

# REST(ful) Webservices im Social Web



Matthias Steinbauer  
matthias@steinbauer.org  
@climbinggeek

## Matthias Steinbauer

Studium Informatik JKU, derzeit PhD am  
Institut für Telekooperation  
(Big Data, Temporal Graphs)

Seit 2012 Universitätsassistent @JKU  
- 8 angeleitete Bachelor- und Masterarbeiten  
- 15 wissenschaftliche Publikationen

Seit 2004 Selbstständig  
- Softwareentwicklung Java EE Umfeld  
- Trainings Prozessmodellierung  
- Beratung Big Data, Prozesse, Java EE



matthias@steinbauer.org  
http://steinbauer.org/

## Hinweise zum Slide-Deck

Zusammenfassung

Beispiel Übung

Agenda/Pause

## Agenda

- Einführung zu Webservices
- Grundlagen zur Service Seite mit Java (JAX-RS, Jersey)
- Web basierter Client mit AngularJS
- Methoden Aufrufe (Actions) vs. CRUD
- Fortgeschrittenes und Reflexion

# Einführung zu Webservices

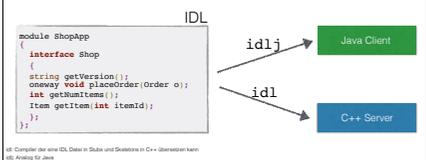
- Einsatzgebiete
- Arten von Webservices
- Stateful vs. Stateless Calls
- Grundlagen im HTTP Protokoll
- Basics zu RESTful Services

# Netzwerk Modell

- ↑ **Web Services (SOAP, XML-RPC, JSON-RPC, REST, ...)**
  - Basieren auf etablierten Transport-Protokollen (Binär, HTTP, etc.)
  - Procedure Call ist standardisiert, Daten-Kodierung ist normiert
  - Häufig Beschreibung des Service erforderlich
- Remote Procedure Calls (Corba, RPC, ActiveX, RMI, ...)**
  - Protokoll-Details vom Programmierer versteckt
  - Schränkt Nutzung häufig auf eine bestimmte Technologie ein
  - Client und Server sind eng miteinander verbunden
  - Clients werden häufig "generiert"
- Standardisierte (Text) Protokolle (FTP, HTTP, Telnet, ...)**
  - Protokoll durch mehrere Hersteller (Client und Server) implementiert
  - Häufig zu ganz speziellen Zwecken entworfen
- Binäre Protokolle (TCP, RTP, IP, ...)**
  - Schwer durch Menschen zu interpretieren
  - Bei proprietären Protokollen kompliziert Clients zu bauen

# CORBA

- Aus einer Beschreibung des Kommunikations-Interface werden Stubs für die Clients und Skeletons für die Server generiert
- Clients können mit dem Server über ein binäres Protokoll kommunizieren, Programmierer hat grundsätzlich keinen weiteren Einfluss auf dieses Protokoll



# SOAP

## Simple Object Access Protocol

- Eine WSDL Datei beschreibt per XML Nachrichten die zwischen Client und Server getauscht werden können
- Die Nachrichten selbst sind auch XML und können von Menschen verstanden werden
- Client und Server können generiert werden
- Viel häufiger wird aber die WSDL von einer Server Implementierung generiert und davon werden Client Calls generiert



## Vergleich

	RPC	SOAP	REST
Datentransport	Binär / undefiniert	HTTP Andere Transports	???
Dienstbeschreibung	IDL	WSDL	???
Session Status	Stateful	optional Stateful	???
Kopplung	Sehr eng oft Technologie	Eng über WSDL / XML	???

## Was ist REST

- **RE**presentational **St**ate **T**ransfer
- REST Services sind eine **benutzerfreundliche** Möglichkeit **verteilte Systeme** zu bauen. Die Struktur von REST Services ist **einfach zu durchschauen** und **leicht skalier**
- Ist vielmehr eine **Konvention** als ein **Standard**

## Benutzerfreundlich / Einfach

- Es finden Begriffe die wir aus dem Web kennen Anwendung:
  - URL / URI
- Basiert auf HTTP und kann daher grundsätzlich mit Web-Browsern aufgerufen werden (Web == REST)
- Lose über das HTTP Protokoll gekoppelt > keine komplizierten Abhängigkeiten zwischen den verteilten Komponenten

## Skalierbar

- REST per Definition und Vereinbarung Stateless
- Jeder Client Request kann daher von einem beliebigen Server in einem Serververbund beantwortet werden
- Horizontale Skalierung unter Zuhilfenahme von Round-Robin-DNS oder Load-Balancern möglich
- Backend-Services müssen natürlich dafür ausgelegt sein

# Vergleich

	RPC	SOAP	REST
Datentransport	Binär / undefiniert	HTTP Andere Transports	HTTP
Dienstbeschreibung	IDL	WSDL	Keine URLs
Session Status	Stateful	optional Stateful	Stateless
Kopplung	Sehr eng oft Technologie	Eng über WSDL / XML	Lose wie Web
Anwendung	Innerhalb Softwarepaket	Verschiedene Systeme eines Unternehmens	Über Unternehmensgrenzen hinweg

# Konzepte

- Resource (Objekte und deren Eigenschaften)
- Representation (Darstellung einer Resource)
- Operation (z.B. Anlegen oder Löschen)
- Hypertext (Verfolgen von Links)
- Statelessness (Application State am Client)

# Resource

- Jedes **Objekt** ist eine **Resource** und hat daher einen Uniform Resource Identifier (URI)
  - `/shop/rest/items/7`, `/shop/rest/items`
- Mit einem Uniform Resource Locator (URL) kann beschrieben werden wie man auf eine Resource zugreifen kann
  - `https://myshop.at/shop/rest/items`
  - `http://www.land-oberoesterreich.gv.at/default.htm`
- Listen und Elemente sollen in Zusammenhang stehen
- Resources können statisch sein oder sich verändern

# REST und URLs

- URLs sollen beschreiben **wie**, wir **wo** auf **was** zugreifen können
  - `https://myshop.at/shop/rest/items`
  - `https://myshop.at/shop/rest/items/7`
- URL Parameter sind **Parameter** und identifizieren keine Resource
- URLs sollen keine technischen Details enthalten
  - `https://myshop.at/index.php?retrieveItem=7&aa=json`
  - `https://myshop.at/my-net-share/cgi-bin/rest.cgi?function=retrieveItem&itemId=7`

## Representation

- Unter einem einzigen URL kann der gleiche Inhalt in verschiedener Ausprägung abgerufen werden
- <https://myshop.at/shop/rest/items/7> könnte HTML, XML, JSON, Text, JPEG, etc. liefern
- Im HTTP Protokoll kann über den Header: **Accept** spezifiziert werden welches Encoding der Client erwartet
- Web-Browser werden typischerweise **Accept: text/html** spezifizieren
- Ein Web-Service Client wird aber vermutlich explizit JSON, XML oder ein anderes strukturiertes Format anfordern wollen

## Representation

- Häufig sieht man die Verwendung verschiedener URLs zur Abfrage der gleichen Resource in verschiedenem Encoding
  - <http://steinbauer.org/reports/2015/Q4/sales.html>
  - <http://steinbauer.org/reports/2015/Q4/sales.xml>
  - <http://steinbauer.org/reports/2015/Q4/sales.json>
- In REST sollten aber alle Representations unter einem URL verfügbar sein
  - <http://steinbauer.org/reports/2015/Q4/sales>

<http://steinbauer.org/reports/2015/Q4/sales>

```
Accept: application/xml
<salesReport>
  <category name="Non-Food">
    287362.83
  </category>
  <category name="Food">
    827736.3
  </category>
  <category name="Services">
    983726.98
  </category>
</salesReport>

Accept: application/json
{
  "Non-Food": 287362.83,
  "Food": 827736.3,
  "Services": 983726.98
}

Accept: text/html
<html>
  <head>
    <title>Q4 Sales Report</title>
  </head>
  <body>
    <table>
      <thead>
        <tr>
          <th>Category</th>
          <th>EUR</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Non-Food</td>
          <td>287362.83</td>
        </tr>
        <tr>
          <td>Food</td>
          <td>827736.3</td>
        </tr>
        <tr>
          <td>Services</td>
          <td>983726.98</td>
        </tr>
      </tbody>
    </table>
  </body>
</html>
```

## Operation

- In REST gibt es **vier** Standard-Operationen (HTTP verbs) die auf einer Resource definiert sein können
  - **GET**: Eine Representation der Resource abrufen
  - **PUT**: Eine neue Resource unter dem URI anlegen oder eine existierende Resource unter dem gegebenen URI aktualisieren (Überschreiben)
  - **POST**: Resource anlegen für die kein URI bekannt ist, teilweises aktualisieren von Resource oder ausführen beliebiger Operationen
  - **DELETE**: Resource unter dem angegebenen URI löschen

## Operation Beispiele

- Wir können den URL <http://myshop.at/shop/rest/items/7> per **GET** anfragen um eine HTML Repräsentation eines Artikels zu bekommen
- Wir können per **POST** auf den URL <http://myshop.at/shop/rest/items> einen **neuen** Artikel im Shop anlegen

## Einfach verständlich

- Wenn wir Ressourcen als Dinge/Objekte betrachten die wir über einen URL **abrufen** und **manipulieren** können ist es **nicht notwendig** aufwändige **Beschreibungen** für Services zu erstellen
  - **GET, PUT, POST** und **DELETE** funktionieren immer gleich
- REST ist also kein Paradigma in dem man Services baut die 100erte verschiedene Funktionen anbieten sondern meist ein Framework mit dem man **wenige Basis-Funktionen** auf URLs anwendet und damit komplexe Prozesse aufbauen kann

## Sichere Operationen

- Es gibt bestimmte Regeln dafür wie sich ein REST Service verhalten soll
- Sogenannte sichere (**safe**) Methoden dürfen eine **Resource nicht verändern** (**GET** auf einen URL darf die Resource nicht ändern)
- Weiters wird erwartet dass die **PUT** Methode Idempotent ist. Bedeutet dass ein Aufruf von **PUT** mit den **gleichen Parametern** mehrfach hintereinander den Systemstatus in gleicher Weise beeinflussen muss wie ein einziger Aufruf (PUT kann man nicht zum zählen verwenden)

## GET

- **Sicher** und **idempotent**: mehrfaches **GET** auf <http://steinbauer.org/reports/2015/Q4/sales> darf den Report **nicht ändern**
- Wir würden ja auch bei mehrfachem Aufruf von <http://www.google.at> nicht die Startseite von Google ändern wollen

```
{
  "Non-Food": 287362.83,
  "Food": 827736.3,
  "Services": 983726.98
}
```

## DELETE

- **Nicht-Sicher** aber **idempotent**: **DELETE** auf <https://myshop.at/items/7> löscht einen Artikel kann aber mehrfach ausgeführt werden
- Annahme: Unser Backend Service beschwert sich nicht über ein DELETE auf einen nicht existierenden Artikel

## PUT

- **Nicht sicher** und **idempotent**: Schreiben eines Dokuments auf eine Resource Ändert die Resource

```
{  
  "id": 7,  
  "name": "Raspberry Pi",  
  "price": 35.9,  
  "stock": 82  
}
```

- Mehrfaches PUT des Artikels lt. JSON auf <https://myshop.at/items/7> wird zwar mehrere Transaktionen auslösen aus Benutzersicht aber gleicher Systemzustand

## POST

- **Nicht sicher, nicht idempotent**: verändert Resource(n) und wird bei wiederholter Ausführung den Systemzustand verändern
- z.B. Anlegen eines neuen Artikels POST auf <https://myshop.at/items>



## Safe and Idempotent

	safe	idempotent
GET	yes	yes
DELETE	no	yes
PUT	no	yes
POST	no	no

# Hypertext

- Wenn wir per **HTTP** im **Web** surfen enthält der **Response** vom Server (meist HTML) **Links** mit denen wir durch das Web **navigieren**
- REST Services sollen wenn sie sich in ihrem **Response** auf **andere Resource beziehen** ebenfalls **Links** auf diese Resource beinhalten
- z.B. Abfrage von einer Liste von Artikel unter dem URL <http://myshop.at/items> kann entweder direkt Daten liefern oder einfach eine Liste an URLs unter denen die Details einzelner Artikel verfügbar sind
- In der Praxis ist dies auch eine Frage der **Performance!**

<http://myshop.at/items>

Einzelner Request

```
{
  {
    "id":1,
    "name":"Raspberry Pi Model 2",
    "price":55.0,
    "stock":124
  },
  {
    "id":2,
    "name":"Raspberry Pi Model A",
    "price":25.0,
    "stock":4
  },
  {
    "id":3,
    "name":"Raspberry Pi Model B",
    "price":35.0,
    "stock":7
  },
  {
    "id":4,
    "name":"MicroSD Card 32 GB",
    "price":27.0,
    "stock":153
  }
}
```

Mehrere Request

```
http://localhost:8080/items/1',
'http://localhost:8080/items/2',
'http://localhost:8080/items/3',
'http://localhost:8080/items/4'
]

{
  "id":1,
  "name": "Raspberry Pi Model 2",
  "price": 55.0,
  "stock": 124
}
```

insgesamt 5 Server Round-Trips

# Stateless

- **REST** und **HTTP** als Übertragungsprotokoll sind **Stateless**
- **State** wird erst durch erweiterte Frameworks (HTML/ Web **Session Cookies**) hergestellt
- **REST** verlangt, dass der Status entweder in der **Resource** abgelegt wird oder am **Client** gehalten wird
- **Clients** sind für **Anwendungsstatus** zuständig, **Server** für **Resource Status**

# Vorteile Statelessness

- Clients sind **unabhängig** von **Änderungen** am **Server** System (Reboots, Software-Updates, etc.)
- Redundanz ist leichter herzustellen (Server stirbt > Requests können vom **Load-Balancer** an einen anderen Server gegeben werden)
- Bietet einfachste Möglichkeit die **Performance** durch Proxy Services zu verbessern (GET auf Resource die sich selten ändern)





---

---

---

---

---

## Entwicklungsumgebung

- Eclipse Mars 1
- Java JRE 1.8
- Erweiterungen aus den JBoss Tools
  - WebService Tester
- Tomcat 8.0.28



N:\SL\Alle\WebServiceWorkshop\_01122015

**Im Home Verzeichnis auspacken**

---

---

---

---

---

## Entwicklungsumgebung einrichten

- In der ausgepackten Entwicklungsumgebung im Ordner `software\eclipse\eclipse.exe` starten
- Beliebigen Workspace am Rechner einrichten
- Danach die JBoss **WebService Tester** View starten

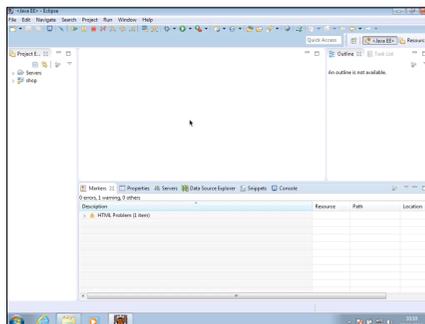
---

---

---

---

---



---

---

---

---

---

## REST Service aufrufen

Wir rufen Finanzdaten zu Unternehmen ab

<http://dev.markitondemand.com/MDApis/#companylookup>

URLs die Company Lookups machen können

<http://dev.markitondemand.com/Api/v2/Lookup/json>

<http://dev.markitondemand.com/Api/v2/Lookup/xml>

Parameter

**URL**

<http://dev.markitondemand.com/Api/v2/Lookup/json?input=Google>

**Request**



## Agenda

- Einführung zu Webservices
- Grundlagen zur Service Seite mit Java (JAX-RS, Jersey)
- Web basierter Client mit AngularJS
- Methoden Aufrufe (Actions) vs. CRUD
- Fortgeschrittenes und Reflexion

## REST Frameworks

- Frameworks sind in einer **Vielzahl** an **Sprachen** verfügbar
- Häufig kann REST sehr einfach in **bestehenden Code** eingebaut werden
- Ziel von Frameworks ist es meist **JSON/XML** (Repräsentation) und Request **Methoden** vom Entwickler zu abstrahieren

## PHP ohne Framework

- Header auf bestimmten **Content-type** setzen
- Eingebaute Funktion **json\_encode** konvertiert PHP Objekte in JSON Repräsentation
- Wird häufig als Einstieg in REST gefunden

```
<?php
error_reporting(E_ALL);
include_once('dbsettings.php');
header('Content-type: application/json; charset=utf-8');

$data_model = array();
$json = json_encode($data_model);
echo $json;
?>
```

## Node.js / JavaScript

- Unter Verwendung des Plugin **restify**

```
npm install restify

var restify = require('restify');
var hostname = 'restify.tk.jku.at';
var port = 80;
var server = restify.createServer({
  name: 'demoapp'
});
server.use(restify.queryParser());
server.use(restify.bodyParser());
server.use(restify.CORS());
server.listen(port, hostname, function() {
  console.log('listen %s:%s', server.name, server.url);
});
```

## A Node.js REST service

```
var path = '/jobs';
server.get({ path: path, version: '0.0.1', listJobs};
server.get({ path: path + '/:jobid', - }, getJob);
server.post({ path: path, - }, postNewJob);
server.del({ path: path + '/:jobid', - }, deleteJob);
```

```
function getJob(request, response, next) {
  res.setHeader('Access-Control-Allow-Origin', '*');
  console.log('getting job: ' + request.params.jobid);
  --
  response.send(200, jsonString);
  return next();
}
```

## RESTful **Java** Service

	Pro	Con
Dropwizard	Jetty, Jackson, Debugging	Fehler immer <code>text/plain</code>
Jersey	<b>JAX-RS Referenz Impl.</b>	V1 vs. V2, Debugging
Ninja Web Framework	Schnell, Standalone	Doku, Community
Play Framework	Akka Stateless Architecture	Kein Servlet, Scala
RestExpress	Performance, Best Practice	Wenig Doku (eBook)
Restlet	<b>Powerful, EE, Standalone</b>	Geschlossene Community
Restx	MongoDB, Micro, Schnell	Sehr junges Framework
Spark Framework	AngularJS, generisch Web	Große Projekte
Spring	Im Spring Ökosystem	Braucht Spring ...

## JAX-RS Spezifikation

- Wie andere Standards in **Java Spezifikation** mit mehreren **Implementierungen**
- **API** stellt hauptsächlich **Interfaces** und **Annotations** zur Verfügung
- Populärste Implementierungen sind **Jersey** und **Restlet**

## JAX-RS Web Resource

- In JAX-RS können Java **Objekte** (POJOs) **deklarativ** als REST Service **freigegeben** werden
- Konfiguration spezifiziert Packages/Klassen die mit Annotations versehen sind
- Unterstützt diverse Representations über Drop-In-Converter
- **Objekte** werden über **JAXB** Mechanismen in **Repräsentationen** überführt (XML als Standard)

---

---

---

---

## JAX-RS Client

- Die **Client**-Seite ist ebenfalls **Teil** der **Spezifikation**
- Ähnliche technologische Abhängigkeiten (JAXB)
- Standardisierte Zugriffspfade auf beliebige REST Dienste
- Ermöglicht nahtloses Übertragen von POJOs  
Java-Service <-> REST <-> Java-Client

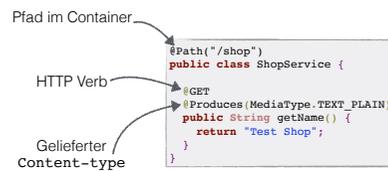
---

---

---

---

## Deklarative Freigabe



---

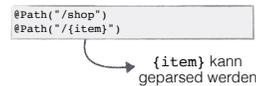
---

---

---

## @Path

- Definiert den **Pfad** zur implementierten Resource relativ zum **Servlet Container**
- Kann auf **Klasse** oder eine **Methode** angewendet werden
- Darf **Parameter** (`@PathParam`) enthalten



---

---

---

---

## @GET / HTTP Verb

- Die in REST verwendeten HTTP Verbs stehen als Annotation zu Verfügung: **@GET**, **@PUT**, **@POST**, **@DELETE**
- Das JAX-RS kompatible Framework ruft je nach **Request Methode** die korrekte **Methode** am **POJO** auf
- Es können aber für **einen Pfad** mehrere Methoden mit dem **gleichen Verb** aber unterschiedlichem **Content-type** deklariert werden

---

---

---

---

---

## @Produces

- Gibt an welche(n) **Content-type** (Representation) die Methode generieren kann
- Kann ein einzelner MIME-type oder eine Liste an Types sein
- Das JAX-RS Framework wählt aufgrund des angegebenen **Content-types** und der Methode die korrekte POJO Methode aus
- Der Client kann den akzeptierten **Content-type** über den Accept Header setzen

Accept: application/json

---

---

---

---

---

## Jersey als Referenz-Implementierung von JAX-RS

- Jersey ist die **offizielle** Referenz-Implementierung zum **JAX-RS Standard**
- Open-Source
- Läuft in Servlet Containern
- Anbindung von Representation Konvertern durch Service Provider Interfaces



<https://jax-rs-spec.java.net/>

---

---

---

---

---

## /download/

- Grundsätzlich zwei Möglichkeiten
  - Maven / Ivy / Gradle Dependencies auf <http://repol.maven.org>
- Uber-JAR als direkter Download von
  - <https://jersey.java.net/download.html>

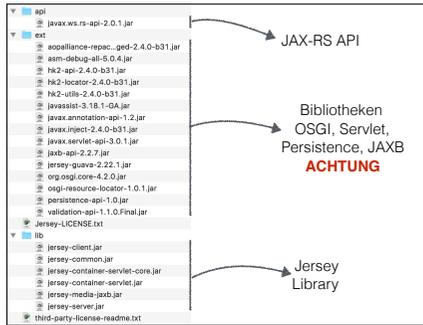
---

---

---

---

---



## Drop-Ins / SPI

- Grundsätzlich **schreibt** die Spezifikation von JAX-RS **keine Repräsentationstypen** vor
- Es ist in Jersey (vermutlich daher) keine Bibliothek für JSON eingebunden
  - Die Bibliotheken **Jackson** und **Genson** können als Drop-In verwendet werden
  - SPI: Wird eine Lib am Classpath gefunden wird sie verwendet

## /download/genson

- Im Demo-Projekt wird **Genson** verwendet
- Ist wiederum **JAXB** kompatibel > JAXB Annotationen können für **XML** und **JSON** Repräsentationen verwendet werden
- Einziges JAR: `genson-1.3.jar`

<http://owlike.github.io/genson/>

## Jersey Demo Projekt

- Online Shop Anwendung **JAX-RS / Jersey**
- Ausgeführt als **Dynamic Web Project**
- Deployment auf Apache **Tomcat 8.x.x**
- Datenhaltung in **MapDB** (KeyValue Store der File und Memory based arbeiten kann)
- Bietet ein REST Interface unter <http://localhost:8080/shop/rest> an und ein User Interface unter <http://localhost:8080/shop/>

## MapDB

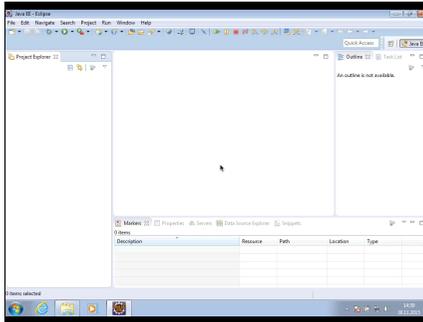
- Sehr einfacher Key-Value Store; erlaubt uns das Erzeugen einer DB In-Memory oder persistent in einem File
- Im Demo-Projekt hinter der Klasse `ShopDAO` versteckt
- Mit `Settings.USE_FILE_DB` kann gesteuert werden ob In-Memory oder File-basierte DB
- File liegt im TEMP Verzeichnis des Benutzers als `shop.mapsdb`
- Einzelnes JAR: `mapdb-1.0.8.jar`



<http://www.mapdb.org>

## Log4J

- Wäre doch überraschend gewesen eine Demo ohne ein Logging Framework?
- Log4j JAR: `log4j-1.2.16.jar`
- Plus Konfiguration in `src/log4j.properties`



## web.xml

Package mit Service Objects

Servlet Klasse

```
<!-- REST Service dispatcher configuration -->
<!-- See how the parameter provider.packages points to our shop package -->
<servlet>
  <servlet-name>REST Service Servlet</servlet-name>
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-name>jersey.config.server.provider.packages</param-name>
    <param-value>at.gv.landoo.shop.rest</param-value>
  </init-param>
  <init-param>
    <param-name>javax.ws.rs.Application</param-name>
    <param-value>at.gv.landoo.shop.ServiceResourceConfig</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

Erweiterte Konfiguration für einzelne Resources

### web.xml

REST Servlet

```
<!-- Servlet Mappings -->
<servlet-mapping>
  <servlet-name>REST Service Servlet</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

Im Pfad /rest/\*

---

---

---

---

---

---

---

---

---

---

### Projekt Importieren + Starten

- Projekt `empty-rest-project.zip` importieren
- Tomcat 8 Server im Eclipse anlegen, konfigurieren und starten
- Das leere Projekt auf Tomcat deployen und versuchen Tomcat zu starten

---

---

---

---

---

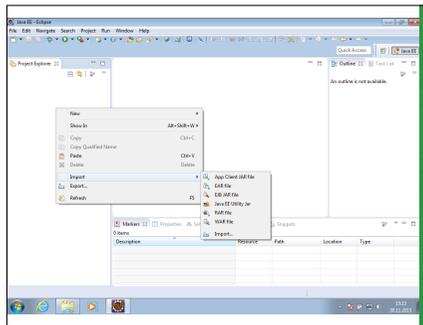
---

---

---

---

---



---

---

---

---

---

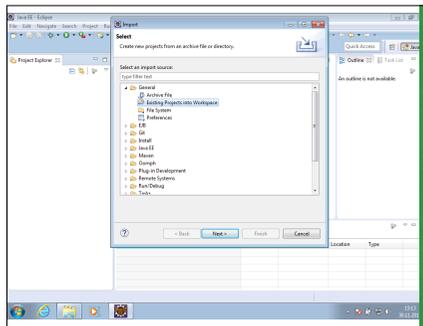
---

---

---

---

---



---

---

---

---

---

---

---

---

---

---





## Accept Header

- Implementieren sie außerdem eine Methode `getItemsPlain` oder `getItemsHTML`, die eine Text bzw. HTML Representation der Items liefert
- Rufen sie den URL mit verschiedenen Accept Headers auf

• `Accept=text/html`,  
`Accept=application/json`, etc.



Request Details

Prompt for Basic Authentication

Request Headers

Accept=application/json Add

Accept=application/json Edit

## Agenda

- Einführung zu Webservices
- Grundlagen zur Service Seite mit Java (JAX-RS, Jersey)
- Web basierter Client mit AngularJS
- Methoden Aufrufe (Actions) vs. CRUD
- Fortgeschrittenes und Reflexion



## Agenda

- Einführung zu Webservices
- Grundlagen zur Service Seite mit Java (JAX-RS, Jersey)
- Web basierter Client mit AngularJS
- Methoden Aufrufe (Actions) vs. CRUD
- Fortgeschrittenes und Reflexion

# Angular JS



- Von **Google** getriebenes **JavaScript** Framework
- Unterstützt das Binden von **JavaScript Models** an **HTML5 UIs**
- Bietet **direkten Support** für **REST** basierte Interfaces
- Models sind in sogenannten **Scopes** abgebildet und können über **{{var}}** Patterns eingebunden werden

<https://angularjs.org>

Controller Include      Methoden Call      Conditionals

```
<html>
<head>
<link rel="stylesheet"
href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css
bootstrap.min.css">
<link rel="stylesheet" href="css/shop.css">
<script
src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/
angular.min.js"></script>
<script src="js/ItemController.js"></script>
</head>
<body ng-app="angularShopApp" ng-controller="ItemController">
<div id="container" class="container">
<div class="row">
<div id="title" class="col-md-6">
<h1>Shop Application</h1>
</div>
<div id="controls" class="col-md-6 shopControls">
<button type="button" class="btn btn-default"
ng-click="placeOrder">
ng-show="cart.length" Place Order</button>
<button type="button" class="btn btn-default"
ng-click="emptyCart()"
ng-show="cart.length">Empty Shopping Cart</button>
<button type="button" class="btn btn-default"
ng-click="loadItems()">Update Item List</button>
</div>
</div>
</div>
</body>
</html>
```

Gebundene Scope Variable      Conditionals

```
<div class="row alert alert-danger" ng-show="errorMessage">
<div class="col-md-12">
<div>
{{ errorMessage }}
</div>
</div>
<div id="serverResponse" ng-show="serverResponse">
{{ serverResponse }}
</div>
</div>
</div>
```

Angular Table Loop  
Click auf TR

```
<tr ng-repeat="item in items | orderBy: 'id'" ng-click="addToCart(item)"
class="clickable">
<td class="center">{{ item.id }}</td>
<td>{{ item.name }}</td>
<td class="right">{{ item.price }}</td>
<td class="right">{{ item.stock }}</td>
</tr>
```

Scope Variablen  
(in HTML sichtbar)

Scope Funktionen

```
var angularShopApp = angular.module("angularShopApp", []);
angularShopApp.controller('ItemController', ['$scope', '$http',
function($scope, $http) {
$scope.customerName = "Thomas Testthuster";
$scope.items = [];
$scope.cart = [];

// function retrieve items from the back-end
$scope.loadItems = function() {
$http.get('http://localhost:8080/shop/rest/items',
config).then(function(response) {
$scope.items = response.data;
console.log('Received response');
console.log($scope.items);
}, function(error) {
$scope.errorMessage = renderError(error);
$scope.serverResponse = error.data;
});
});
});
```

## AngularJS Project Importieren

- Entfernen sie ihr Projekt **shop** aus dem Workspace
- Importieren sie stattdessen **angular-rest-project.zip**
- Dieses Projekt enthält zusätzlich **HTML, CSS** und **JS** Dateien im **WebContent**

## AngularJS Client

- Wenn unser REST Service richtig gebaut wurde sollte es möglich sein <http://localhost:8080/shop/> aufzurufen
- Der Controller ruft per REST <http://localhost:8080/shop/rest/items> auf und erwartet einen **application/json** kodiertes Response
- Analysieren sie den Code in **ItemController.js / loadItems**

## Bestellung aufgeben

- Das Datenmodell enthält bereits die Klassen **Order** und **OrderItem** (Zeilen auf einer Order)
- Erstellen sie eine neue Klasse **OrdersService** im Package **at.gv.landooe.shop.rest**
- Ermöglichen sie es dass per **POST** eine neue Order gegen den URL <http://localhost:8080/shop/rest/orders> aufgegeben wird

## Bestellung aufgeben

```
@POST
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
@Consumes({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
public Order postOrder(Map<String, Object> jsonMap) {
    return null;
}
```

ShopDAO.saveOrder kann verwendet werden um eine Bestellung zu speichern

Map die das JSON Objekt enthält Erlaubt komfortables Lesen eines Objektes

```
private Order parseOrder(Map<String, Object> jsonMap)
```

## Bestellung Testen

- Verwenden sie den bestehenden AngularJS Client um eine Bestellung aufzugeben
- Der Client stellt durch klicken auf Artikel einen Warenkorb zusammen; dieser kann durch **Place Order** abgeschickt werden
- Der Client holt danach automatisch neue Daten von <http://localhost:8080/shop/rest/items> ab
- Testen sie das Vorgehen auch im Eclipse per **Web Service Tester**

---

---

---

---

---

---

## Agenda

- Einführung zu Webservices
- Grundlagen zur Service Seite mit Java (JAX-RS, Jersey)
- Web basierter Client mit AngularJS
- Methoden Aufrufe (Actions) vs. CRUD
- Fortgeschrittenes und Reflexion

---

---

---

---

---

---

## CRUD

- REST Dienste eignen sich besonders für das **Create, Retrieve, Update, Delete** Schema

REST	POST	GET	PUT	DELETE
CRUD	Create	Retrieve	Update	Delete

---

---

---

---

---

---

## CRUD in REST

- Liste an Elementen über einen Basis-URL abrufbar <http://localhost:8080/shop/rest/items>
- Gleicher URL wird zum Erzeugen von Ressourcen verwendet
- Details, Update (PUT) und Delete über Detail URL <http://localhost:8080/shop/rest/items/7>

---

---

---

---

---

---

## @PathParam

- Ermöglicht es Patterns aus URLs zu lesen
- Teile des URLs/Pfads können als Parameter verwendet werden

```
@Path("/items")
public class ItemService {
    @Context
    UriInfo uriInfo;
    @Context
    Request request;

    @Path("/{item}")
    public ItemService getItem(@PathParam("item") String id) {
        return new ItemService(this.uriInfo, this.request, id);
    }
}
```

## @Context

- JAX-RS Context lässt sich über die @Context Annotation in ein POJO injizieren
- Informationen zum URL und Request sind so verfügbar

```
@Path("/items")
public class ItemService {
    @Context
    UriInfo uriInfo;
    @Context
    Request request;

    @Path("/{item}")
    public ItemService getItem(@PathParam("item") String id) {
        return new ItemService(this.uriInfo, this.request, id);
    }
}
```

```
@Path("/items")
public class ItemService {
    @Context
    UriInfo uriInfo;
    @Context
    Request request;

    @Path("/{item}")
    public ItemService getItem(@PathParam("item") String id) {
        return new ItemService(this.uriInfo, this.request, id);
    }
}
```

Aufrufe auf den Pfad  
**http://localhost:8080/shop/rest/items/{item}**  
werden an den neuen Handler  
ItemService delegiert

ItemService: /shop/rest/items > Liste  
ItemService: /shop/rest/items/7 > einzelnes Item

```
public class ItemService {
    @Context
    UriInfo uriInfo;
    @Context
    Request request;

    String id;

    public ItemService(UriInfo uriInfo, Request request, String id) {
        this.uriInfo = uriInfo;
        this.request = request;
        this.id = id;
    }

    @GET
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public Item getItem() {
        return ShopDAO.getItem(Integer.parseInt(this.id));
    }
}
```

## Einzelne Artikel verwalten

- Erzeugen sie die Klasse `ItemService` im Paket `at.gv.landooe.rest`

```
public class ItemService {
    @Context
    UriInfo uriInfo;
    @Context
    Request request;
    String id;
    public ItemService(UriInfo uriInfo, Request request, String id) {
        this.uriInfo = uriInfo;
        this.request = request;
        this.id = id;
    }
}
```

- Delegieren sie Aufrufe aus `ItemsService` für einzelne Artikel

## Einzelne Artikel verwalten

```
@Path("/items")
public class ItemsService {

    @Context
    UriInfo uriInfo;
    @Context
    Request request;

    /** Other code */

    @Path("/{item}")
    public ItemService getItem(@PathParam("item") String id) {
        return new ItemService(this.uriInfo, this.request, id);
    }

    /** Other code */
}
```

## AngularJS CRUD

- Im unteren Bereich der AngularJS Demo App findet sich ein Formular zum Editieren von Artikeln
- Das Formular ist bereits mit dem Scope Model `$scope.editItem` verbunden
- Implementieren sie die Funktionen `$scope.getItem`, `$scope.postItem`, `$scope.putItem` und `$scope.deleteItem`

`$scope.editItem` ist wie ein Artikel / `Item` aufgebaut

`id`  
`name`  
`price`  
`stock`

```
$scope.getItem = function() {
    $http.get('http://localhost:8080/shop/rest/items/' + $scope.editItem.id,
    config).then(function (response) {
        $scope.editItem = response.data;
    }, function (error) {
        $scope.errorMessage = renderError(error);
        $scope.serverResponse = error.data;
    });
};
```

GET Methode am Service muss ein JSON liefern  
dann können wir dies direkt hier setzen  
die Formularfelder sind auf Element von  
`$scope.editItem` gebunden

GET holt einen Artikel aus der DB und gibt ihn als JSON / XML aus

```
@GET
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
public Item getItem(@Integer.parseInt(this.id)) {
    return shopDAO.getItem(Integer.parseInt(this.id));
}
```

Testen sie  
Get Item  
im Frontend

Edit Items here

ID	4
Name	MicroSD Card 32 GB
Price	27
Stock	152
Post Item Put Item Get Item Delete Item	

POST / \$http.post / @POST

ItemService.java

```
@POST
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
@Consumes({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
public Item putItem(Item item) {
    return shopDAO.saveItem(item);
}
```

JSON kann vom Framework  
direkt auf POJOs gemapped werden

ItemController.js

```
$scope.putItem = function() {
    $http.post('http://localhost:8080/shop/rest/items', $scope.editItem,
    config).then(function(response) {
        $scope.loadItems();
        $scope.infoMessage = 'Successfully created a new item: ' +
        response.data.id;
        $scope.editItem = response.data;
    }, function(error) {
        $scope.errorMessage = renderError(error);
        $scope.serverResponse = error.data;
    });
};
```

PUT / \$http.put / @PUT

ItemService.java

```
@PUT
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
@Consumes({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
public Item putItem(Item item) {
    return shopDAO.saveItem(item);
}
```

ItemController.js

```
$scope.putItem = function() {
    $http.put('http://localhost:8080/shop/rest/items/' + $scope.editItem.id,
    $scope.editItem, config).then(function(response) {
        $scope.loadItems();
        $scope.infoMessage = 'Successfully updated the item: ' +
        response.data.id;
        $scope.editItem = response.data;
    }, function(error) {
        $scope.errorMessage = renderError(error);
        $scope.serverResponse = error.data;
    });
};
```

DELETE / \$http.delete / @DELETE

ItemService.java

```
@DELETE
public void deleteItem() {
    shopDAO.deleteItem(Integer.parseInt(this.id));
}
```

In unserem Fall idempotent implementiert  
kein Fehler beim Löschen nicht existierender Artikel

ItemController.js

```
$scope.deleteItem = function() {
    $http.delete('http://localhost:8080/shop/rest/items/' +
    $scope.editItem.id, config).then(function(response) {
        $scope.editItem = {};
        $scope.editItem.id = 0;
        $scope.editItem.name = '';
        $scope.editItem.price = 0;
        $scope.editItem.stock = 0;
        $scope.loadItems();
    }, function(error) {
        $scope.errorMessage = renderError(error);
        $scope.serverResponse = error.data;
    });
};
```

## Komplettes Projekt

- Wir können jetzt das Projekt `finished-rest-project.zip` anstatt unseres aktuellen Projektes importieren
- Es enthält alles was wir bisher gemacht haben und noch etwas Zusatzfunktionalität für den nächsten Block

---

---

---

---

---

## Agenda

- Einführung zu Webservices
- Grundlagen zur Service Seite mit Java (JAX-RS, Jersey)
- Web basierter Client mit AngularJS
- Methoden Aufrufe (Actions) statt CRUD
- Fortgeschrittenes und Reflexion

---

---

---

---

---

## Fortgeschrittenes und Reflexion

- Weiterführende Themen kurz angeschnitten
  - Authentifizierung
  - Versionierung
- REST Kritikpunkte

---

---

---

---

---

## Authentifizierung

- Kann als Aufgabe der **Anwendung** gesehen werden
  - REST Methoden die auf ein erfolgreiches Login mit **Session Token** reagieren
- Aufgabe des **Containers**
  - HTTP **Basic Authentication**
  - **OAuth** 1 und 2 Standard

---

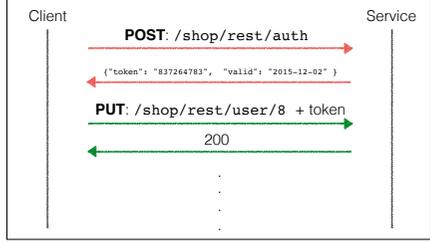
---

---

---

---

## Session Tokens



---

---

---

---

---

---

---

---

## Basic Authentication

- Im HTTP Protokoll **"eingebaut"**
- Username und Passwort wird in einem **HTTP Header** mitgesendet
- JAX-RS Implementierungen könnten **deklarativ** auf Authentifizierung konfiguriert werden
- **Benutzerdatenbank** kommt dann aus dem **Servlet Container**

---

---

---

---

---

---

---

---

## Tomcat Benutzer

- Tomcat hat im Eclipse Workspace ein **Servers** Project
- In diesem sind alle Konfigurationen abgelegt
- Default Benutzer-DB in **tomcat-users.xml**



---

---

---

---

---

---

---

---

tomcat-users.xml

```
<tomcat-users version="1.0" xmlns="http://tomcat.apache.org/xml" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://tomcat.apache.org/xml tomcat-users.xsd">
  <role rolename="admin" />
  <user username="matthias" password="matthias" roles="admin" />
</tomcat-users>
```

Wird von der server.xml referenziert  
andere User DBs können per JNDI  
angekoppelt werden

---

---

---

---

---

---

---

---

Der Servlet Container kann mit Security Constraints ausgestattet werden

web.xml

```
<security-constraint>
<web-resource-collection>
  <web-resource-name>Secured</web-resource-name>
  <url-pattern>/rest/admin</url-pattern>
</web-resource-collection>
<auth-constraint>
  <role-name>admin</role-name>
</auth-constraint>
</security-constraint>
<security-role>
  <role-name>admin</role-name>
</security-role>
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Login</realm-name>
</login-config>

<init-param>
  <param-name>javax.ws.rs.Application</param-name>
  <param-value>at.gv.landoo.shop.ServiceResourceConfig</param-value>
</init-param>
```

Erweiterte Konfiguration für einzelne Resources

ServiceResourceConfig.java

```
@ApplicationPath("/")
public class ServiceResourceConfig extends ResourceConfig {
  public ServiceResourceConfig() {
    super(AdminService.class);
    register(RolesAllowedDynamicFeature.class);
  }
}
```

AdminService.java

```
@Path("/admin")
public class AdminService {
  static Logger logger = Logger.getLogger(AdminService.class);

  @RolesAllowed({ "admin" })
  @POST
  public void createItem(String content) {
    logger.info("Admin user posted " + content);
  }
}
```

## AngularJS Beispiel

```
$scope.putUser = function() {
  $http.defaults.headers.common['Authorization'] = 'Basic ' +
    'Basic: encode('matthias' + ':' + 'matthias');
  $http.put('http://localhost:8080/shop/rest/users/' + $scope.editUser.id,
    $scope.editUser, config).then(function(response) {
    $scope.editUser = response.data;
  }, function(error) { alert('error'); });
};
```

Wird

`$http.defaults.headers.common['Authorization']` gesetzt wird Basic Authentication immer mitgesendet

## Test Authentifizierung

- Im kompletten Projekt ist der URL <http://localhost:8080/shop/rest/admin> durch Basic Authentication geschützt
- Legen sie einen Benutzer in `tomcat-users.xml` an und testen sie die Basic Authentication mit dem **Web Service Tester**

```
<tomcat-users version="1.0" xmlns="tomcat-users.xsd">
  <role rolename="admin" />
  <user username="matthias" password="matthias"
    roles="admin" />
</tomcat-users>
```

## Versionierung

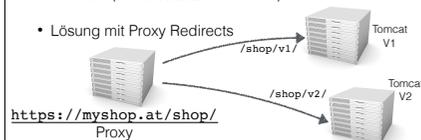
- Wie jede andere Software auch unterliegen REST Services einer Wartung, Weiterentwicklung und Verbesserung > **Versionierung** notwendig
- Da der Service über Unternehmensgrenzen hinweg genutzt wird können Service Nutzer nur bedingt zum Software-Upgrade gezwungen werden
- Services müssen daher oft versioniert werden

## Ansätze zur Versionierung

- **URL Pfad**
  - Findet bei den großen Playern (Yahoo, Flickr, Google) derzeit noch häufig Anwendung
  - <http://dev.markitondemand.com/Api/v2/lookup/json>
- **HTTP Header**
  - Entweder eigener Header für die Version
  - Codierung der Version im **Content-type**

## Pfad Versionierung

- Direkt in Jersey und Servlet Container umständlich zu realisieren
- Man müsste de facto Kopien der Service Klassen erstellen (siehe `@Path` Annotation)
- Lösung mit Proxy Redirects



## Version im Header

- Es kann ein eigener Request Header verwendet werden z.B:
  - `Land00E-REST-API-Version: v2.0.1`
- Dann beschreibt ein URL nach wie vor eine Resource (unabhängig der Version)
- URLs können versendet werden, sie agieren wie in der aktuellsten Version
- Legacy Clients können explizit eine API Version definieren

## VersionService.java

```
@Path("/version")
public class VersionService {
    public static final String CURRENT_STABLE_VERSION = "1.0.3";

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getVersion(@HeaderParam("LandOOE-REST-API-Version")
        String version) {
        version = version == null ? CURRENT_STABLE_VERSION : version;
        switch (version) {
            case "1.0.3":
                return getVersion103();
            case "2":
                return getVersion2();
        }
        throw new WebApplicationException(Response.Status.NOT_IMPLEMENTED);
    }

    private String getVersion103() { return "Version 1.0.3 (Stable)"; }
    private String getVersion2() { return "Version 2 (Beta)"; }
}
```

## Accept Header

- Ähnlich wie die Definition des eigenen Header aber im MIME des **Accept** Header kodiert
  - `application/vnd.landooe.v2+json`
- Komplizierter in der Handhabung in JAX-RS
- Schwieriger für die Nutzer weil es nicht mehr reicht einen URL zu kennen; man muss konkret wissen wie ein Request aussehen darf

## Test eines Version Header

- Rufen sie den URL <http://localhost:8080/shop/rest/version> mit dem **Web Service Tester** auf
- Was passiert mit und ohne Request Header **LandOOE-REST-API-Version**
- Es sind die Versionen 1.0.3 und 2 hinterlegt

Toll für CRUD aber nicht für Service Methoden

Rein formell ist jede Berechnung als Resource beschreibbar

**GET** <http://steinbauer.org/sum/a=2&b=3>  
(sicher, cachebar)

REST sicher besser geeignet für Projekte wo auf Ressourcen immer das gleiche Mustern an Manipulation angewendet wird

“ Es gibt keine  
Contracts zwischen  
Anbieter und Konsument ”

Hier sind traditionelle Services mit  
der WSDL überlegen

WSDL ist aber häufig dazu da Messages  
zu beschreiben. Diese Messages sind XML,  
REST könnte auch XML transportieren

Es wäre trivial Schema (XSD) Checks in die eigenen  
Service Methoden einzubauen

---

---

---

---

---

“ REST unterstützt  
keine Transaktionen ”

Transaktionen über Service Calls hinweg  
wie bei Spring oder SOAP sind nicht möglich

Einzelner HTTP Request ist die größte  
Einheit für eine Transaktion

**Workaround:**

Temporäre Resources werden durch mehrere **PUT & POST**  
Aufrufe geschrieben

Ein POST zu einem Transaction URL prozessiert Daten in  
Transaktion

---

---

---

---

---

“ Kein  
Publish-Subscribe  
Support ”

Benachrichtigungen müssen per Polling abgerufen werden

Stalling per GET auf definierten URLs ist in der AJAX-Welt  
seit Jahren erprobt

GET Aufrufe sind im Web stark optimiert  
(Server, Proxy, Client, Transport Protokoll, etc.)

---

---

---

---

---

“ Langlaufende  
asynchrone  
Prozesse können  
nicht abgebildet werden ”

In HTTP gibt es den Status Code **202 Accept**  
dieser kann verwendet werden um zu signalisieren  
dass ein Request verarbeitet wird, aber derzeit keine  
Antwort vorliegt

Client kann per Polling auf einem URL  
so lange mit **202** warten bis mit **200**  
geantwortet wird

---

---

---

---

---

## Problempunkte

- REST Services mit sehr **vielen** verschiedenen **Objekttypen** (Ressourcen) sind **aufwändig** in Erstellung und Wartung (viele Service Klassen)
- Echtzeit-System mit garantierten Antwortzeiten und Systeme mit Bandbreiten-Beschränkung sind problematisch (Client kann unter Umständen für eine Aufgabe eine ganze Reihe von Requests absetzen müssen)
- REST ist kein Standard verschieden Frameworks folgen den unterschiedlichen Konventionen und Paradigmen



Die vorliegenden Schulungsunterlagen und das zugehörige Material sind urheberrechtlich geschützt. Urheber ist Matthias Steinbauer, Gartenweg 5, 4225 Luffenberg.

Als Schulungsteilnehmer dürfen die Schulungsunterlagen zur Information anderer Mitarbeiter im Haus und als Basis für eigene Entwicklungen verwendet werden. Dies schließt auch die Verwendung von Kopien der Demo Beispiele mit ein. Dies schließt auch mit ein, dass Mitarbeiterinnen weiche an keinem Training teilgenommen haben anhand der vorliegenden Schulungsunterlagen im Haus durch Peers unterwiesen werden. Die Schulungsunterlagen dürfen außerdem an staatlich anerkannten Bildungseinrichtungen in Österreich (Schulen, Hochschulen) frei verwendet werden sofern ein Hinweis auf den Urheber erhalten bleibt.

Explizit untersagt ist die kommerzielle Weiterverbreitung des Materials bei Beratungsdienstleistungen oder Trainings gegenüber Dritten.