

StudMap

Indoor Navigation

Projektdokumentation

im Fach Fortgeschrittene Internetanwendungen



Westfälische Hochschule

Gelsenkirchen Bocholt Recklinghausen

vorgelegt von: Thomas Buning, Marcus Büscher,
Daniel Hardes, Christoph Inhestern,
Dennis Miller, Fabian Paus,
Christian Schlütter

Studienbereich: Informationstechnik

Gutachter: Prof. Dr. Martin Schulten

Abgabetermin: 21.01.2014

Inhaltsverzeichnis

1 Domänenmodell	1
1.1 Architektur	1
1.1.1 Webservice (StudMap.Service)	1
1.1.2 Admin-Oberfläche (StudMap.Admin)	1
1.1.3 Navigations-Client (StudMap.Navigator)	1
1.1.4 Sammler-Client (StudMap.Collector)	1
1.2 Benutzerrollen	1
1.2.1 Anwender (User)	1
1.2.2 Administrator	2
1.3 Begriffe	2
1.3.1 Karte (Map)	2
1.3.2 Stockwerk (Floor)	2
1.3.3 Schicht (Layer)	2
1.3.4 Route	2
1.3.5 Graph	2
1.3.6 Knoten (Node)	3
1.3.7 Kante (Edge)	3
1.3.8 Point of Interest (POI)	3
2 Recherche	4
2.1 QR-Codes	4
2.1.1 Serverseitig	4
2.1.2 Clientseitig	4
2.2 NFC-Tags	5
2.2.1 Umsetzung	5
2.3 OCR der Raumschilder	5
2.3.1 Bibliothek tesseract	6
2.3.2 App Google Goggles	7
2.4 WLAN Fingerprinting	7
2.4.1 Sammeln von WLAN Fingerprints	8
2.4.2 Kalibrierung	8
2.4.3 Positionierung mittels WLAN Fingerprints	8
3 Positionsermittlung	10
3.1 QR-Tags	10



4	Benutzerverwaltung	11
5	Webservice	14
5.1	Allgemeine Objekte	14
5.1.1	Edge	14
5.1.2	Node	15
5.1.3	Graph	15
5.1.4	Pathplot	15
5.1.5	FloorPlanData	16
5.1.6	Room	16
5.1.7	PoiType	16
5.1.8	PoI	17
5.1.9	NodeInformation	17
5.1.10	Floor	18
5.1.11	Map	18
5.1.12	User	18
5.1.13	FloorGraph	19
5.1.14	NodeInformationWithId	19
5.2	Rückgabe Objekte	19
5.2.1	BaseResponse	19
5.2.1.1	ResponseStatus	20
5.2.1.2	ResponseError	20
5.2.2	ObjectResponse	21
5.2.3	ListResponse	21
5.3	MapsController	21
5.3.1	CreateMap	21
5.3.2	DeleteMap	21
5.3.3	GetMaps	22
5.3.4	SaveGraphForFloor	22
5.3.5	DeleteGraphForFloor	22
5.3.6	GetGraphForFloor	22
5.3.7	GetNodeInformationForNode	23
5.3.8	GetFloorPlanData	23
5.3.9	SaveNodeInformation	23
5.3.10	GetRouteBetween	23
5.3.11	GetPoiTypes	24
5.3.12	GetPoIsForMap	24
5.3.13	GetRoomsForMap	24
5.3.14	GetNodeForNFC	24
5.3.15	GetNodeForQRCode	25



5.4	UserController	25
5.4.1	Register	25
5.4.2	Login	25
5.4.3	Logout	26
5.4.4	GetActiveUsers	26
6	Verwendung des Webservices	27
6.1	Verwendung der Benutzerschnittstelle	27
6.1.1	Registrierung	27
6.1.2	Aktive und inaktive Benutzer	27
6.1.3	Aktive Benutzer abfragen	27



1 Domänenmodell

Durch das Domänenmodell legen wir Begriffe fest, mit denen die Kommunikation im Projektteam einfacher wird.

1.1 Architektur

1.1.1 Webservice (StudMap.Service)

Stellt Funktionen zur Ablage und Abfrage von Navigationsinformationen und Benutzerdaten öffentlich bereit.

1.1.2 Admin-Oberfläche (StudMap.Admin)

Weboberfläche zum Anlegen, Bearbeiten von Navigationsinformationen und Benutzerdaten. Die Weboberfläche kann nur von der Benutzerrolle Administrator bedient werden.

1.1.3 Navigations-Client (StudMap.Navigator)

App zur Anzeige von Karten und Navigation zwischen Wegpunkten. Wird bedient durch den Anwender.

1.1.4 Sammler-Client (StudMap.Collector)

App zur Eingabe von Navigationsinformationen. Wird bedient durch Administratoren.

1.2 Benutzerrollen

1.2.1 Anwender (User)

Muss identifiziert sein und verwendet den Navigations-Client.



1.2.2 Administrator

Verwendet Admin-Oberfläche und den Sammler-Client. Muss registriert sein.

1.3 Begriffe

1.3.1 Karte (Map)

Beschreibt das gesamte Gebäude mit allen Stockwerken.

1.3.2 Stockwerk (Floor)

2-dimensionale Ansicht mit allen Layern der Ebene.

1.3.3 Schicht (Layer)

Es gibt mehrere Schichten, die jeweils Detailinformationen zu einem Stockwerk enthalten.

- Bild-Layer: Enthält grafische Darstellung des Stockwerks.
- Graph-Layer: Enthält Kanten und Knoten für Routen.
- POI-Layer: Zusatzinformationen zu speziellen Orten.
- Routen-Layer: Darstellung grafischer Elemente zur Navigation.
- Personen-Layer: Darstellung anderer Anwender.

1.3.4 Route

Hat einen Start- und einen Endknoten. Verbindet diese beiden Knoten über Zwischenknoten und Kanten.

1.3.5 Graph

Gesamtheit aller Knoten und Kanten der Karte (Stockwerk-übergreifend).



1.3.6 Knoten (Node)

Besteht aus eindeutigem Identifier, X- und Y-Koordinate und Stockwerk. Zu dem Knoten können zusätzliche Informationen hinterlegt werden: Name, Raumnummer, NFC-Tag, QR-Tag, ..., Verweis auf POI.

1.3.7 Kante (Edge)

Verbindung zweier Knoten. Bedeutet, dass man von einem Punkt zum anderen laufen kann.

1.3.8 Point of Interest (POI)

Ort besonderen Interesses (z.B. Bibliothek, Mensa, ...)

2 Recherche

Für die Navigation innerhalb von Gebäuden konnten wir nicht auf die gängigen Standards zurückgreifen, sondern mussten uns andere Wege überlegen, wie wir die Nutzer innerhalb der Gebäude lokalisieren.

Außerhalb von Gebäuden ist die Lokalisierung mittels GPS sehr verbreitet und auch einfach und akkurat. Um eine ähnliche Lokalisierung unserer Nutzer zu ermöglichen haben wir uns die folgenden Möglichkeiten überlegt.

2.1 QR-Codes

QR-Codes sind weit verbreitet, einfach zu erstellen und mit vielen Geräten einzulesen. Dadurch bieten QR-Codes die Möglichkeit Informationen einfach an Orten anzubringen und von Maschinen einzulesen.

Wir haben uns zwei Konzepte überlegt, QR-Codes in unserem Projekt einzubauen. Dazu haben wir einmal die Möglichkeit betrachtet die Auswertung des QR-Codes am Server zu realisieren und mit der Möglichkeit verglichen die Auswertung direkt am Client des Nutzers zu implementieren.

2.1.1 Serverseitig

Für die serverseitige Umsetzung hat gesprochen, dass das Smartphone keine Rechenleistung benötigt um die QR-Codes zu dekodieren. Des Weiteren wird durch eine serverseitige Implementierung vermieden, dass der Nutzer weitere Apps auf seinem Smartphone installieren muss.

Für die Implementierung in den Webservice haben wir uns für die offene Bibliothek [ZXing.NET](https://zxing.net/) benutzt. Allerdings ist uns bei der Implementierung und Testen der Bibliothek direkt aufgefallen, dass die Bilder zuvor am Smartphone verkleinert werden müssen um Bandbreite zu sparen und die Laufzeit der Bibliothek zu verringern. Dadurch wird, obwohl es ein Ziel der serverseitigen Umsetzung war, Rechenleistung benötigt. Darüber hinaus mussten wir feststellen, dass die Bibliothek keine zuverlässige Dekodierung der QR-Codes bietet.

2.1.2 Clientseitig

Im Gegensatz zur serverseitigen Umsetzung wird bei dieser Implementierung die gesamte Dekodierung am Client vorgenommen. Dadurch wird die Netzlast verringert,

der Aufwand am Client aber erhöht.

Hier bot sich zum einen an eine Bibliothek in den Client aufzunehmen, oder eine externe Anwendung zum Dekodieren der QR-Codes zu benutzen. Wir haben uns schlussendlich dazu entschieden

Entscheidung
beim QR-
Code
Reader
aufschrei-
ben und
warum

2.2 NFC-Tags

NFC ist eine Technologie, auf die wir im Alltag immer häufiger stoßen, sei es bei Werbung, Bezahl- oder Ticketsystemen. NFC steht für Near Field Communication und besteht aus zwei Komponenten, der passiven, dem NFC-Tag, und der aktiven Komponente, beispielsweise dem Smartphone.

Die als Datenspeicher fungierenden NFC-Tags werden immer preiswerter und sind zudem relativ klein, was eine Anbringung an den gewünschten Orten problemlos ermöglicht. Auf der anderen, der aktiven Seite stehen immer mehr Smartphones bereit, die diese Technologie unterstützen.

Die steigende Beliebtheit der NFC-Technologie verdankt diese dem Komfort. Wie der Name bereits aussagt, reicht schon die Nähe der aktiven Komponente zur passiven um Daten zu kommunizieren. Diesen Komfort bieten wir dem Benutzer, um seine Position dem Navigator mitzuteilen.

Des Weiteren lassen sich auf dem NFC-Tag zusätzliche Informationen hinterlegen, die Unwissende auf die Navigationsmöglichkeit aufmerksam machen.

2.2.1 Umsetzung

Der gesamte Prozess der Positionsermittlung findet clientseitig statt. Zum einen ist das Client-Gerät unumgänglich für die Kommunikation mit dem NFC-Tag und zum anderen sind die Datenmengen und der Aufwand der Interpretation sehr gering.

Die Android NFC API ermöglicht uns die native Umsetzung der Positionsermittlung für Android-Geräte.

2.3 OCR der Raumschilder

Anstatt QR- oder NFC-Tags an den Räumen der FH anzubringen, besteht die Möglichkeit die bereits angebrachten Türschilder zu verwenden. Hierzu ist eine Erkennung der Raumnummer auf dem Türschild notwendig. Die OCR (Optical Character Recognition) kann über eine Bibliothek sowohl auf Client-, als auch auf Serverseite erfolgen. Erfolgt die Erkennung auf dem Client, dann könnten ebenfalls bestehende Apps zur Texterkennung verwendet werden.



Abbildung 2.1: Gesamtes Bild Raumschild

2.3.1 Bibliothek tesseract

Tesseract¹ ist eine native Bibliothek für die Erkennung von Text in Bilddateien. Es gibt sowohl für .NET als auch für Java (Android) entsprechende Wrapper, die von uns verwendet werden können. Bei den Tests zu der OCR-Bibliothek haben sich allerdings einige Schwächen gezeigt. Tesseract ist für die Texterkennung von gescannten Dokumenten gedacht und arbeitet deshalb nur stabil, wenn sich auf dem Bild ausschließlich Text befindet. Dies wird an folgenden Beispielbildern deutlich.

Auf dem Bild mit Raumschild und Wand wird nur unzuverlässig Text erkannt (siehe 2.1). Die Ergebnisse variieren von "Kein Text erkannt" bis hin zu "Buchstabensalat mit Raumnummer". Schneidet man den relevanten Teil per Hand aus (siehe 2.2), wird der Text einwandfrei erkannt.

Damit die Raumschilder zuverlässig erkannt werden, ist es notwendig, dass aufgenommene Bilder zunächst auf den Bereich mit der Raumnummer zugeschnitten werden. Dies erfordert einen hohen Entwicklungsaufwand.

¹<https://code.google.com/p/tesseract-ocr/>

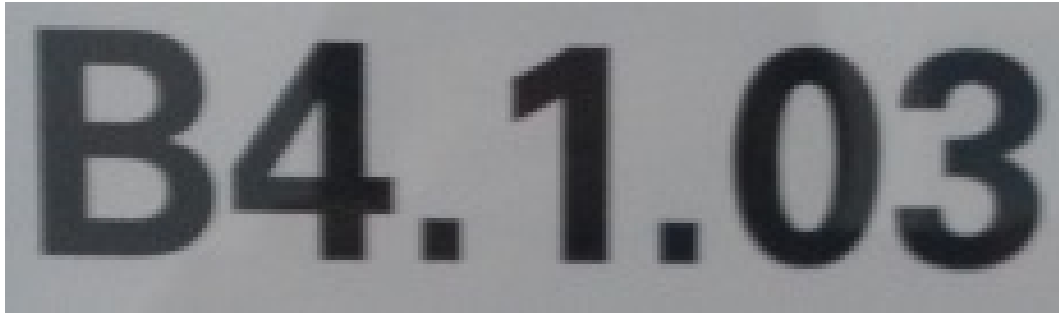


Abbildung 2.2: Ausschnitt Raumschild

2.3.2 App Google Goggles

Google Goggles² ist eine Android-App, die zur Erkennung von Text, Symbolen und QR-Tags verwendet werden kann. Diese App lieferte in Tests auch bei suboptimalen Bildern gute Ergebnisse. Außerdem ist die App gut in das Android-Umfeld eingebettet und lässt sich leicht bedienen.

Allerdings gibt es für die Verwendung von Google Goggles noch keine öffentliche API³. Diese ist zwar von Google geplant, aber nie umgesetzt worden. Auch wenn die App eine komfortable Möglichkeit zur OCR-Erkennung bietet, kann diese ohne API nicht in unserem Projekt verwendet werden.

2.4 WLAN Fingerprinting

Bei der Positionierung des Nutzers mittels WLAN haben wir über eine für den Nutzer passive Positionierung recherchiert. Alle anderen Positionierungsmethoden benötigten eine Eingabe des Nutzers. Wir haben hier die Eingabe der Position auf einer Karte und das Einlesen von QR-Codes oder NFC-Tags behandelt. Die Positionierung ermöglicht es allerdings im Hintergrund zu laufen und ohne Eingabe des Nutzers die Position zu bestimmen.

WLAN ist zur Positionierung innerhalb von Gebäuden geeignet, da es zum einen eine weit verbreitete Infrastruktur ist, auf vielen mobilen Plattformen verfügbar ist, Wände durchdringt und Standard WLAN Access Points bereits eine Lokalisierung auf Raum-Genauigkeit ermöglicht.

Aus all diesen Gründen haben wir uns mit der Positionierung mittels WLAN beschäftigt.

²<https://play.google.com/store/apps/details?id=com.google.android.apps.unveil>

³<http://stackoverflow.com/questions/2080731/google-goggles-api>

2.4.1 Sammeln von WLAN Fingerprints

Um später Vergleiche im Client anstellen zu können mussten wir zuerst Daten des Netzwerkes sammeln. Ein Access Point wird dabei eindeutig durch eine **BSSID** gegenzeichnet und der Client gibt Auskunft über die empfangene Signalstärke (**RSS**⁴), welche beobachtet und aufgezeichnet werden kann.

Ziel dieser Phase war es an möglichststen vielen Punkten in der Hochschule die *RSS* zu messen und diese zu einem Punkt auf der Karte der Hochschule zu speichern.



Dieses Feature ist im *Collector* umgesetzt.

2.4.2 Kalibrierung

Da das Sammeln der WLAN Fingerprints mit einem Smartphone realisiert wird und wir davon ausgehen mussten, dass nicht jeder Nutzer das gleiche Smartphone besitzt, mussten wir uns eine Möglichkeit der Kalibrierung überlegen. Dazu haben wir überlegt, dass der Nutzer zuerst in einer Kalibrierungsphase selbst einen Fingerprint erstellt von einem von uns festgelegten Ort und diesen mit dem von uns gemessenen Fingerprint vergleicht. Dadurch bekommen wir einen Faktor um den das Smartphone des Nutzer von unserem Gerät abweicht. Da wir vermuten, dass die WLAN Antennen der Smartphones auch in verschiedenen Bereichen, hohe, mittlere und niedrige Signalstärke, sich stark unterscheiden berechnen wir diesen Faktor für die gerade genannten Bereiche.

Dieser Teilbereich ist in der *StudMap-App* umgesetzt.

2.4.3 Positionierung mittels WLAN Fingerprints

Um die Position eines Nutzers ermitteln zu können muss dieser, wie der *Collector* einen Fingerprint des WLANs an seiner aktuellen Position erstellen. Diesen Fingerprint und seine Faktoren, welche während der Kalibrierung ermittelt wurden, schickt

⁴RSS: received signal strength wird in dBm gemessen.



2 Recherche

der Client zum Server, welcher durch Vergleiche den Standpunkt ermittelt und zurückgibt.

Dieses Feature ist auch in der *StudMap-App* umgesetzt.

Wie werden die FP verglichen.

3 Positionsermittlung

In diesem Kapitel wird beschrieben, wie die Position eines Anwenders auf der Karte bestimmt wird. Dazu werden die im Kapitel [Recherche](#) beschriebenen Verfahren verwendet.

3.1 QR-Tags

An den Räumen werden QR-Tags angebracht, die eine Zuordnung zu einem Knoten auf der ermöglicht. Hier ist zu beachten, dass die Informationen auf dem QR-Tag auch für andere Anwendungen nützlich sein sollen. Deshalb kann hier nicht nur eine Knoten-ID hinterlegt werden.

Folgendes JSON-Format ist ein möglicher Kandidat:

```
1 {  
2   "General": {  
3     "Label": "A2.1.10",  
4     "Name": "Aquarium"  
5   },  
6   "StudMap": {  
7     "NodeId": "12",  
8     "Url": "https://code.google.com/p/studmap/"  
9   }  
10 }
```

4 Benutzerverwaltung

In einem ASP.NET MVC 4 Projekt ist bereits eine vollständige Benutzerverwaltung integriert, die wir auch in unserem Projekt benutzen wollen. Durch die integrierte Benutzerverwaltung sind Webseiten zur Registrierung und für den Login / Logout bereits fertig.

Abbildung 4.1: Webseite zur Registrierung im StudMap Admin

Für die Benutzerverwaltung verwendet das ASP.NET MVC 4 Projekt folgende Datenbankstruktur:

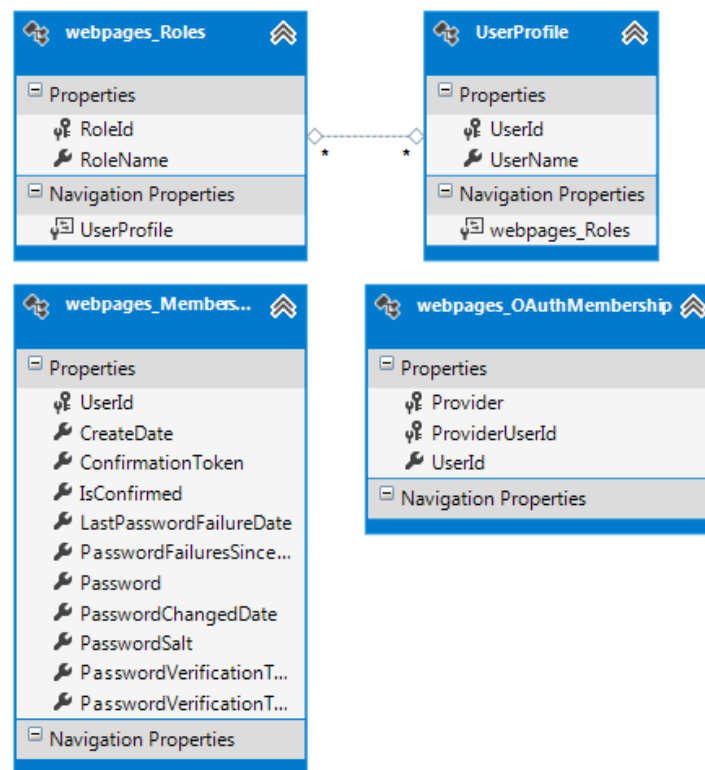
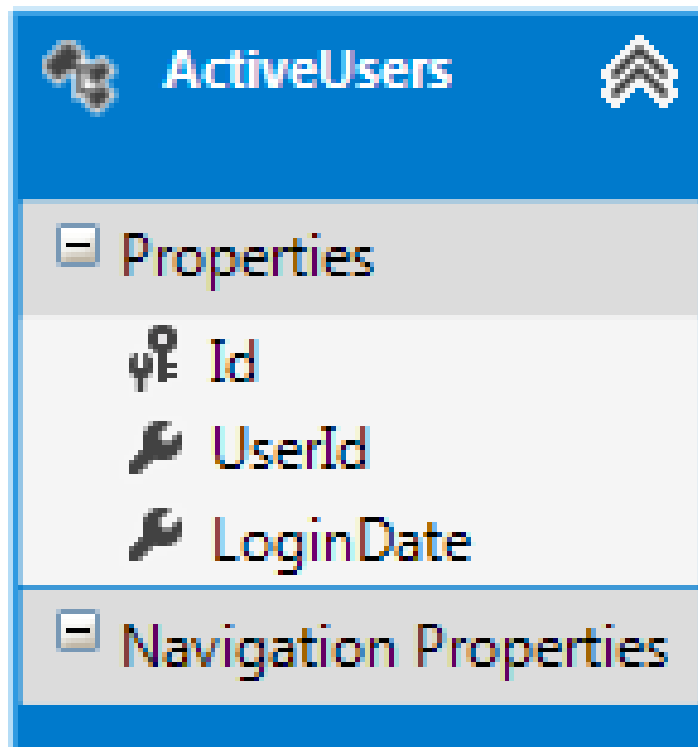


Abbildung 4.2: Datenbankstruktur der integrierten Benutzerverwaltung

Für unser Projekt sind nur die drei Tabellen **UserProfile**, **webpages_Roles** und **webpages_Membership** relevant. Wie im Domain Model bereits beschrieben unterscheiden wir zwischen den Benutzerrollen Benutzer und Administrator. Jeder Anwender kann sich in mehreren Benutzerrollen befinden. Zusätzlich sind in der Tabelle **webpages_membership** weitere Anwenderdaten wie beispielsweise das Datum der Registrierung das Passwort hinterlegt. Damit ist die Benutzerverwaltung für den Administrationsbereich vollständig.

Um die (beispielsweise über das Smartphone) am System angemeldeten Clients zu verwalten haben wir eine weitere Tabelle **ActiveUsers** hinzugefügt:



ActiveUsers	
[-] Properties	
Id	
UserId	
LoginDate	
[-] Navigation Properties	

Abbildung 4.3: Tabelle ActiveUsers

Für den StudMap Admin genügt die bereits integrierte Benutzerverwaltung. Allerdings benötigen wir noch eine Schnittstelle, damit auch die Web bzw. Smartphone Clients auf die Benutzerverwaltung zugreifen können. Siehe dazu Kapitel [Verwendung der Benutzerschnittstelle](#).



5 Webservice

Für den Zugriff auf die Daten stellen wir den Webservice `StudMap.Service` zur Verfügung. Der Webservice besteht dabei aus zwei Schnittstellen in Form von sogenannten Controller Klassen, die jeweils von der Klasse `ApiController`⁵ abgeleitet sind:

- `MapsController`: Verwaltung von Karten- und Routeninformationen
- `UsersController`: Verwaltung von Benutzerinformationen

Bevor nun die Funktionen der jeweiligen Controller Klasse erläutert werden, folgt eine Übersicht, über die verschiedenen Rückgabe Werte und ihre Bedeutung.

5.1 Allgemeine Objekte

Im folgenden werden die im Webservice verwendeten Objekte aufgeführt. Für jedes Objekt werden Eigenschaften und die Repräsentation der Daten im JSON-Format aufgelistet.

5.1.1 Edge

Repräsentiert eine Kante in einem Graphen.

Eigenschaften:

StartNodeId	ID des Start-Knotens der Kante.
EndNodeId	ID des End-Knotens der Kante.

Beispiel:

```
1 {  
2   "StartNodeId": 12,  
3   "EndNodeId": 6  
4 }
```

⁵siehe: [MSDN Dokumentation](#)

5.1.2 Node

Repräsentiert einen Knoten in einem Graphen.

Eigenschaften:

Id	ID des Knotens.
X	X-Koordinate auf dem Bild des Stockwerks. Wertebereich: 0.0 - 1.0. 0.0 bedeutet linker Bildrand. 1.0 bedeutet rechter Bildrand.
Y	Y-Koordinate auf dem Bild des Stockwerks. Wertebereich: 0.0 - 1.0. 0.0 bedeutet oberer Bildrand. 1.0 bedeutet unterer Bildrand.
FloorId	ID des Stockwerks auf dem sich der Knoten befindet.

Beispiel:

```
1 {  
2   "Id": 12,  
3   "X": 0.45,  
4   "Y": 0.76,  
5   "FloorId": 2  
6 }
```

5.1.3 Graph

Repräsentiert für ein Stockwerk den entsprechenden Graphen.

Eigenschaften:

FloorId	ID des Stockwerks, das der entsprechende Graph repräsentiert.
Edges	Eine Liste von Kanten (s. Edge).
Nodes	Eine Liste von Knoten (s. Node).

Beispiel:

```
1 {  
2   "FloorId": 12,  
3   "Edges": { {...}, {...}, {...} },  
4   "Nodes": { {...}, {...}, {...} }  
5 }
```

5.1.4 Pathplot

Repräsentiert für die Darstellung benötigte Daten.

Eigenschaften:

Id	??
Classes	??
Points	Eine Liste von Node Objekten.

Bitte noch
entspre-
chend
vervoll-
ständigen!

Beispiel:

```
1 {  
2   "Id": "flt-1",  
3   "Classes": "planned",  
4   "Points": { {...}, {...}, {...} }  
5 }
```

5.1.5 FloorPlanData

Repräsentiert Daten die auf einem Bild dargestellt werden können.

Eigenschaften:

Pathplot	
Graph	

Bitte noch
entspre-
chend
vervoll-
ständigen!

Beispiel:

```
1 {  
2   "Pathplot": {...},  
3   "Graph": {...}  
4 }
```

5.1.6 Room

Repräsentiert Daten die für einen Raum relevant sind.

Eigenschaften:

NodeId	ID des Knotens an dem sich der Raum befindet.
DisplayName	Der Anzeigename für den Raum (z.B. Aquarium)
RoomName	Der eigentliche Raumname (z.B. A 2.0.11).

Beispiel:

```
1 {  
2   "NodeId": 12,  
3   "DisplayName": "Aquarium",  
4   "RoomName": "A2.0.11"  
5 }
```

5.1.7 PoiType

Repräsentiert einen **Poi** Typen, wie beispielsweise Mensa oder Bibliothek.

Eigenschaften:

Id	ID des Poi Typs.
Name	Der Name des Poi Tpes (z.B. Mensa).

Beispiel:

```
1 {  
2   "Id": 12,  
3   "Name": "Mensa"  
4 }
```

5.1.8 Poi

Repräsentiert einen Point Of Interest, wie beispielsweise eine Mensa oder eine Bibliothek.

Eigenschaften:

Type	Typ des PoIs (s. PoiType).
Description	Beschreibung des PoIs.

Beispiel:

```
1 {  
2   "Type": 1,  
3   "Description": "In der Mensa kann man essen."  
4 }
```

5.1.9 NodeInformation

Repräsentiert die für einen Knoten relevanten Daten.

Eigenschaften:

DisplayName	Anzeigename für den Knoten (z.B. Dr. Schulten, Martin).
RoomName	Raumname für den Knoten (z.B. B2.0.03).
Node	??
PoI	??
QRCode	Dem Knoten zugeordnetem QR Code.
NFCTag	Dem Knoten zugeordnetem NFC Tag.

Beispiel:

```
1 {  
2   "DisplayName": "Dr. Schulten, Martin",  
3   "RoomName": "B2.0.03",  
4   "Node": {...},  
5   "PoI": {...},  
6   "QRCode": "",  
7   "NFCTag": ""  
8 }
```

5.1.10 Floor

Repräsentiert die für ein Stockwerk relevanten Daten.

Eigenschaften:

Id	ID des Stockwerks.
MapId	ID der Karte zu dem das Stockwerk gehört.
Name	Name des Stockwerks.
ImageUrl	Der Dateipfad auf dem Server zum Bild des Stockwerks.
CreationTime	Zeitstempel, an dem das Stockwerk erstellt wurde.

Beispiel:

```
1 {  
2   "Id": 1011,  
3   "MapId": 3,  
4   "Name": "Ebene 0",  
5   "ImageUrl": "Images/Floors/RN_Ebene_0.png",  
6   "CreationTime": "2013-11-18 14:36:24.607"  
7 }
```

5.1.11 Map

Repräsentiert die für eine Karte relevanten Daten.

Eigenschaften:

Id	ID der Karte.
Name	Name der Karte.

Beispiel:

```
1 {  
2   "Id": 3,  
3   "Name": "Westfälische Hochschule",  
4 }
```

5.1.12 User

Repräsentiert die für einen Benutzer relevanten Daten.

Eigenschaften:

Name	Name des Benutzers.
------	---------------------

Beispiel:

```
1 {  
2   "Name": "Daniel",  
3 }
```

5.1.13 FloorGraph

Repräsentiert einen Teilgraph für ein Stockwerk.

Eigenschaften:

floorId	ID des Stockwerks.
graph	Der Teilgraph (s. Graph) für das Stockwerk.

Beispiel:

```
1 {  
2   "floorId": 2,  
3   "graph": {...}  
4 }
```

5.1.14 NodeInformationWithId

nodeId	ID des Node .
nodeInf	Die Knoteninformation die gespeichert werden soll (s. NodeInformation).

Beispiel:

```
1 {  
2   "nodeId": 12,  
3   "nodeInf": {...}  
4 }
```

5.2 Rückgabe Objekte

5.2.1 BaseResponse

Allgemeine Rückgabe vom Service, die einen Status und ggf. einen Fehler enthält.
Die Daten werden im JSON Format zurück gegeben.

Beispiel:

```
{ "Status":1, "ErrorCode":0 }
```



5.2.1.1 ResponseStatus

Wir unterscheiden zwischen:

- `None` = 0: Defaultwert
- `Ok` = 1: Funktion erfolgreich ausgeführt
- `Error` = 2 Fehler bei Funktionsausführung

5.2.1.2 ResponseError

Ist beim `ResponseStatus` `Error` gesetzt. Es werden folgende Fehlerszenarien unterschieden:

Allgemein:

- 001 - `DatabaseError`:
Fehler bei der Ausführung einer Datenbankabfrage.

Registrierung:

- 101 - `UserNameDuplicate`:
Der Benutzername ist bereits vergeben.
- 102 - `UserNameInvalid`:
Der Benutzername ist ungültig.
- 103 - `PasswordInvalid`:
Das Passwort ist ungültig.

Anmeldung:

- 110 - `LoginInvalid`:
Die Logindaten (Name oder Passwort) sind ungültig.

Maps:

- 201 - `MapIdDoesNotExist`:
Zur angeforderten `MapId` existiert keine Map.
- 202 - `FloorIdDoesNotExist`:
Zur angeforderten `FloorId` existiert kein Floor.



5.2.2 ObjectResponse

Eine generische Klasse die **BaseResponse** um ein Feld **Object** erweitert, indem die Nutzdaten gespeichert werden.

Beispiel:

ObjectResponse
als JSON
String
einfügen

5.2.3 ListResponse

Eine generische Klasse die **BaseResponse** um eine Liste **List** erweitert, indem Nutzdaten in Form einer Collection gespeichert werden.

Beispiel:

```
{"List": [], "Status": 1, "ErrorCode": 0}
```

5.3 MapsController

5.3.1 CreateMap

Erstellt eine neue Karte mit dem vorgegebenen Namen.

POST [/api/Maps/CreateMap?mapName=WHS](#)

Parameter:

mapName	Sprechender Name der Karte.
---------	-----------------------------

Rückgabewert: **ObjectResponse**, **Map**

5.3.2 DeleteMap

Löscht eine Karte mit der angegebenen ID.

POST [/api/Maps/DeleteMap?mapId=2](#)

Parameter:

mapId	ID der Map, die gelöscht werden soll.
-------	---------------------------------------

Rückgabewert: **BaseResponse**



5.3.3 GetMaps

Liefert eine Liste aller Karten zurück.

GET </api/Maps/GetMaps>

Rückgabewert: [ListResponse](#), [Map](#)

5.3.4 SaveGraphForFloor

Speichert den Graphen zu einem Stockwerk ab.

POST </api/Maps/SaveGraphForFloor>

POST Body: [FloorGraph](#)

Rückgabewert: [ObjectResponse](#), [Graph](#)

5.3.5 DeleteGraphForFloor

Löscht den Teilgraphen auf einem Stockwerk. Die Teilgraphen auf anderen Stockwerken werden nicht verändert. Kanten die Stockwerke mit diesem Stockwerk verbinden, werden ebenfalls entfernt.

POST </api/Maps/DeleteGraphForFloor?floorId=2>

Parameter:

floorId	ID des Stockwerks, dessen Teilgraph gelöscht werden soll.
---------	---

Rückgabewert: [BaseResponse](#)

5.3.6 GetGraphForFloor

Liefert den Teilgraphen für ein Stockwerk.

GET </api/Maps/GetGraphForFloor?floorId=2>

Parameter:

floorId	ID des Stockwerks.
---------	--------------------

Rückgabewert: [ObjectResponse](#), [Graph](#)

5.3.7 GetNodeInformationForNode

Liefert weitere Informationen zu einem Knoten. Diese umfassen Raumnummern, zugeordnete NFC- und QR-Tags, usw.

GET </api/Maps/GetNodeInformationForNode?nodeId=12>

Parameter:

nodeId	ID des Knotens.
--------	-----------------

Rückgabewert: [ObjectResponse](#), [Graph](#)

5.3.8 GetFloorPlanData

Liefert die verschiedenen Schichten auf einem Stockwerk.

GET </api/Maps/GetFloorPlanData?floorId=2>

Parameter:

floorId	ID des Stockwerks.
---------	--------------------

Rückgabewert: [ObjectResponse](#), [FloorPlanData](#)

5.3.9 SaveNodeInformation

Speichert zusätzliche Informationen zu einem Knoten ab.

POST </api/Maps/SaveNodeInformation>

POST Body: [NodeInformationWithId](#)

Rückgabewert: [ObjectResponse](#), [NodeInformation](#)

5.3.10 GetRouteBetween

Liefert die Route zwischen zwei Knoten, wenn diese existiert.

GET </api/Maps/GetRouteBetween?mapId=2&startNodeId=12&endNodeId=46>

Parameter:

mapId	ID der Karte, auf der die Route bestimmt werden soll.
startNodeId	ID des Startknotens.
endNodeId	ID des Zielknotens.

Rückgabewert: [ListResponse](#), [Node](#)



5.3.11 GetPoiTypes

Liefert eine Liste aller Typen von PoIs zurück.

GET </api/Maps/GetPoiTypes>

Rückgabewert: [ListResponse](#), [PoiType](#)

5.3.12 GetPolsForMap

Liefert eine Liste aller PoIs auf einer Karte zurück.

GET </api/Maps/GetPoIsForMap?mapId=2>

Parameter:

mapId	ID der Karte.
-------	---------------

Rückgabewert: [ListResponse](#), [Poi](#)

5.3.13 GetRoomsForMap

Liefert eine Liste aller Räume auf einer Karte zurück.

GET </api/Maps/GetRoomsForMap?mapId=2>

Parameter:

mapId	ID der Karte.
-------	---------------

Rückgabewert: [ListResponse](#), [Room](#)

5.3.14 GetNodeForNFC

Sucht auf einer Karte nach einem Knoten mit einem bestimmten NFC-Tag.

GET </api/Maps/GetNodeForNFC?mapId=2&nfcTag=46A6CG739ED9>

Parameter:

mapId	ID der Karte.
nfcTag	NFC-Tag, nach dem gesucht werden soll.

Rückgabewert: [ObjectResponse](#), [Node](#)



5.3.15 GetNodeForQRCode

Sucht auf einer Karte nach einem Knoten mit einem bestimmten QR-Code.

GET </api/Maps/GetNodeForQRCode?mapId=2&qrCode=46A6CG739ED9>

Parameter:

mapId	ID der Karte.
qrCode	QR-Code, nach dem gesucht werden soll.

Rückgabewert: [ObjectResponse](#), [Node](#)

5.4 UsersController

Der `UserController` stellt klassische Funktionen zur Benutzerverwaltung zur Verfügung.

5.4.1 Register

Registriert einen neuen Anwender in der Benutzerrolle Benutzer.

POST </api/Users/Register?userName=test&password=geheim>

Parameter:

userName	Der Benutzername.
password	Passwort im Klartext.

Rückgabewert: [BaseResponse](#)

5.4.2 Login

Meldet einen bereits registrierten Anwender am System an.

POST </api/Users/Login?userName=test&password=geheim>

Parameter:

userName	Der Benutzername.
password	Passwort im Klartext.

Rückgabewert: [BaseResponse](#)



5.4.3 Logout

Meldet einen angemeldeten Anwender vom System ab.

GET <http://localhost:1129/api/Users/Logout?userName=test>

Parameter:

userName	Der Benutzername.
----------	-------------------

Rückgabewert: [BaseResponse](#)

5.4.4 GetActiveUsers

Ermittelt eine Liste der aktuell am System angemeldeten Anwender.

GET </api/Users/GetActiveUsers>

Rückgabewert: [ListResponse](#), [User](#)

6 Verwendung des Webservices

6.1 Verwendung der Benutzerschnittstelle

6.1.1 Registrierung

Bevor sich ein Benutzer am StudMap System anmelden kann, muss er sich zunächst über die Funktion **Register** registrieren.

6.1.2 Aktive und inaktive Benutzer

Im StudMap System wird zwischen aktiven und inaktiven Benutzern unterschieden. Nachdem sich ein Benutzer am System registriert hat gilt dieser als inaktiv. Über die Funktion **Login** kann er sich am System anmelden und gilt somit als aktiv.

Damit der angemeldete Benutzer auch aktiv bleibt, sollte sich dieser in einem Zeitintervall von fünf Minuten über die Methode **Login** am System aktiv melden. Nach einer Inaktivität von 15 Minuten wird der Benutzer automatisch inaktiv.

Über die Funktion **Logout** kann sich ein Benutzer wieder vom System abmelden und wird somit inaktiv.

6.1.3 Aktive Benutzer abfragen

Die aktiven Benutzer können über die Funktion **GetActiveUsers** abgefragt werden. Damit die Anzeige der aktiven Benutzer im Client möglichst aktuell ist, sollte diese Abfrage ebenfalls in regelmäßigen Zeitabständen erfolgen.

Best
Practices
"Programmier-
Handbuch" für
MapsCon-
troller
schreiben.