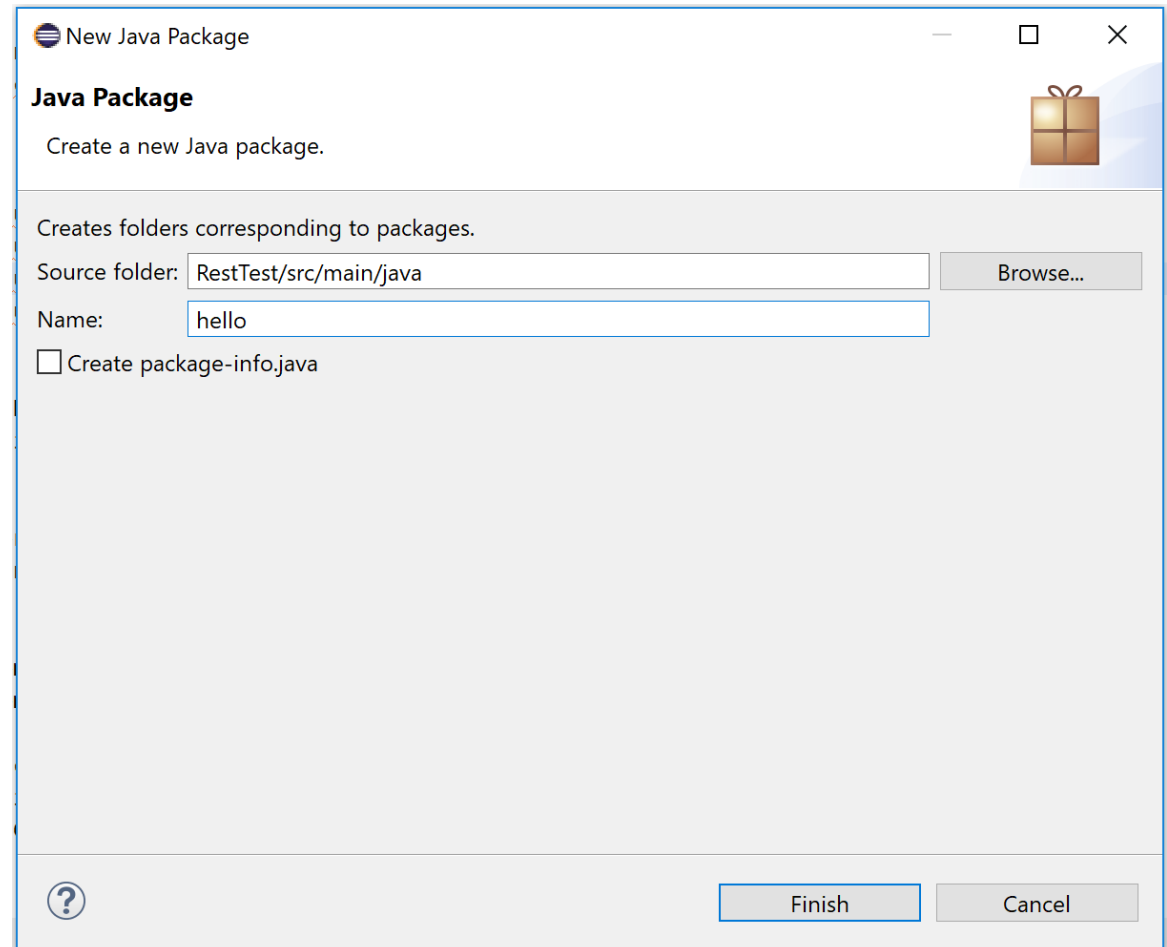


1.6 Netzwerkkommunikation: REST Implementierung – Schritt 5

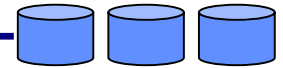


❑ Schritt 5: Quellcode erstellen

- Neues Paket erstellen: `hello`
- Rechtsklick auf Projekt → Neu → Paket



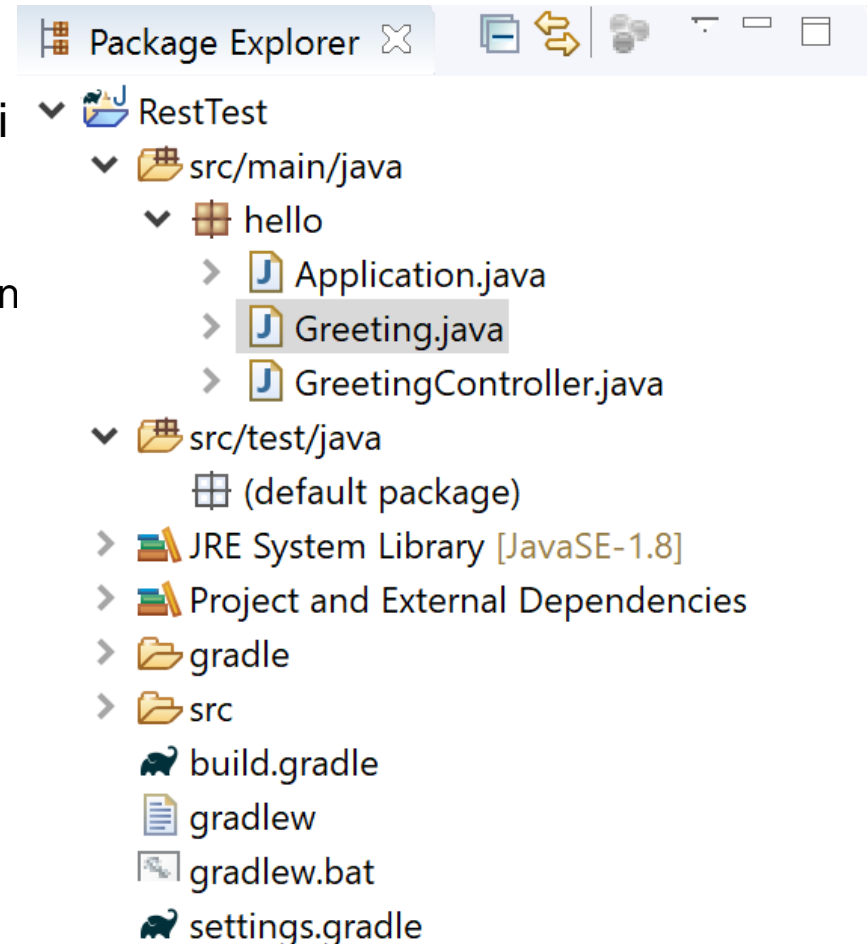
1.6 Netzwerkkommunikation: REST Implementierung – Schritt 5



❑ Schritt 5: Quellcode erstellen

- In diesem Paket erstellen wir drei Dateien

- ◆ `Greeting.java`: Enthält die Daten die wir am Ende ausgeben wollen
- ◆ `GreetingController.java`: Beschreibt das Webservice, welches `Greeting.java` aufruft
- ◆ `Application.java`: Lässt unser Projekt als JAR File ausführbar machen



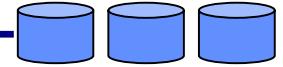
1.6 Netzwerkkommunikation: REST Implementierung – Schritt 5



❑ Schritt 5: Quellcode erstellen – Greeting.java

```
public class Greeting {  
  
    private final long id;  
    private final String content;  
  
    public Greeting(long id, String content) {  
        this.id = id;  
        this.content = content;  
    }  
  
    public long getId() {  
        return id;  
    }  
  
    public String getContent() {  
        return "ID: " + id + "; Greeting: " + content;  
    }  
}
```


1.6 Netzerkcommunication: REST Implementierung – Schritt 5



❑ Schritt 5: Quellcode erstellen – GreetingController.java

```
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {
    private static final String template = "Hello, %s!";
    private final AtomicLong counter = new AtomicLong();

    @RequestMapping("/greeting", produces = MediaType.APPLICATION_XML_VALUE)
    public Greeting greeting(@RequestParam(value="name", defaultValue="World")
        String name) {

        return new Greeting(counter.incrementAndGet(), String.format(template, name));
    }
}
```

1.6 Netzwerkkommunikation: REST Implementierung – Schritt 5

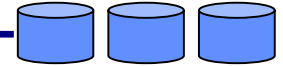


❑ Schritt 5: Quellcode erstellen – Application.java

```
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

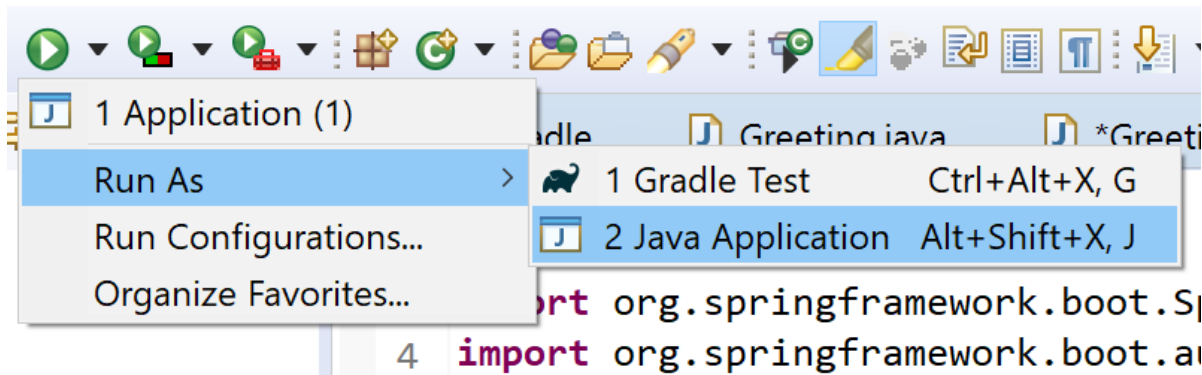
```
@SpringBootApplication  
public class Application {  
  
    public static void main(String[] args) {  
        SpringApplication.run(Application.class, args);  
    }  
}
```

1.6 Netzwerkkommunikation: REST Implementierung – Schritt 6



❑ Schritt 6: Applikation ausführen

- Klick auf den grünen Pfeil



- Gehe zu `http://localhost:8080/greeting`
 - ◆ Rückgabe: ID: 1; Greeting: Hello, World!
- Gehe zu `http://localhost:8080/greeting?name=Student`
 - ◆ Rückgabe: ID: 2; Greeting: Hello, World!
- Die `id` wird bei jedem Besuch inkrementiert.

1.6 Netzwerkkommunikation: REST Implementierung – Client

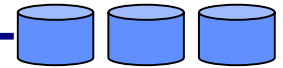


❑ Erstellen des REST Clients mit dem Spring Framework

- **Ziel:** REST Services abfragen und Infos verarbeiten
- Gradle Projekt erstellen
- `build.gradle` wie hier beschrieben editieren
- Nützen Sie die Beispielprojekte auf Moodle, diese beinhalten eine vollständige und passende Gradle Konfiguration für die Übung.

```
buildscript {  
    repositories {  
        mavenCentral()  
    }  
    dependencies {  
        classpath("org.springframework.boot:spring-boot-gradle-plugin:2.1.6.RELEASE")  
    }  
}  
  
apply plugin: 'java'  
apply plugin: 'eclipse'  
apply plugin: 'org.springframework.boot'  
  
repositories {  
    mavenCentral()  
}  
  
sourceCompatibility = 1.11  
targetCompatibility = 1.11  
  
dependencies {  
    compile("org.springframework.boot:spring-boot-starter-web:2.1.6.RELEASE")  
    compile("org.springframework.boot:spring-boot-starter-webflux:2.1.6.RELEASE")  
    compile("com.sun.xml.bind:jaxb-impl:2.3.1")  
    compile("javax.xml.bind:jaxb-api:2.3.1")  
}
```

1.6 Netzwerkkommunikation: REST Implementierung – Main Klasse



❑ Zugriff auf das Webservice mit WebClient (empfohlen)

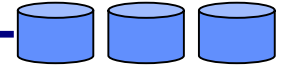
- Der String `received` kann nun z.B. mit JAXB (wenn es sich um ein XML handelt) weiterverarbeitet werden

```
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.MediaType;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

public class ExampleRestClient {
    private static WebClient baseWebClient = WebClient.builder()
        .baseUrl("http://localhost:8080")
        .defaultHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_XML_VALUE)
        .defaultHeader(HttpHeaders.ACCEPT, MediaType.APPLICATION_XML_VALUE)
        .build();

    public static void main(String[] args) {
        Mono<String> webAccess = baseWebClient.method(HttpMethod.POST)
            .uri("/greeting")
            .retrieve()
            .bodyToMono(String.class);

        String received = webAccess.block();
        System.out.println(received);
    }
}
```



1.1 XML: Grundlagen

1.2 XML: Definition

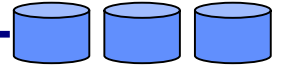
1.3 XML: Schema

1.4 Netzwerkkommunikation: Einführung

1.5 Netzwerkkommunikation: REST

1.6 Netzwerkkommunikation: REST Implementierung

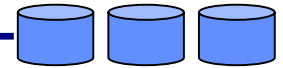
1.7 Netzwerkkommunikation: Message Bus, Sockets, SOAP



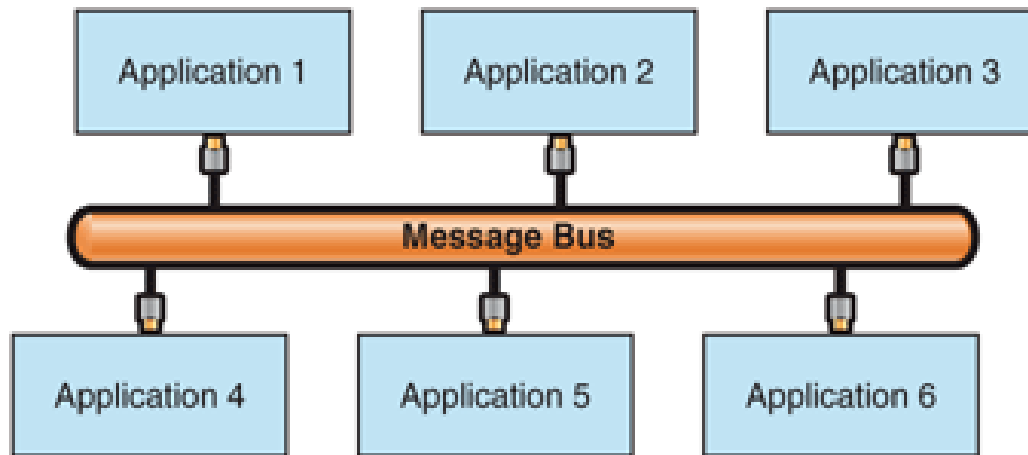
Netzwerkkommunikation

MESSAGE BUS

1.7 Netzwerkkommunikation: Message Bus 1



- ❑ **Message Bus:** Applikationen werden über den *Message Bus*, einer logischen Komponente, miteinander verbunden.
 - Ziel: Applikationen von verschiedenen Herstellern miteinander verbinden.
 - Besteht aus drei Hauptkomponenten.
 - ◆ Nachrichtenschemas
 - ◆ Gemeinsame Command Messages
 - ◆ Geteilte Infrastruktur um Nachrichten zu verschicken und empfangen

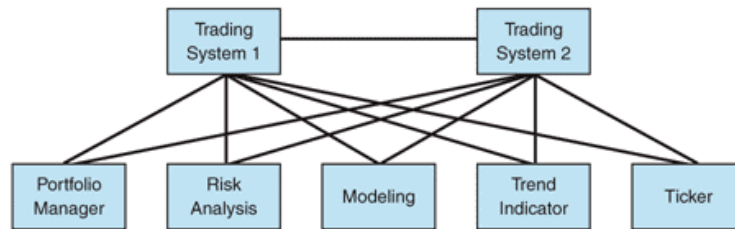


Quelle: <https://msdn.microsoft.com/en-us/library/ff647328.aspx>

1.7 Netzwerkkommunikation: Message Bus 2



Ohne Message Bus

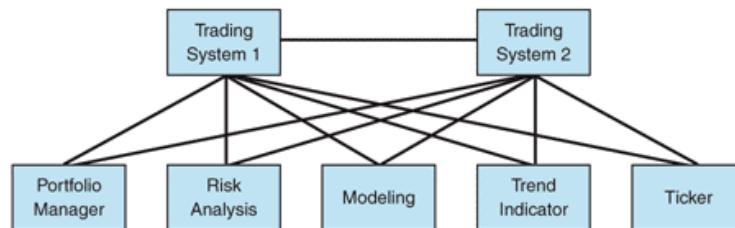


Quelle: <https://msdn.microsoft.com/en-us/library/ff647328.aspx>

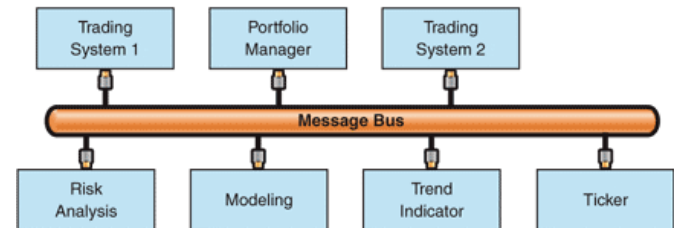
1.7 Netzwerkkommunikation: Message Bus 3



Ohne Message Bus

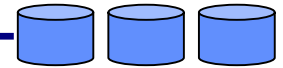


Mit Message Bus

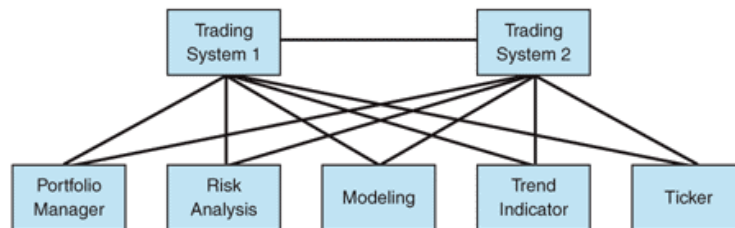


Quelle: <https://msdn.microsoft.com/en-us/library/ff647328.aspx>

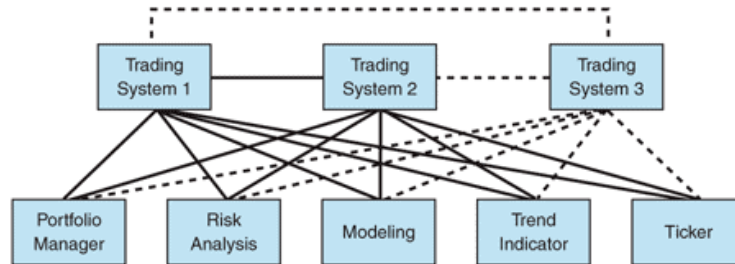
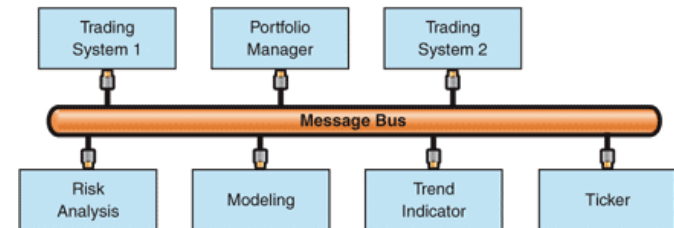
1.7 Netzwerkkommunikation: Message Bus 4



Ohne Message Bus



Mit Message Bus

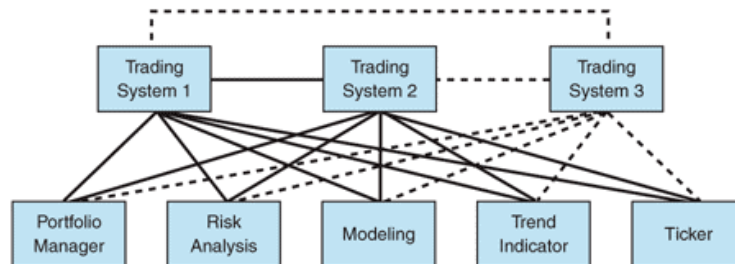
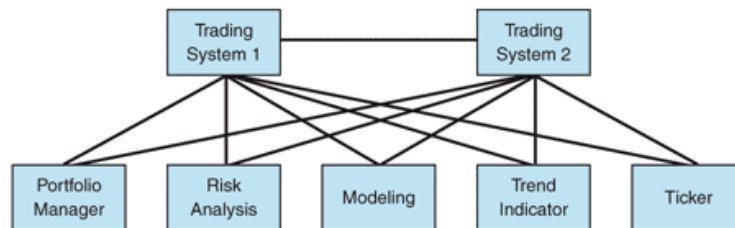


Quelle: <https://msdn.microsoft.com/en-us/library/ff647328.aspx>

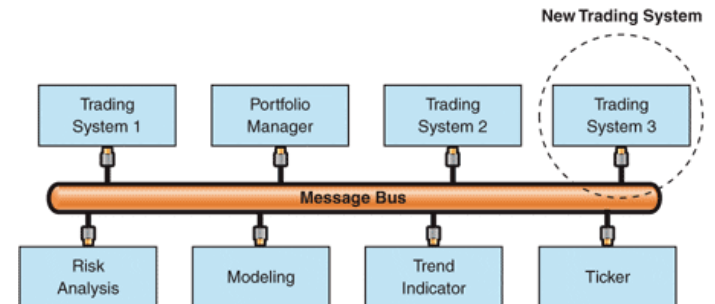
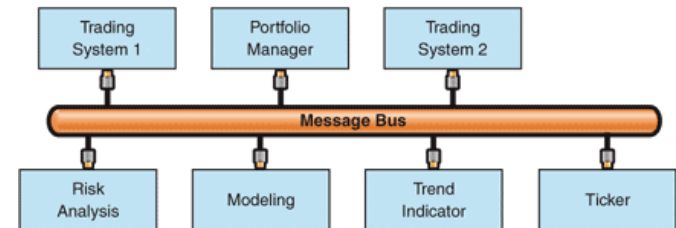
1.7 Netzwerkkommunikation: Message Bus 5



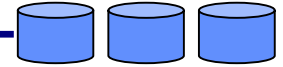
Ohne Message Bus



Mit Message Bus



Quelle: <https://msdn.microsoft.com/en-us/library/ff647328.aspx>



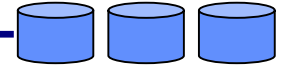
❑ Message Bus, Vorteile

- Alle Komponenten müssen nur mehr mit dem Message Bus verbunden sein um miteinander Nachrichten austauschen zu können.
- Keine direkten Verbindungen zu anderen Komponenten notwendig.

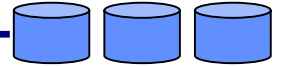
❑ Message Bus, Nachteile

- Eigene Infrastruktur wird benötigt.
- Bus Interface kann nur schwierig geändert werden.
- Geringere Sicherheit – Broadcast Messages erreichen alle Teilnehmer.
- Geringe Toleranz gegenüber der Nichtverfügbarkeit von Empfängern.

1.7 Netzwerkkommunikation: Message Bus 7



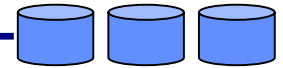
- ❑ **Message Bus:** Geteilte Infrastruktur über Message Router oder Publish / Subscribe Mechanismen.
- ❑ **Publish / Subscribe**
 - Ein Teilnehmer veröffentlicht eine Nachricht.
 - Diese Nachricht wird an alle Teilnehmer geschickt die sich zuvor registriert haben.
 - Möglichkeiten:
 - ◆ List-based Publish / Subscribe
 - ◆ Broadcast-based Publish / Subscribe
 - ◆ Content-based Publish / Subscribe



Netzwerkcommunication

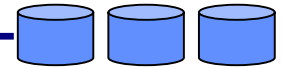
SOCKETS

1.7 Netzwerkkommunikation: Sockets 1



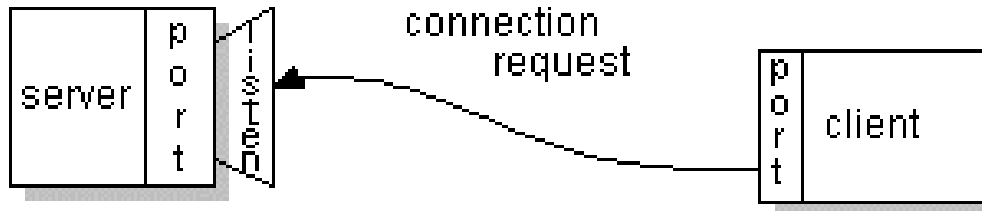
- ❑ **Sockets:** Endpunkte für Kommunikation zwischen zwei Teilnehmern wie beispielsweise Client und Server.
 - Arbeitet mit TCP oder UDP.
 - Immer an eine Portnummer gebunden.
 - Zwei Klassen in `java.net`
 - ◆ `Socket` (Client Seite)
 - ◆ `ServerSocket` (Server Seite)
 - Sockets sind geeignet für Client-Server Applikationen aber nicht optimal für die Anzeige von Webseiten und ähnlichen Inhalten – dafür gibt es besser geeignete (spezialisierte) Klassen:
 - ◆ `URLConnection`
 - ◆ `URLEncoder`

1.7 Netzwerkkommunikation: Sockets 2



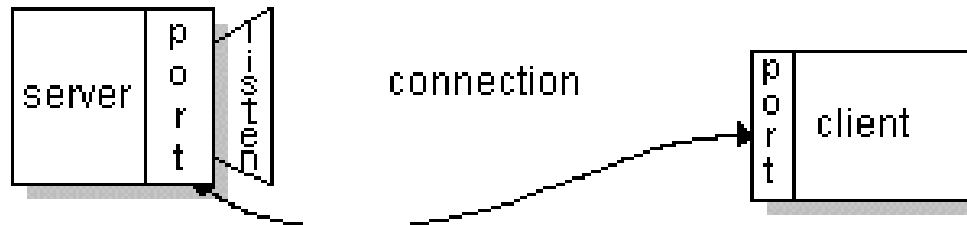
❑ Sockets: Verbindungsaufbau

- Client schickt eine Verbindungsanfrage an den Server.
 - ◆ Client kennt Hostname des Servers + dessen Port



Quelle: <https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html>

- Server akzeptiert im Idealfall die Verbindung.
 - ◆ Merkt sich Adresse + Port des Clients um mit diesem weiter kommunizieren zu können.
 - ◆ Bereit für weitere Clients



Quelle: <https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html>

1.7 Netzwerkkommunikation: Sockets 3



❑ Sockets: Server Beispiel

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class GreetingServer {

    public static void main(String[] args) throws IOException {
        ServerSocket listener = new ServerSocket(9091);
        try {
            while (true) {
                Socket socket = listener.accept();
                try {
                    PrintWriter out =
                        new PrintWriter(socket.getOutputStream(), true);
                    out.println("Hello student");
                } finally {
                    socket.close();
                }
            }
        } finally {
            listener.close();
        }
    }
}
```

1.7 Netzerkcommunication: Sockets 4



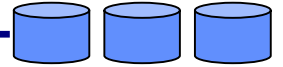
□ Sockets: Client Beispiel

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;

import javax.swing.JOptionPane;

public class GreetingClient {

    public static void main(String[] args) throws IOException {
        Socket s = new Socket("127.0.0.1", 9091);
        BufferedReader input =
            new BufferedReader(new InputStreamReader(s.getInputStream()));
        String answer = input.readLine();
        JOptionPane.showMessageDialog(null, answer);
        System.exit(0);
    }
}
```



Netzwerkkommunikation

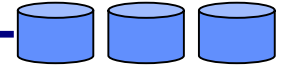
SOAP

1.7 Netzwerkkommunikation: SOAP 1



- ❑ **SOAP (Simple Object Access Protocol):** Definiert ein Format für Netzwerkkommunikation.
 - Plattformunabhängig
 - Basiert auf XML
- ❑ **SOAP Bausteine**
 - **Envelope:** Definiert, dass es eine SOAP Nachricht ist
 - **Header:** Allgemeine Infos über die Nachricht
 - **Body:** Information des Aufrufs und der Antwortmöglichkeit
 - **Fault:** Information über Fehler und Status
- ❑ **SOAP Vorteile**
 - Plattformunabhängig
 - Entkoppelt Kommunikationsprotokoll von verwendeter Sprache / Umgebung.
- ❑ **SOAP Nachteile**
 - Langsamer als andere Ansätze.
 - Recht großer Overhead in Spezifikation der Endpunkte.

1.7 Netzwerkkommunikation: SOAP 2



□ SOAP: Beispiel

```
<?xml version="1.0"?>

<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

  <soap:Header>
    ...
  </soap:Header>

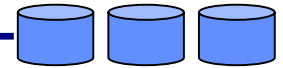
  <soap:Body>
    ...
    <soap:Fault>
      ...
    </soap:Fault>
  </soap:Body>

</soap:Envelope>
```



- ❑ **SOAP WSDL (Web Service Description Language):** Ist die Beschreibung von Webservices.
 - Welche Funktionen sind verfügbar.
 - Welche Daten werden angeboten.
 - Basiert auf XML
 - Mehrere Hauptelemente
 - ◆ `types`: Definiert die Datentypen
 - ◆ `message`: Definiert, wie die Daten übertragen werden
 - ◆ `portType`: Definiert die durchführbaren Operationen
 - ◆ `binding`: Definiert Protokoll und Datenformat für jeden `portType`

1.7 Netzwerkkommunikation: SOAP WSDL – Struktur



□ SOAP WSDL: Struktur

```
<definitions>

  <types>
    Welche Datentypen gibt es?
  </types>

  <message>
    Wie werden die Daten übertragen?
  </message>

  <portType>
    Welche Operationen stehen zur Verfügung?
  </portType>

  <binding>
    Welche Protokolle und Datenformate stehen zur Verfügung?
  </binding>

</definitions>
```


1.7 Netzwerkkommunikation: SOAP – WSDL portType 1



□ SOAP WSDL: portType Beispiel

```
<definitions>

...

<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>

...

</definitions>
```

1.7 Netzwerkkommunikation: SOAP – WSDL portType 2



□ SOAP WSDL: portType Beispiel

```
<definitions>
```

```
...
```

```
<message name="getTermRequest">  
  <part name="term" type="xs:string"/>  
</message>
```

```
<message name="getTermResponse">  
  <part name="value" type="xs:string"/>  
</message>
```

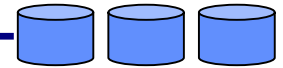
```
<portType name="glossaryTerms">  
  <operation name="getTerm">  
    <input message="getTermRequest"/>  
    <output message="getTermResponse"/>  
  </operation>  
</portType>
```

```
...
```

```
</definitions>
```

mögliche
Nachrichten

1.7 Netzwerkkommunikation: SOAP – WSDL portType 3



□ SOAP WSDL: portType Beispiel

```
<definitions>

...

<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>

...

</definitions>
```

werden in den
Operationen
verwendet

1.7 Netzwerkkommunikation: SOAP – WSDL portType 4



□ SOAP WSDL: portType Beispiel

```
<definitions>

...

<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

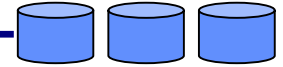
<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>

...

</definitions>
```

werden in den
Operationen
verwendet

1.7 Netzwerkkommunikation: SOAP – WSDL Binding 1



□ SOAP WSDL: Binding zu einem SOAP Service

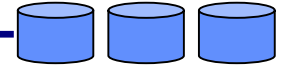
```
<definitions>
...

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>

<binding type="glossaryTerms" name="b1">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <operation>
    <soap:operation soapAction="http://example.com/getTerm"/>
    <input><soap:body use="literal"/></input>
    <output><soap:body use="literal"/></output>
  </operation>
</binding>

...
</definitions>
```

1.7 Netzwerkkommunikation: SOAP – WSDL Binding 2



□ SOAP WSDL: Binding zu einem SOAP Service

```
<definitions>
...

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest" />
    <output message="getTermResponse" />
  </operation>
</portType>

<binding type="glossaryTerms" name="b1">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <operation>
    <soap:operation soapAction="http://example.com/getTerm"/>
    <input><soap:body use="literal"/></input>
    <output><soap:body use="literal"/></output>
  </operation>
</binding>

...
</definitions>
```

Diagram illustrating the binding of a SOAP service to a WSDL definition. A blue arrow points from the **glossaryTerms** portType definition to the **glossaryTerms** binding type, indicating the reference.

Name des referenzierten portTypes

1.7 Netzwerkkommunikation: SOAP – WSDL Binding 3



□ SOAP WSDL: Binding zu einem SOAP Service

```
<definitions>
...

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>

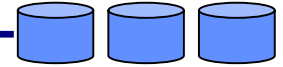
<binding type="glossaryTerms" name="b1">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <operation>
    <soap:operation soapAction="http://example.com/getTerm"/>
    <input><soap:body use="literal"/></input>
    <output><soap:body use="literal"/></output>
  </operation>
</binding>

...
</definitions>
```

URL des
SOAP Services



1.7 Netzwerkkommunikation: SOAP – WSDL Binding 4



□ SOAP WSDL: Binding zu einem SOAP Service

```
<definitions>
...

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>

<binding type="glossaryTerms" name="b1">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <operation>
    <soap:operation soapAction="http://example.com/getTerm"/>
    <input><soap:body use="literal"/></input>
    <output><soap:body use="literal"/></output>
  </operation>
</binding>

...
</definitions>
```

**für jede
Operation muss eine
SOAP Aktion
spezifiziert werden**