Fachhochschule Köln
Cologne University of Applied Sciences

Web-basierte Anwendungen 2

Dokumentation
Smart Buildings

Ein Projekt im Rahmen des Studiengangs Medieninformatik an der Fachhochschule Köln
Campus Gummersbach

Christian Ries
und
Bruno José Carvalho Gonzaga

Sommersemester 2013
Fachsemester: 4

Inhaltsverzeichnis

1. Smart Buildings – Beschreibung

Entwicklung eines Systems zur Abfrage, Steuerung und Abonnierung von Gebäude-spezifischen Daten & Funktionen. Dabei wird auf REST & Publish/Subscribe gesetzt.

Die oben genannten Daten beziehen sich dabei auf, von diversen Sensoren, erfasste Werte. Diese Sensoren werden in zwei Kategorien gegliedert:

- Sensoren die zeit-diskret eine bestimmte Art von Wert erfassen und synchron abgefragt werden können (Stromverbrauch, Temperatur, Luftfeuchte,)

- Sensoren die bei einer Veränderung eines Zustandes eine asynchrone Meldung auslösen (Feuermelder, Türklingel, Bewegungsmelder, Fenster-/ Türkontake) Diese verschiedenen Meldungs-Typen werden einzelnen Topics zugeordnet die von den Benutzern abonniert werden können.

Durch die Verwendung beider Sensortypen ergibt sich ein System zur Überwachung der Energiebilanz, sowie zur Überwachung der Sicherheit eines Gebäudes.

Zusätzlich zu dieser Überwachung wird einem Benutzer die Möglichkeit gegeben aktiv die Funktionen des System zu steuern. z.B.: Raumtemperatur Soll-Wert einstellen, Licht ein-/ausschalten, Verschattung steuern.

Über eine mobile Anwendung bzw. Client-Software für stationäre Rechner, wird gewährleistet dass ein Benutzer diese Überwachung bzw. Steuerung auch aus der Ferne tätigen kann.

2. Szenarien

2.1. Energieüberwachung

Dieses System kann z.B. Anwendung in einem Bürogebäude finden. Man nehme das Beispiel eines Chefs der versucht die Energiebilanz seiner Gesellschaft zu verbessern. Durch die verschiedenen Sensoren ist es möglich eine Statistik über den Energieverbrauch zu erstellen. Anhand dieser Statistik können Schwachstellen erkannt, und mit Hilfe des Systems ausgeschaltet werden. Die Schwachstellen könnten z.B. eine ungünstig eingestellte Heizung (Tag-/Nachttemperatur), Geräte mit einem sehr hohen Energieverbrauch, schlechte Belüftung oder ungünstige Verschattung von Räumen sein.

2.2. Alarmfunktion

Die zweite Funktion ist die Überwachung des Gebäudes bezogen auf seine Sicherheit. Eine Sicherheitsfirma oder beispielsweise die Polizei benutzt die Anwendung um z.B. Bewegungsmelder und Tür-/Fensterkontakte zu überwachen. Wird ein Sensor/Melder ausgelöst so wird eine Meldung abgesetzt um über diese Veränderung zu berichten. So kann sofort auf einen möglichen Einbruch reagiert werden. Weiter könnte z.B. die Feuerwehr auf auslösende Feuermelder reagieren und möglichst schnell einen entstandenen Brand bekämpfen.

2.3. Steuerung

Das System bietet je nach Anwender die Möglichkeit der Steuerung einzelner Funktionen.
Ein Mitarbeiter des Gebäudes zum Beispiel hätte die Möglichkeit nach Feierabend von daheim aus überprüfen zu können ob er das Licht in seinem Büro angelassen hat oder die Heizung sinnlos aufgedreht ist.
Anhand der Steuerungsfunktion könnte er dies durch direkten Zugriff sofort ändern. Eine positive Energiebilanz könnte daraus resultieren.

Oder man stelle sich auch den Luxus vor im Winter früh morgens in sein Büro zu kommen, welches schon auf 20 Grad vorgeheizt wurde.

3. Funktionen

REST (synchron)

- Abfrage der Energiedaten (aktueller Gesamtenergieverbrauch)
- Abfrage der Energiedaten je Raum (aktueller Energieverbrauch)
- Abfrage der Innentemperatur (IST) je Raum
- Abfrage der Luftfeuchtigkeit je Raum
- Abfrage der Verschattung je Raum
- Abfrage der Lichtquellen je Raum
- Abfrage der Steckdosen je Raum
- Steuerung der Innentemperatur (SOLL) je Raum
- Steuerung der Verschattung je Raum
- Steuerung der Lichtquellen je Raum
- Steuerung der Steckdosen je Raum

Publish / Subscribe (asynchron)

- Meldung von Bewegungen (Bewegungsmelder)
- Meldung von Tür-, Fensterkontakten
- Meldung von Feuermeldern / Gasmelder

Das System ermöglicht synchrone sowie asynchrone Funktionen. Die synchronen Funktionen liefern dem Benutzer auf Anfrage Informationen zu bestimmen Ressourcen als auch die Möglichkeit, diese zu steuern.

Die asynchronen Funktionen erlauben es dem Benutzer verschiedene Ressourcen zu abonnieren und somit eine Meldung zu erhalten sofern eine Zustandsänderung dieser Ressource eintritt.

4. Rollen

Das Smart Building System sieht 6 Typen von Benutzern vor. Das wäre der Chef, der Hausmeister, der Mitarbeiter,die Sicherheitsfirma, die Polizei und die Feuerwehr. Der Umfang der Zugriffsrechte durch das System auf ein Gebäude unterscheiden sich von Benutzer zu Benutzer. Für ein Bürogebäude würde der Chef und der Hausmeister vollen Zugriff haben, der Mitarbeiter, die Sicherheitsfirma, die Polizei und die Feuerwehr hingegen nur einen Teilzugriff.
Die Verteilung der Zugriffsrechte erfolgte aus rein logischer Sicht. Es würde zum Beispiel wenig Sinn machen der Sicherheitsfirma das Abrufen der Raumtemperatur zu ermöglichen.

Nachfolgend eine detaillierte Auflistung aller Rollen und den zugehörigen Funktionen.

REST (synchron)

Chef	-> Vollzugriff
Hausmeister	-> Vollzugriff
Mitarbeiter	-> Teilzugriff
Sicherheitsfirma	-> Teilzugriff

Chef

Abfrage der Energiedaten je Raum (aktueller Energieverbrauch)
Abfrage der Innentemperatur (IST) je Raum
Abfrage der Luftfeuchtigkeit je Raum
Abfrage der Verschattung je Raum
Abfrage der Lichtquellen je Raum
Abfrage der Steckdosen je Raum
Steuerung der Innentemperatur (SOLL) je Raum
Steuerung der Verschattung je Raum
Steuerung der Lichtquellen je Raum
Steuerung der Steckdosen je Raum

Hausmeister

Abfrage der Energiedaten je Raum (aktueller Energieverbrauch)
Abfrage der Innentemperatur (IST) je Raum
Abfrage der Luftfeuchtigkeit je Raum
Abfrage der Verschattung je Raum
Abfrage der Lichtquellen je Raum
Abfrage der Steckdosen je Raum
Steuerung der Innentemperatur (SOLL) je Raum
Steuerung der Verschattung je Raum
Steuerung der Lichtquellen je Raum
Steuerung der Steckdosen je Raum

Mitarbeiter

Abfrage der Energiedaten je Raum (aktueller Energieverbrauch)
Abfrage der Innentemperatur (IST) je Raum
Abfrage der Luftfeuchtigkeit je Raum
Abfrage der Verschattung je Raum
Abfrage der Lichtquellen je Raum
Abfrage der Steckdosen je Raum
Steuerung der Innentemperatur (SOLL) je Raum
Steuerung der Verschattung je Raum
Steuerung der Lichtquellen je Raum
Steuerung der Steckdosen je Raum

Sicherheitsfirma

Bewegungsmelder
Tür und Fensterkontakte

Publish / Subscribe (asynchron)

Chef	-> Vollzugriff
Hausmeister	-> Vollzugriff
Polizei	-> Teilzugriff
Feuerwehr	-> Teilzugriff
Sicherheitsfirma	-> Teilzugriff

Chef

- Bewegungsmelder
- Tür und Fensterkontakte
- Feuermelder

Hausmeister

-
- Bewegungsmelder
- Tür und Fensterkontakte
- Feuermelder

Polizei

- Bewegungsmelder
- Tür und Fensterkontakte

Sicherheitsfirma

- Bewegungsmelder
- Tür und Fensterkontakte

Feuerwehr

- Feuermelder

5. Meilensteine

5.1. Projektspezifisches XML Schema

XML-Schema

Das XML-Schema ist modular aufgebaut. Für jede Ressource wird ein Complex-Type verwendet, da diese unterschiedliche Informationen bzw. weitere Complex-Typen enthalten können.

Die gewählte Verschachtelung von Etagen & Räumen wird auch hier angewendet.
Die Listen werden über den Typennamen in der Mehrzahl dargestellt. z.B. etagen ist eine Liste von Etagen, welche vom Typ Etage sind.

Für die Elemente einer Liste muss zusätzlich noch eine ID definiert werden damit diese eindeutig zuordenbar sind. Bei der Wahl des Typs für die ID ergaben sich zwei Möglichkeiten: die ID als Integer oder als String Wert abspeichern. Der Vorteil einer numerischen ID ist der, dass das System die Verwaltung der ID's übernehmen kann und die Eingabe einer solchen ID nicht vom Benutzer abhängt., sondern automatisch erfolgt. Aus diesem Umstand heraus fiel die Wahl auf eine numerische ID.

Der Umstand dass in diesem System einzelne Ressourcen abgefragt werden können, erfordert diesen Modularen Aufbau ohne explizites Root-Element. Dieses Schema kann also XML-Dokumente mit unterschiedlichen Root-Elementen validieren.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="gebaeudeEl" type="gebaeude" />
<xsd:element name="etagenEl" type="etagen" />
<xsd:element name="etageEl" type="etage" />
<xsd:element name="raeumeEl" type="raeume" />
<xsd:element name="raumEl" type="raum" />
<xsd:element name="feuchtigkeitEl" type="feuchtigkeit" />
<xsd:element name="energieRaumEl" type="energieRaum" />
<xsd:element name="temperaturEl" type="temperatur" />
<xsd:element name="temperaturIstEl" type="temperaturIst" />
<xsd:element name="temperaturSollEl" type="temperaturSoll" />
<xsd:element name="lichterEl" type="lichter" />
<xsd:element name="lichtEl" type="licht" />
<xsd:element name="verschattungenEl" type="verschattungen" />
<xsd:element name="verschattungEl" type="verschattung" />
<xsd:element name="steckdosenEl" type="steckdosen" />
<xsd:element name="steckdoseEl" type="steckdose" />
<xsd:element name="kontakteEl" type="kontakte" />
<xsd:element name="kontaktEl" type="kontakt" />
<xsd:element name="bewegungenEl" type="bewegungen" />
<xsd:element name="bewegungEl" type="bewegung" />
<xsd:element name="feuermelderEl" type="feuermelder" />
<xsd:element name="feuermeldEl" type="feuermeld" />

<xsd:complexType name="gebaeude">
```

```
	<xsd:sequence>
		<xsd:element ref="etagenEl" minOccurs="0"/>
	</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="etagen">
	<xsd:sequence>
		<xsd:element ref="etageEl" minOccurs="0" maxOccurs="unbounded"/>
	</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="etage">
	<xsd:sequence>
		<xsd:element name="info" type="xsd:string" />
		<xsd:element ref="raeumeEl" minOccurs="0"/>
	</xsd:sequence>
	<xsd:attribute name="id" type="xsd:positiveInteger"/>
</xsd:complexType>

<xsd:complexType name="raeume">
	<xsd:sequence>
		<xsd:element ref="raumEl" minOccurs="0" maxOccurs="unbounded"/>
	</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="raum">
	<xsd:sequence>
		<xsd:element name="info" type="xsd:string" />
		<xsd:element ref="feuchtigkeitEl" minOccurs="0"/>
		<xsd:element ref="energieRaumEl" minOccurs="0"/>
		<xsd:element ref="temperaturEl" minOccurs="0"/>
		<xsd:element ref="lichterEl" minOccurs="0"/>
		<xsd:element ref="verschattungenEl" minOccurs="0"/>
		<xsd:element ref="steckdosenEl" minOccurs="0"/>
		<xsd:element ref="kontakteEl" minOccurs="0"/>
		<xsd:element ref="bewegungenEl" minOccurs="0"/>
		<xsd:element ref="feuermelderEl" minOccurs="0"/>
	</xsd:sequence>
	<xsd:attribute name="id" type="xsd:positiveInteger"/>
</xsd:complexType>

<xsd:complexType name="feuchtigkeit">
	<xsd:sequence>
		<xsd:element name="wert" type="xsd:decimal" />
		<xsd:element name="einheit" type="xsd:string" />
	</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="energieRaum">
	<xsd:sequence>
		<xsd:element name="wert" type="xsd:decimal" />
		<xsd:element name="einheit" type="xsd:string" />
	</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="temperatur">
	<xsd:sequence>
		<xsd:element ref="temperaturIstEl" minOccurs="0"/>
		<xsd:element ref="temperaturSollEl" minOccurs="0"/>
	</xsd:sequence>
</xsd:complexType>
```

```
<xsd:complexType name="temperaturIst">
	<xsd:sequence>
		<xsd:element name="wert" type="xsd:decimal" />
		<xsd:element name="einheit" type="xsd:string" />
	</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="temperaturSoll">
	<xsd:sequence>
		<xsd:element name="wert" type="xsd:decimal" />
		<xsd:element name="einheit" type="xsd:string" />
	</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="lichter">
	<xsd:sequence>
		<xsd:element ref="lichtEl" minOccurs="0" maxOccurs="unbounded"/>
	</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="licht">
	<xsd:sequence>
		<xsd:element name="info" type="xsd:string" />
		<xsd:element name="zustand" type="xsd:boolean" />
	</xsd:sequence>
	<xsd:attribute name="id" type="xsd:positiveInteger"/>
</xsd:complexType>

<xsd:complexType name="verschattungen">
	<xsd:sequence>
		<xsd:element ref="verschattungEl" minOccurs="0" maxOccurs="unbounded"/>
	</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="verschattung">
	<xsd:sequence>
		<xsd:element name="info" type="xsd:string" />
		<xsd:element name="wert" type="xsd:decimal" />
	</xsd:sequence>
	<xsd:attribute name="id" type="xsd:positiveInteger"/>
</xsd:complexType>

<xsd:complexType name="steckdosen">
	<xsd:sequence>
		<xsd:element ref="steckdoseEl" minOccurs="0" maxOccurs="unbounded"/>
	</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="steckdose">
	<xsd:sequence>
		<xsd:element name="info" type="xsd:string" />
		<xsd:element name="zustand" type="xsd:boolean" />
	</xsd:sequence>
	<xsd:attribute name="id" type="xsd:positiveInteger"/>
</xsd:complexType>

<xsd:complexType name="kontakte">
	<xsd:sequence>
		<xsd:element ref="kontaktEl" minOccurs="0" maxOccurs="unbounded"/>
	</xsd:sequence>
</xsd:complexType>
```

```
<xsd:complexType name="kontakt">
	<xsd:sequence>
		<xsd:element name="info" type="xsd:string" />
		<xsd:element name="zustand" type="xsd:boolean" />
	</xsd:sequence>
	<xsd:attribute name="id" type="xsd:positiveInteger"/>
</xsd:complexType>

<xsd:complexType name="bewegungen">
	<xsd:sequence>
		<xsd:element ref="bewegungEl" minOccurs="0" maxOccurs="unbounded"/>
	</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="bewegung">
	<xsd:sequence>
		<xsd:element name="info" type="xsd:string" />
		<xsd:element name="zustand" type="xsd:boolean" />
	</xsd:sequence>
	<xsd:attribute name="id" type="xsd:positiveInteger"/>
</xsd:complexType>

<xsd:complexType name="feuermelder">
	<xsd:sequence>
		<xsd:element ref="feuermeldEl" minOccurs="0" maxOccurs="unbounded"/>
	</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="feuermeld">
	<xsd:sequence>
		<xsd:element name="info" type="xsd:string" />
		<xsd:element name="zustand" type="xsd:boolean" />
	</xsd:sequence>
	<xsd:attribute name="id" type="xsd:positiveInteger"/>
</xsd:complexType>

</xsd:schema>
```

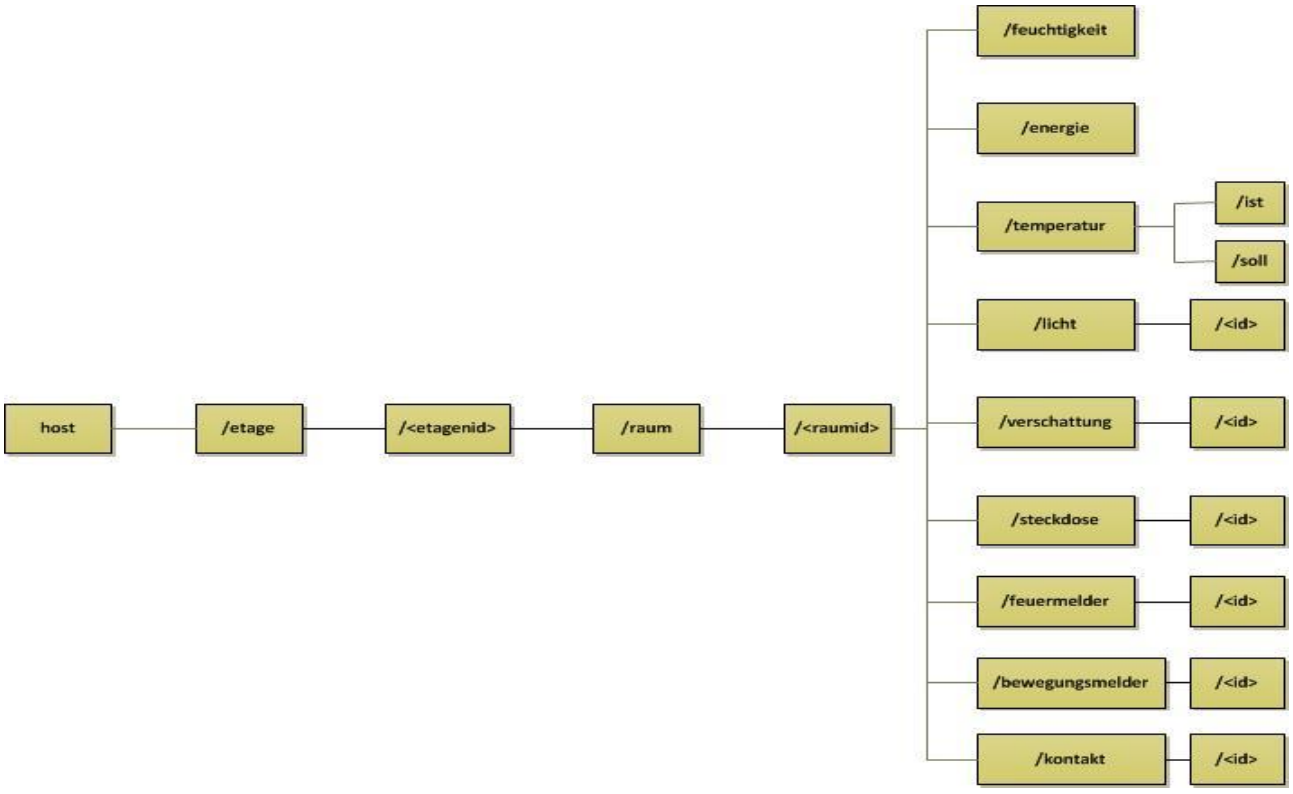
5.2. Ressourcen und die Semantik der HTTP-Operationen

REST-Hierarchie

Die entwickelte REST-Hierarchie und die verfügbaren Ressourcen wurden stark an den realen Aufbau eines Gebäudes angelehnt. Jedes Gebäude ist in Etagen unterteilt, die wiederrum in einzelne Räume aufgeteilt sind. Für jeden Raum können Sensoren & Aktoren hinzugefügt werden.
Daraus ergibt sich folgende Struktur

http://host:port/etage/<id>/raum/<id>/<funktion>/...

Die jeweiligen Etagen bzw. Räume und Funktionen (falls vorhanden) werden über die **id** identifiziert.
Wird eine Ressource ohne eine bestimmte **id** angesprochen z.B ***GET /etage*** wird eine **Liste** mit allen in der Ressource befindlichen Unterressourcen zurückgegeben, in diesem Fall eine Liste aller Etagen. Für Räume gilt das gleiche Prinzip ***GET /etage/1/raum*** holt eine Liste aller Räume auf Etage 1.
Ressourcen welche nicht über eine **id** verfügen, stellen somit keine Listen dar.
Beispiel ***GET /etage/1/raum/2/feuchtigkeit*** gibt einen Datensatz zurück.
Eine Besonderheit stellt die Unterteilung einer Temperatur-Ressource dar. Dort findet die beiden Ressourcen Soll & Ist. Dies war nötig, da die Temperatur auf einen bestimmten Soll-Wert eingestellt werden kann, der Ist-Wert ändert sich aber nur langsam. Um beide Zustände darstellen zu können wurden daraus 2 Ressourcen gebildet.
Der Soll-Wert kann z.B über ***PUT /etage/1/raum/1/temperatur/soll*** gesetzt werden.



HTTP-Operationen

Damit die Interaktion mit einem RESTful Webservice gewährleistet werden kann sind verschiedene HTTP-Operationen nötig die auf die Ressourcen angewendet werden können.

Es wurde sich auf die HTTP-Operationen GET, POST, PUT und DELETE begrenzt.

GET - Mit GET fordert man eine angebene Ressource vom Server an. Es werden dabei keine Änderungen vorgenommen, deshalb wird GET auch als „sicher“ bezeichnet.

POST - Mit POST kann man eine (Sub-)Ressource unterhalb einer angebenen Ressource eingefügen. Die neue Ressource besitzt noch keine URI, deswegen muss die übergeordnete Ressource angesprochen werden.

PUT - Wird genutzt um eine angegebene, bereits vorhandene Ressourcen zuverändern.

DELETE - Mit DELETE kann eine angegebene Ressource gelöscht werden.

	GET	PUT	POST	DELETE
/etage	x		x	x
/etage/<id>	x			x
/etage/<id>/raum	x		x	x
/etage/<id>/raum/<id>	x			x
/etage/<id>/raum/<id>/feuchtigkeit	x	x	x	x
/etage/<id>/raum/<id>/energie	x	x	x	x
/etage/<id>/raum/<id>/temperatur	x		x	x
/etage/<id>/raum/<id>/temperatur/soll	x	x		x
/etage/<id>/raum/<id>/temperatur/ist	x			x
/etage/<id>/raum/<id>/licht	x		x	x
/etage/<id>/raum/<id>/licht/<id>	x	x		x
/etage/<id>/raum/<id>/verschattung	x		x	x
/etage/<id>/raum/<id>/verschattung/<id>	x	x		x
/etage/<id>/raum/<id>/steckdose	x		x	x
/etage/<id>/raum/<id>/steckdose/<id>	x	x		x
/etage/<id>/raum/<id>/feuermelder	x		x	x
/etage/<id>/raum/<id>/feuermelder/<id>	x	x		x

Das Projekt teilt sich in zwei Hauptakteure auf:

Benutzer - Physischer Benutzer des Systems, hat Zugriff auf Ressourcen (GET, POST, PUT, DELETE)

Sensoren - Virtuelle Sensoren verändern die Werte der Ressourcen unabhängig vom Benutzer. (PUT)

Beispiel: Die Feuchtigkeit eines Raums kann nicht vom Benutzer gesteuert werden. Dies würde den Endschluss nach sich ziehen eben keine PUT-Operation für diese Ressource zu definieren. Jedoch der Umstand dass ein Feuchtigkeitssensor neue Werte liefert, verlangt eine PUT-Operation mit welcher der Sensor den Wert aktualisieren kann.

Deshalb wurden für die Ressourcen die nicht vom Benutzer verändert werden können, trotzdem PUT-Operationen definiert, welche dann von den virtuellen Sensoren genutzt werden um ihre Werte zu aktualisieren.

Nachfolgend eine Auflistung der PUT-Operationen aufgeteilt nach Akteuren

Benutzer	**Sensoren**
Temperatur (SOLL) Licht Verschattung Steckdose	Feuchtigkeit Energie Temperatur (IST) Licht Feuermelder Tür- & Fensterkontakte Bewegungsmelder

5.3. RESTful Webservice

Der RESTful Webservice wird in Java mit Hilfe von verschiedenen API's realisiert.
Folgende API's wurden verwendet:

- ***JAXB - Java Architecture for XML Binding***
  Eine API die es ermöglicht Daten aus einer XML-Schema-Instanz heraus automatisch an Java-Klassen zu binden, und diese Java-Klassen aus einem XML-Schema heraus zu generieren. Dabei sind zwei Begriffe von großer Bedeutung:
  - Marshalling
    Generiert aus einem Baum von Java-Objekten ein XML-Dokument
  - Unmarshalling
    Generiert aus einem XML-Dokument einen Baum von Java-Objekten
- ***Jersey Framework***
  Auch noch JAX-RS (Java API for RESTful Web Services) genannt.
  Eine API für Java welche die Verwendung der REST-Architektur ihm Rahmen von Webservices ermöglicht.
- ***Grizzly Server***
  Java basierter HTTP-Server welcher für die Realisierung des RESTful-Webservices verwendet wird.

Aufbau

Die Kommunikation zwischen Client und dem Server läuft über die HTTP-Operationen ab.
Es ist erstmal wichtig alle möglichen HTTP-Operationen in einer Klasse zu implementieren. Über diese Klasse, welche als Schnittstelle fungiert, kann der Client dann die auf dem Server hinterlegten Ressourcen abfragen.

Client <-> Request Handler <-> Daten Handler

Es gibt somit eine eindeutige Trennung zwischen den Ressourcen und deren Operationen.

Der Datenhandler kümmert sich dann darum, die vom Requesthandler erhaltenen Anfragen in sinnvolle Operationen auf den Datensätzen umzusetzen und ihm dann eine entsprechende Antwort zurück zu geben.
Der Requesthandler verarbeitet diese Antwort und schickt einen HTTP-Statuscode mit ggf. den angeforderten Daten an den Client zurück.

Statuscode	Bedeutung	Beschreibung
200	OK	Rückgabe einer Ressource bzw. der Daten
201	Created	Ressource wurde angelegt und Location wird zurück gegeben
204	No content	Ressource wurde gelöscht oder geändert. Keine Daten werden zurück gegeben
404	Not found	Ressource konnte nicht gefunden werden bzw. fehlgeschlagene Operation

src/de/fhkoeln/gm/wba2/phase2/rest/resource/GebaeudeRessource.java

Diese Klasse fungiert als RequestHandler.

Anhand eines Codeauschnitts soll die Funktion einer GET-Anfrage erklärt werden:

```
@GET
@Path("/etage/{etagenid}")
@Produces(MediaType.APPLICATION_XML)
public Response getEtage(@PathParam("etagenid") BigInteger etagen_id) {
	String etage = dh.getEtage(etagen_id);
	if (etage != null)
		return Response.ok().entity(etage).type(MediaType.APPLICATION_XML)
				.build();
	else
		return Response.status(404).build();
}
```

`@GET`
Die erste Zeile stellt eine typische javax-Annotation dar. Sie sorgt dafür das die angegebene HTTP-Operation an die richtige Methode weitergeleitet wird.

`@Path("/etage/{etagenid}")`
Über diese Zeile wird die Ressource angegeben. Die geschweiften Klammern geben an dass es sich hier um eine Variable handelt, also eine bestimmte Ressouce die über diese Variable angesprochen wird.

`@Produces(MediaType.APPLICATION_XML)`
Legt den Rückgabe-Typ der Operation fest. Da in unserem Fall mit XML gearbeitet wird wird der Mime-Type auf APPLICATION_XML gesetzt.

`public Response getEtage(@PathParam("etagenid") BigInteger etagen_id)`
Die eigentliche Methoden-Kopf. Als Parameter werden hier die Variablen aus dem Ressource-Path angegeben um sie innerhalb der Methode nutzen zu können.

`String etage = dh.getEtage(etagen_id);`
Nun wird über den Datenhandler die benötigte Methode mit den gegebenen Parametern aufgerufen, die Antwort ist in dem Fall ein XML-String für die gewünschte Etage.

```
if (etage != null)
	return Response.ok().entity(etage).type(MediaType.APPLICATION_XML).build();
else
	return Response.status(404).build();
```

Diese Abfrage überprüft ob ein Datensatz vom Datenhandler abgekommen, oder die Anfrage fehlgeschlagen ist.
Ist die Anfrage erfolgreich gewesen dann gibt der Server einen Statuscode 200 (OK) mit den entsprechenden Daten als Entität zurück.
Im Fehlerfall wird der Statuscode 404 (Not found) zurück gegeben, und der Client kann somit auf den Fehler reagieren.

src/de/fhkoeln/gm/wba2/phase2/rest/jaxb/DataHandler.java

Die Klasse fungiert als Datenhandler und enthält alle nötigen Methoden zur Datenmanipulation.

```
public String getEtage(BigInteger id) {
	return marshall(getEtageObj(id));
}
```

Diese Methode wird von dem RequestHandler aufgerufen (basierend auf dem vorherigen Beispiel). Sie setzt ein Java-Objekt mit Hilfe des Marshallers in einen XML-String, basierend auf dem XML-Schema, um und liefert es an den Requesthandler zurück.
Um das gewünschte Java-Objekt zu finden wird die folgende Methode benutzt.

```
private Etage getEtageObj(BigInteger etagen_id) {
	Etage etageObj = null;
	List<Etage> etagen_list = rootEl.getEtagenEl().getEtageEl();

	for (Etage curr_etage : etagen_list) {
		if (curr_etage.getId().equals(etagen_id)) {
			etageObj = curr_etage;
			break;
		}
	}
	return etageObj;
}
```

Diese Hilfsmethode sucht in der Liste von Etagen nach der gewünschten Etage und liefert, falls gefunden, das entsprechende Objekt zurück.

5.4. Konzeption + XMPP Server einrichten

Konzeption

Publish-Subscribe

Publish-Subscribe ist ein Paradigma welches einen entkoppelten Nachrichtenaustausch zwischen Sender und Empfänger ermöglicht. Die Kommunikationsteilnehmer müssen sich dabei nicht kennen. Die Teilnehmer teilen sich in Publisher (Anbieter) und Subscriber (Abonnent) auf. Der Publisher kann Nachrichten für andere Teilnehmer veröffentlichen und der Abonnent erhält diese, sofern er den Leaftopic des Publishers abonniert hat.

Leaftopics

Die Leaftopics sind die verschiedenen Kategorien in welchen Nachrichten von Publishern veröffentlicht werden können.
Leaftopics können also von Subscribern abonniert werden um darin veröffentlichte Nachrichten zu erhalten.

Für unser Projekt wurden 3 Leaftopics bzw Leafnodes definiert:

- Feuermelder
- Bewegungsmelder
- Tür- & Fensterkontakte

Die Funktionsweise dieser Elemente, also das Benachrichtigen bei einem eintreffenden Event was als Auslöser fungiert, spricht dafür diese als Leaftopics zudefinieren.

Publisher

Die Feuermelder, Bewegungsmelder & Tür-/Fensterkontakte werden in dem System als Publisher angesehn. Sie veröffentlichen eine Nachricht bei einer Zustandsänderung. z.B bei einem Einbruch wird ein Kontakt ausgelöst der dann eine Nachricht absetzt.

Subscriber

Die Subscriber sind in dem Falle die eigentlichen Benutzer des Systems mit ihren verschiedenen Rollen. Sie können verschiedene Topics abonnieren und so über etwaige Zustandsänderungen informiert werden.

Datenübertragung

Bei Publish-Subscribe gibt es zwei verschiedene Formen von Benachrichtigungen:

- Light Ping
- Fat Ping

Light Ping: Light Ping Benachrichtigungen enthalten keine Nutzdaten an sich, sondern informieren nur darüber dass neue Daten zur Verfügung stehen.

Fat Ping: Fat Ping Benachrichtigungen enthalten einen Nutzdatenteil. Dies kann aber negative Auswirkungen auf Bandbreite, Latenz sowie CPU-Auslastung führen.

Für unser System ist das Fat Ping Verfahren am sinnvollsten, da der Benutzer somit sofort weiß welcher Sensor ausgelöst hat und dementsprechend reagieren kann. Hinzu kommt die Annahme dass die Benachrichtigungen in großen Intervallen abgestetzt werden, die oben beschriebenen Auswirkungen also nicht relevant sind.

XMPP Server

Der im Projekt benutzte Server zur Realisierung des Benachrichtungssystems ist ein XMPP (Extensible Messaging and Presence Protocol) (*RFC 6120–6122 sowie 3922, 3923*) Server.
Es handelt sich hierbei um den Openfire Server von IgniteRealtime welcher frei verfügbar ist.

Das XMPP Protokoll benutzt XML als Übertragungsformat und bietet verschiedene Dienste, unter anderem Benachrichtigungssysteme.

Um den Zugriff auf den Server zu ermöglichen wird die Smack(x)-API benutzt. Sie stellt alle Funktionen zum Realisieren eines Messaging-Systems zur Verfügung.

Zur Verdeutlichung des Smack-Funktionen in Verbindung mit dem Openfire-Server wird ein Codebeispiel aufgeführt:

src/de/fhkoeln/gm/wba2/phase2/xmpp/connection/ConnectionHandler.java

```
public boolean connect(String hostname, int port){

	if (xmppConn != null && xmppConn.isConnected()){
		return true;
	}
	ConnectionConfiguration connConfig = new ConnectionConfiguration(hostname, port);
	xmppConn = new XMPPConnection(connConfig);
	accMngr = new AccountManager(xmppConn);

	try{
		xmppConn.connect();
		pubSubMngr = new PubSubManager(xmppConn, "pubsub." + xmppConn.getHost());
	} catch(XMPPException e){
		System.out.println(e);
		return false;
	}

	this.hostname = hostname;

	return true;
}
```

Diese Methode verbindet sich mit dem Openfire-Server. Dafür werden der Host und der Port als Parameter übergeben.
Die Variable **xmppConn** ist vom Typ XMPPConnection und dient dazu die Verbindung zu verwalten, und Informationen über diese ab zu fragen.
Es wird eine Konfiguration angelegt mit den entsprechenden Host-Daten. Diese ist vom Typ ConnectionConfiguration.
Die XMPP-Verbindung wird dann mit Hilfe der Konfiguration konfiguriert.
Danach wird versucht mit Hilfe von **xmppConn.connect()** eine Verbindung zum Server aufzubauen.
Wenn dies erfolgreich war kann ein neuer PubSub-Manager angelegt werden.
Dieser greift auf den angebotenen PubSub-Dienst des Servers zu. Der Dienst ist über **pubsub.<hostname>** erreichbar.
Über den Manager ist es möglich neue Leafnodes zu erstellen, Nachrichten zu veröffentlichen usw. Er ist so zu sagen das Herzstück der XMPP-Anwendung.

Das Erstellen einer Node soll ebenfalls anhand eines Beispiel erläutert werden:

```
// Create the node
LeafNode leaf = pubSubMngr.createNode("testNode");

ConfigureForm form = new ConfigureForm(FormType.submit);

form.setPublishModel(PublishModel.open);
form.setAccessModel(AccessModel.open);
form.setDeliverPayloads(true);
form.setPersistentItems(true);

leaf.sendConfigurationForm(form);
```

Zuerst wird über den PubSub-Manager ein Leafnode erzeugt. Dieser bekommt den Namen „testNode“.
Es muss beachtet werden dass ein Node konfiguriert werden muss damit er die gewünschten Eigenschaften erhält.
Die Konfiguration erstellt man über die Klasse ConfigureForm und kann dem Node später zugewiesen werden.
Wichtig sind dabei die Methoden **setPublishModel()** & **setAccessModel().**
Mit **PublishModel.open** wird angegeben dass jeder über diesen Node Nachrichten veröffentlichen kann.
Mit **AccessModel.open** wird angegeben das jeder Zugriff auf den Node hat.
Mit **setDeliveryPayloads()** wird festgelegt ob Nutzdaten übertragen werden können.
Mit **setPersistentItems()** wird festgelegt ob alte Nachrichten im Node gespeichert werden oder nicht.

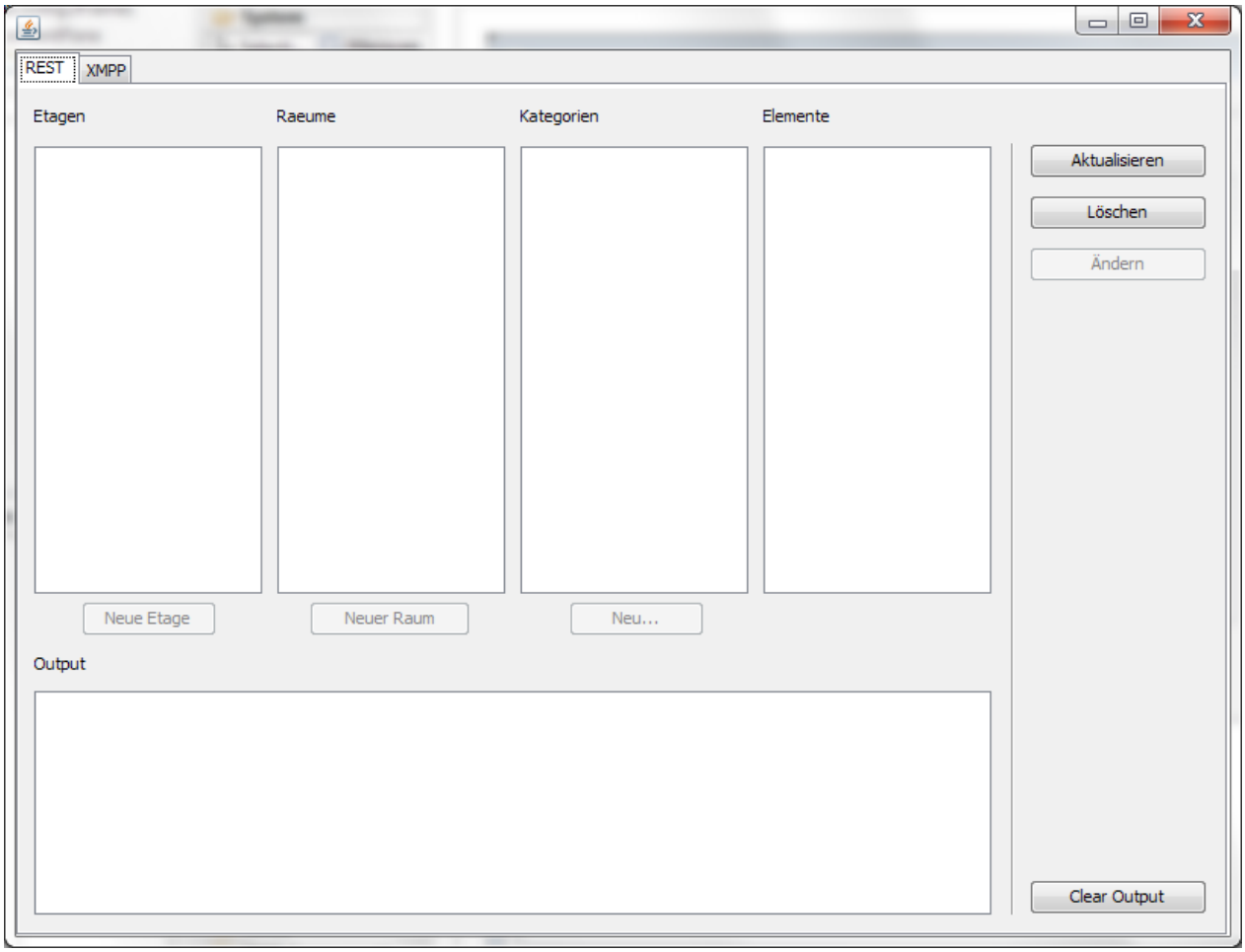
5.5. Client

Der Client ist das Interface über welches ein Benutzer die verschiedenen Funktionen abfragen und steuern kann. Zusätzlich kann er über den Client verschiedene Sensor-Kategorien abonnieren.
Dem Benutzer wird mit Client ermöglicht sein eigenes System zu verwirklichen. Er kann also selber neue Etagen, Räume und Sensoren / Aktoren in sein System einfügen und es so dynamisch verändern und benutzen.

Aufbau

Der Client ist vom Aufbau her sehr simpel. Zuerst erscheint ein kleines Startfenster in welchem der Benutzer sich im System einloggen kann.
Nach erfolgreicher Anmeldung kommt er sofort in das Hauptfenster. Dort findet er 2 verschiedene Tabs, REST und XMPP.

REST-Tab

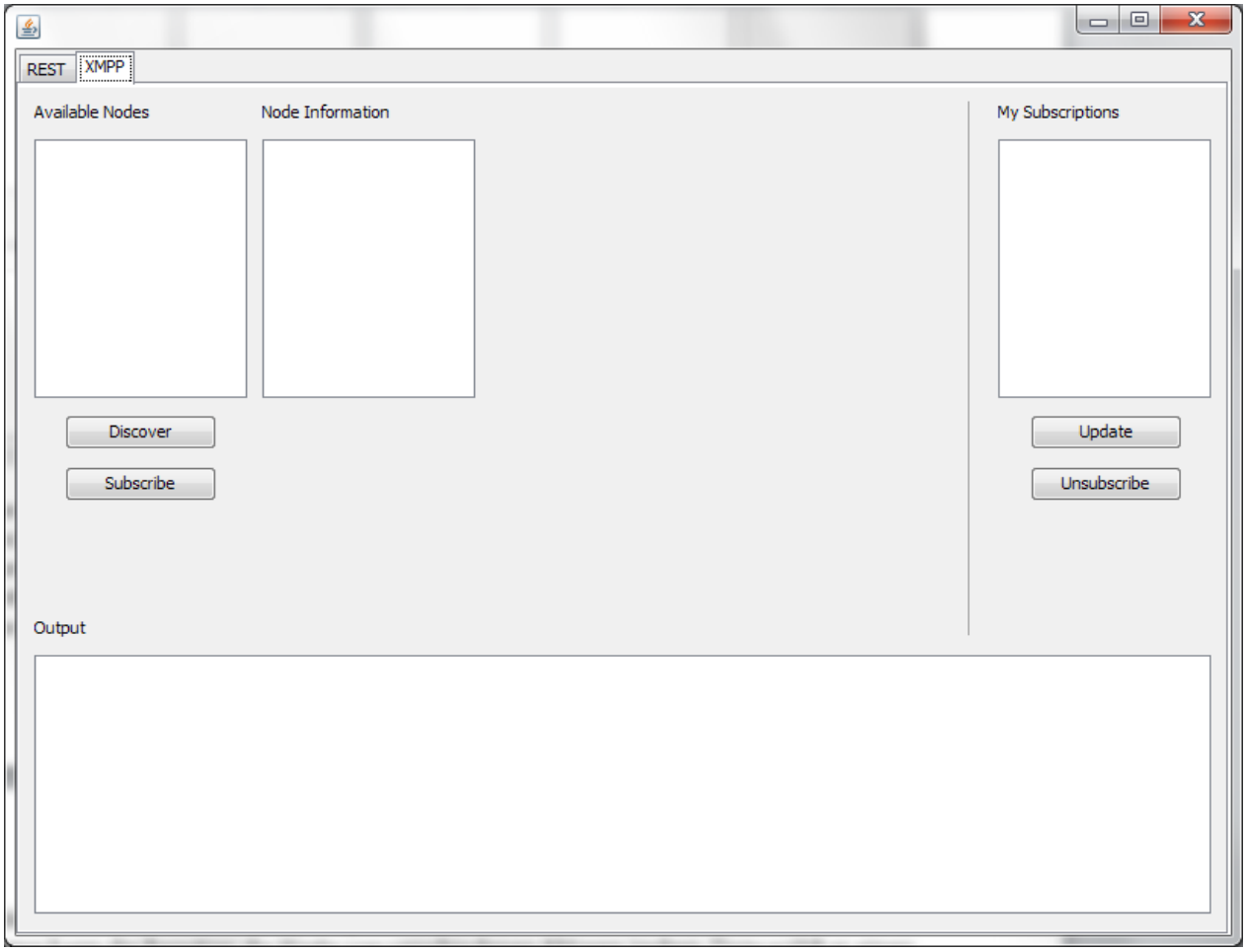


Über die Schaltfläche „Aktualisieren“ kann die Übersicht aller Etagen angezeigt werden.
Durch einen Klick auf eine entsprechende Etage werden, falls vorhanden die darunterliegenden Räume angezeigt.
Es ist die Möglichkeit gegeben neue Etagen / Räume und Elemente anzulegen.
Würde man gerne einen Eintrag löschen, genügt es diesen Eintrag auszuwählen und über einen

Klick auf „Löschen“ wird dieser Eintrag entfernt.
Ebenso kann der Benutzer die Werte von verschiedenen Aktoren ändern. Dazu wählt er einen entsprechenden Aktor aus und klickt auf die Schaltfläche „Ändern“. Es erscheint ein Dialogfenster in welchem der neue Wert gesetzt werden kann.

Da es keine reellen Sensoren gibt, kann der Benutzer den Zustand der Elemente „Feuermelder“, „Bewegungsmelder“ & „Tür- & Fensterkontakte“ manuell verändern, um so das Publishen einer Nachricht auszulösen. Diese wird dann unter dem Tab XMPP angezeigt.

XMPP-Tab



Über den XMPP-Tab hat der Benutzer zugriff auf seine Abonnements, bzw die verfügbaren Topics welche er abonnieren kann.
Mit einem Klick auf „Discover“ werden alle dem Server bekannten Topics angezeigt. Ein Klick auf einen Topic zeigt die Node Informationen an.
Man kann diesen Node nun subscriben über die Schaltfläche „Subscribe“.
Der Benutzer kann sich auch alle abonnierten Topics anzeigen lassen in dem er auf „Update“ drückt. Würde er gerne einen Topic „unsubscriben“ so wählt kann er den Topic in seiner Liste auswählen und dann über einen Klick auf „Unsubscribe“ bestätigen.

Wird vom Publisher eine Nachricht veröffentlicht, so erscheint diese in dem Output Feld.

6. Inbetriebnahme

Zu aller erst sollte der Openfire-Server gestartet werden.
Es muss ein Benutzer **user1** mit dem Passwort **login** angelegt werden, sowie ein Benutzer **server** mit dem Passwort **login**.

Danach sollte dann der REST-Server gestartet werden. Die Klasse **RESTServer.java** ist zu finden im Package **de.fhkoeln.gm.wba2.phase2.rest.server**.

Schlussendlich kann der Client ausgeführt werden. Dafür die Klasse **SmartBuildingClient.java** im Package **de.fhkoeln.gm.wba2.phase2.client** ausführen.

7. Literatur

http://opus.bibl.fh-koeln.de/volltexte/2012/380/pdf/WPF_Krumnow_v2.pdf
Benjamin Krumnow. Ereignisgesteuerte Systeme im Web, Juli 2012

http://xmpp.org/extensions/xep-0060.html
Publish-Subscribe Standard Spezifikation

http://www.cs.ru.nl/~pieter/oss/manyfaces.pdf
The Many Faces of Publish/Subscribe

http://www.igniterealtime.org/builds/smack/docs/latest/documentation
Smack(x) API - Dokumentation