

Indoor Navigation

by

Lennart (XXXXXXX)
Jan (XXXXXXX)
Eridy (XXXXXXX)
Andreas (XXXXXXX)

A project documentation submitted to

Technische Universität Berlin
School IV - Electrical Engineering and Computer Science
Department of Telecommunication Systems
Service-centric Networking

Project Documentation

February 13, 2016

Supervised by:
Sebastian Zickau and Mathias Slawik

Abstract

Short summary of project outcome.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Constraints	4
1.3	Functional Requirements	5
2	Localization Technologies	7
2.1	Related Work	7
2.2	Short range localization techniques	8
2.2.1	WiFi based positioning	8
2.2.2	Bluetooth based positioning	9
2.3	Bluetooth Low Energy	9
2.4	Estimote Beacons	10
2.4.1	iBeacon Protocol	11
2.4.2	Region Monitoring	11
2.4.3	Ranging	11
2.4.4	Estimote Beacon Drawbacks	12
2.5	Apple Core Location Framework	12
2.6	Evaluation of Available Technologies	13
2.7	CISCO MSE API Wrapper Tests	13
3	Concept and Design	17
3.1	Big Picture	17
3.2	API Considerations	18
3.3	Workload Split Between Clients and Backend	18
3.4	Authentication and Session Management	19
3.5	Database Design	20
4	Implementation	21
4.1	Backend	21
4.1.1	Architecture	21
4.1.2	Controllers	21
4.1.3	Routes	22
4.1.4	Models	22
4.1.5	Middleware	22
4.1.6	Test	22

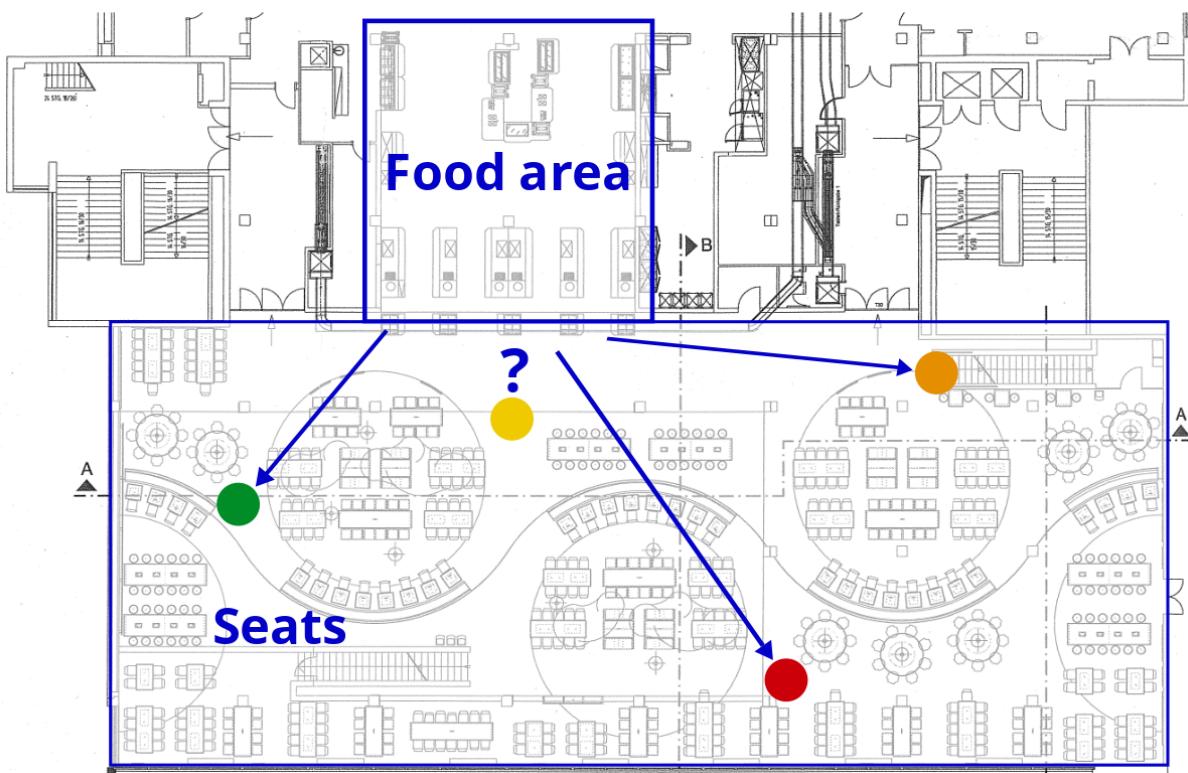
4.1.7	CYCLONE Federation Provider	23
4.2	Android	23
4.3	iOS	23
4.3.1	Floorplans	23
4.3.2	Login	24
4.3.3	Hotspots/ Buildings	24
4.3.4	Indoor Positioning	25
4.3.5	Friends	27
4.3.6	Groups	28
5	Evaluation	31
6	Conclusion	33
6.1	Future Work	33
List of Tables		35
List of Figures		37
Bibliography		38
Appendices		39
1	Tool for performing tests on the CISCO MSE API wrapper	41

1 Introduction

1.1 Motivation



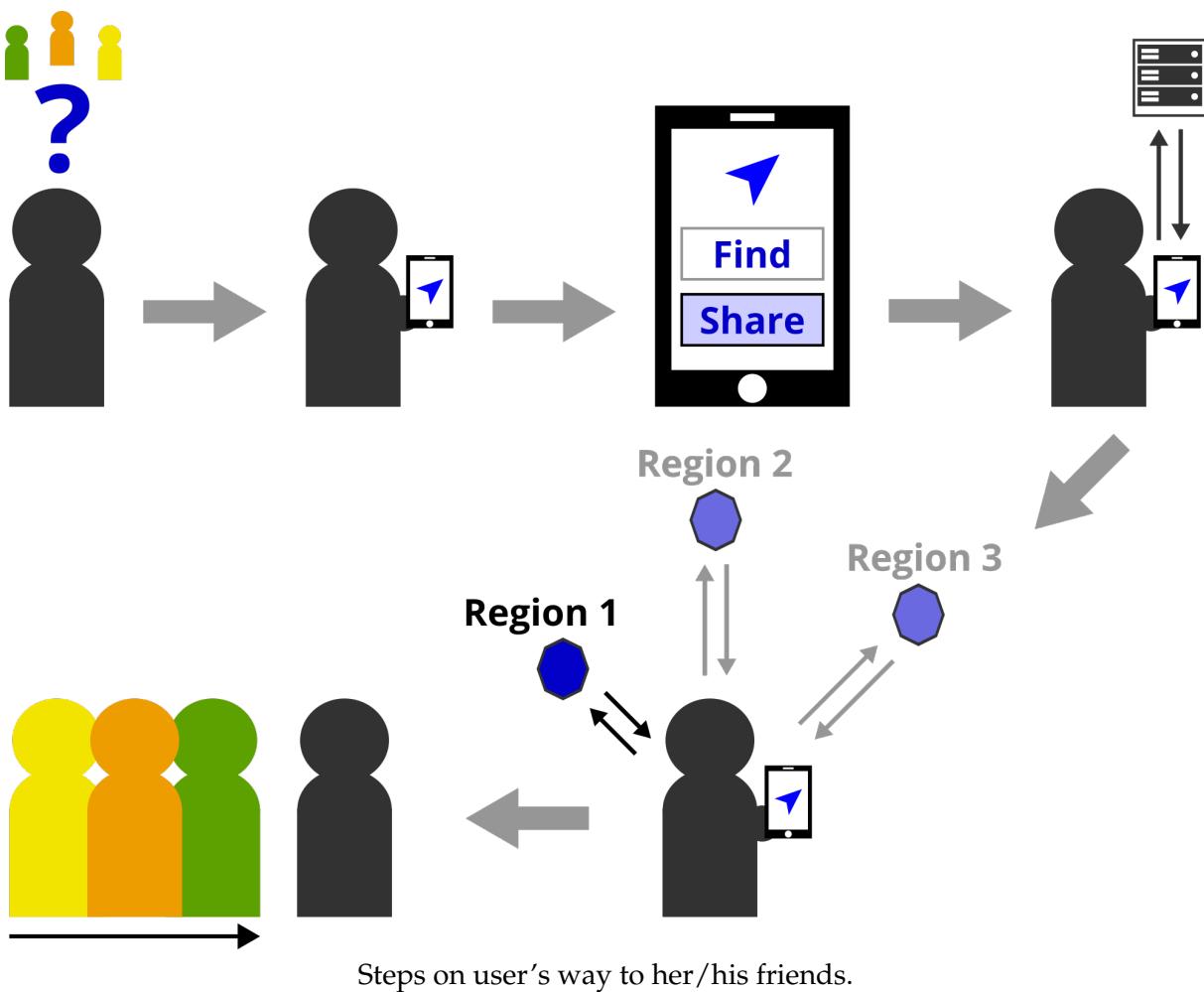
The mensa in Hardenbergstraße at TU Berlin.



Yellow person looking for her/his friends in crowded mensa.



Imagined use of our app on mensa tray.



1.2 Constraints

The idea of this project described above already introduced some implicit constraints in order to make it work. To have a complete overview of what we expect to have available to use our service, we defined the following constraints:

- **Everyone has a smartphone:** Obvious point. As this project involves developing clients for Android and iOS and relying on these clients to interact in an user friendly manner with the backend, a smartphone is needed. We do not, though, require the latest operating system versions. For Android, version 5.0 is required, for iOS, version X is required.
- **Participants have a TUB account (edugain account):** The service we provide needs an authentication prior to being able to modify or delete objects on backend side. Therefore some kind of user management was needed. As we will explain more deeply later in this document, for user authentication and session management we rely on a department service called CYCLONE Federation Provider¹. Via this service users with an edugain (and therefore a TUB) account are able to authenticate against our backend.

¹ <http://www.cyclone-project.eu/>

- **Minimal interaction:** As described in the previous section our intended use case is placed in situations where you probably carry a tray full of food and will not be having your hands free to interact with complex user interfaces. Therefore a very important constraint for our project was to provide applications that do not need a lot of intention in order to work as intended while at the same time strictly comply with the user's set preferences.
- **Easy to use:** This constraint correlates with the previous one. The setting our applications are intended for do not allow for complex user interfaces. Actions needed to be taken while using the services need to be easy to perform.
- **Low impact on device battery:** The possible use of bluetooth introduced the topic of battery usage. We therefore defined low battery usage as an important constraint for our project.

1.3 Functional Requirements

After motivation and constraints were set, we defined our functional requirements. These were the functions we had to implement to follow along our imagined flow throughout the application. Therefore a user has to be able to...

- **...login via university account:** As mentioned in our constraints, login should happen via the university account. We are using the CYCLONE Federation Provider and therefore things work a little different than in a self-developed login mechanism. For us this meant to integrate the provided OpenID Connect authentication flow in a user friendly way into the mobile applications. The applications in turn would then be authenticated against the backend.
- **...add friends via university mail:** All connected edugain members each have their own unique domain name and internally only contain unique users. It was the logical step to provide the functionality to add friends by their mail address as the unique identifier.
- **...share her/his location:** The most important part of our system is the modifiability of the logged in user only. This includes updating or deleting the user's location and defining in which conditions and with which friends the location should be shared.
- **...see shared locations of friends:** Friends' locations were supposed to be visualized on a map. To realise this a functionality to retrieve locations of friends directly in the user's area needed to be developed.

2 Localization Technologies

2.1 Related Work

Before we started our project we did research on related work regarding applied indoor navigation and positioning techniques applied on mobile devices. Most of them rely on calculating Received Signal Strength Indicator (RSSI) triangulation and/or Bluetooth BLE Beacon techniques or Wi-Fi Based Fingerprinting. The latter is a core idea of an indoor navigation system developed by [ChaSo2012]. They combine two methods in order to calculate a precise position. At first they use matching of pre-recorded received signal strength from nearby access points. This method is called “fingerprint matching”. The data is combined with a distance-based trilateration approach with at least three known access point coordinates which are also detected on the device. By this combination of both methods they received a high accuracy of the user position in an indoor environment.

[FelKyaZap2012] implemented an indoor Bluetooth-based positioning system on Bluetooth-capable devices as a PDA, which is comparable to the smartphones of today. They calculated range estimations based on approximations of relations between RSSI and distance between sender and receiver. The location estimation was also calculated via triangulation method.

Apple Inc. presented on WWDC2015 new features of their well known Core Location Frameworks providing Indoor positions and Floor information [wwdc15]. The closed beta is restricted by access to developers that are registered on Apples Indoor Program Maps Connect. Although Apple does not publish its algorithms, as stated on WWDC2015 by Apple Software Engineer Vitali Lovich the Core Location Frameworks also takes advantage of indoor WiFi signals in combination with motion and altitude sensors in order to provide the users with accurate position information as latitude and longitude.

In 2013 Apple also developed the iBeacon protocol which is a Bluetooth Low Energy (BLE) communication standard supported by iOS and Android devices [appleIBeacon]. The iBeacon protocol opened the door for new Beacon technologies in order to improve indoor navigation on mobile devices. The standard is incorporated by Estimote Inc. with Estimote Beacons. This indoor technique has successfully been installed in museums such as the Metropolitan Museum of Art project called: MediaLab Beacon Art Walk [MuseumArt]. However, this project aimed to provide additional locative information to the visitors rather than providing indoor positions. Next to the battery issues also faced in the MediaLab Project, Beacons having difficulties in cases of many people covering the clear line of sight to a device, makes actual retrieving of Latitude/Longitude coordinates in a room more difficult. Next to this, installed beacons made temporarily maintaining of Beacons mandatory, since one missing Beacon could prevent

successful indoor navigation in a calibrated system.

The Cisco MSE approach moves the calculation of a position away from a device [oreilly11]. Correctly installed cisco mse determines the location of any wireless device in a Building. The access points listen for Wi-Fi signals of devices and estimate its position also via trilateration. This solution can be seen at the American Museum of Natural History [AMNH15].

The Cisco MSE determines the location of any wireless device in a specific building. One approach is to make the building do the work instead of the device. Some Wi-Fi installations, such as the Cisco MSE, can determine the location of any wireless device in the building. The access points themselves listen for the Wi-Fi signals created by your phone, then estimate its position via trilateration. This solution has been deployed successfully at a few locations as mobile application for visitors, like the Explorer App at the American Museum of Natural History.

2.2 Short range localization techniques

Different localization techniques are presented and compared in book [brimcombe2009location]. For purpose of this project, we were interested mostly in short range positioning technologies. Short range positioning technologies cover small areas and are frequently employed for positioning in indoor environments. Most popular technologies are WiFi and Bluetooth, which are most commonly used to determine proximity location of mobile device connected to network. Other available technologies are Radio Frequency Identification (RFID), Ultra Wide Band Positioning, ultrasonic positioning, infrared positioning, camera-assisted and sensor-assisted positioning. We focused our research on WiFi and Bluetooth as those were the technologies provided for us by our supervisors.

2.2.1 WiFi based positioning

The basic principle of using WiFi for indoor positioning is measuring strength of signals received at two or more access points. The signals used for positioning are called beacon frames and are primarily intended for announcing the presence of a wireless LAN. There are two ways of transmitting these signals for positioning, uplink, when beacon frames are generated by mobile device and downlink, where beacon frames are generated by access points.

Actual positioning can be done in three ways. The simplest method is determining position of mobile device from position of the access point with the strongest signal, which is the closest to the mobile device. In this case a data linking access points and their locations is needed. The second method uses signal strength data from multiple access points to calculate the position of mobile device. This method provides more accurate position data. Third method uses fingerprinting approach, in which position is determined by matching signal data on mobile device against set of pre-collected data on signal strengths across the area. This approach offer the best results, but it requires initial measurement of the area before position tracking can take place.

WiFi based positioning only works in areas with good wireless network coverage. It is most commonly used in closed spaces such as offices, airports, cafes, shopping markets, hotels etc. It can also be used for outdoor positioning, usually in dense populated urban areas.

In general, WiFi positioning provides a proximity location instead of more specific coordinates. For example, it may provide information, in which building, floor and room mobile device is located. When using WiFi positioning there is no need for additional infrastructure, except for existing WiFi network and positioning server with database of all available access point along with their positions.

2.2.2 Bluetooth based positioning

Various solutions have been developed using Bluetooth for short range positioning. Bluetooth enabled devices are able to transmit signals containing information such as device identity and profile. When mobile device is in short range of device transmitting its location it can pick up such signals and use them for positioning. Signals can also be processed to determine the position of device, especially in case when mobile device can receive multiple signals from different devices. Position data can be exchanged between mobile devices using ad-hoc Bluetooth networks or with the location server in the network.

The signal strength decreases logarithmically with distance. This property can be used for calculating position. One method of positioning device is to triangulate data from different devices. This method offers higher accuracy than just detecting the closest device in range.

Bluetooth positioning can be deployed rapidly with easy maintenance and low cost. It can be used in applications where approximate positioning is sufficient.

Different implementations of WiFi and Bluetooth based localization technologies are described in more detail in following sections.

2.3 Bluetooth Low Energy

Multiple solutions are based on the already mentioned Bluetooth Low Energy (BLE) standard. To understand the reasons for this standard in comparison to classic Bluetooth better, we had "Bluetooth Low Energy - The Developer's Handbook" by Robin Heydon, published with Prentice Hall in 2012, available [[heydon2012bluetooth](#)]. In the following we try to point out some specifics of BLE taken from the just mentioned book.

Heydon starts by stating that although BLE has obvious connections to its parents, the classic Bluetooth standards, "Bluetooth low energy should be considered a different technology, [...]." (*ibid.*, p. 3). This is a consequence derived from the different design goals the developers had when they set out to standardize what would become Bluetooth Low Energy. The new standard is not a performance upgrade but by design focused on totally different obstacles. "When the low energy work started, the goal was to create the lowest-power short-range wireless technology possible." (*ibid.*, p. 7). This resulted in following goals: worldwide operation, low cost, robust, short range, low power (*ibid.*, p. 7).

Bluetooth Low Energy does not try to be yet another bandwidth upgrade for Bluetooth but rather a step towards extremely low power consumption as well as being very cheap. This latter directive makes it possible for Bluetooth Low Energy to be deployed in high volumes. In order to achieve this, Bluetooth Low Energy makes use of following three design points:

- **Operation on 2.4 GHz ISM band.** It's an overused band and has bad characteristics as

it is heavily absorbed by water which is the biggest part of the human body. But also no license fees are taken to operate on this band and therefore, “choosing to use the ISM band lowers the cost” (*ibid.*, p. 5).

- **Intellectual property (IP) license for usage.** Basically, the Bluetooth Special Interest Group (SIG) is claimed to be cheaper than other SIGs. Licenses are given under a FRAND policy which stands for “Fair, Reasonable, and Non-Discriminatory” (*ibid.*, p. 5).
- **Low power consumption.** Aiming at low power consumption overall reduces the costs of accompanying services and material costs.

This results in transmission speeds that are way below the ones of the classic Bluetooth versions, even lower than the speed of some very early standard versions of Bluetooth. Following table reviews the speeds of the existing Bluetooth variants (*ibid.*, table 1-1, p. 4):

Bluetooth standard	Maximum bandwidth
v1.1	1 MBps
v2.0	3 MBps
v3.0	54 MBps
v4.0 (BLE)	0.3 MBps

Therefore BLE is not tailored on the use cases classic Bluetooth tackles, “[...] because single mode Bluetooth low energy does not support audio for headsets and stereo music or high data rates for file transfers.” (*ibid.*, p. 6). It is to be used in situations where extremely low power consumption is needed while at the same time the amount of data transferred is kept low.

To survive in the congested frequency band it operates on, Bluetooth Low Energy uses adaptive frequency hopping. A technique that detects sources of interference, helps to avoid them in the future and quickly recovers from dropped packages. “It is this robustness that is absolutely key to the success of any wireless technology in the most congested radio spectrum available.” (*ibid.*, p. 8).

Another consideration is the distance BLE is able to cover and Heydon concludes that “[s]hort range means that Bluetooth low energy should be a **person area network**” (*ibid.*, p. 8).

2.4 Estimote Beacons

Estimote Beacons and Stickers are wireless sensors that can be attached to any location or object, embodying a wireless sensor network. The Beacons consists of a 32-bit ARM Cortex CPUs, equipped with an accelerometer, temperature sensor and a 2.4 Ghz radio. Using Smart Bluetooth 4.0.(Bluetooth low energy), the beacons send out signals with a range of up to 70 meters (Beacons) and 15 meters (Stickers). Though the signals are oftenly distracted under real world conditions, a range of about 40-50 and (Beacons) can be suspected. The battery can as stated by [developerDocsEstimote] last more than 3 years on default settings on a single CR2477 battery.

Estimote beacons are working with the Apple iBeacon protocol as well as the Eddystone open beacon format introduced by Google. These protocols working on top of the BLE technology standard are implemented in all smartphones devices enabling them to support new technologys like Apple Watch or Fitness trackers.

Using the Estimote SDK [developerDocsEstimote] mobile applications are enabled to receive and understand BLE Estimote signals in order to calculate the proximity to nearby locations and objects. The beacons specifics provide informations about their type, ownership and approximate locations, temperature or motions.

By detecting a beacon signal a phone can estimate the distance by measuring the received signal strength [developerDocsEstimote]. Since Bluetooth Low Energy does not need any pairing process between sender and receiver, the phone can constantly process new signals. This opens the doors for new technologic opportunities as indoor location/ indoor positioning.

2.4.1 iBeacon Protocol

The iBeacon protocol is a Bluetooth low energy communication protocol developed by Apple Inc. in 2013 and was introduced in iOS7 for indoor navigation. The protocol is supported by iOS7 devices as well as Android from version 4.3 up. The signal which is sent by a beacon is called advertisement.

These advertisements provides a so called iBeacon identifier, that is 20 bytes long and divided into three sections:

- UUID (16 bytes)
- major number (2 bytes)
- minor number (2 bytes)

These values provided by the advertisement can be modified according to own wishes. The hierarchical configuration of these values provides identifying informations about the beacon. While the UUID can be distinguished to a corporation, major and minor values can be used to distinguish between regions and sub-regions of a corporation.

2.4.2 Region Monitoring

Region monitoring triggers actions on the device on entering or exiting a beacon defined regions range. This works in depending the devices capabilities while an app is in foreground background or suspended. An app is limited to 20 regions being monitored. However by using a single UUID in multiple locations, a device can monitor many physical locations simultaneously [appleIBeacon].

2.4.3 Ranging

Ranging however triggeres actions on the devise based on the proximity to a beacon. The iBeacon protocol applies filters to the accuracy of a advertisement of one beacon. The filterestimation regarding the proximity to a beacon is indicated using one of four proximity states.

- Immediate

The Immediate proximity state represents a high level of confidence, that the device is physically very close to the beacon. This is in example holding the Smartphone directly on to of a beacon.

- Near

The Near proximity state indicates a proximity of round about 1-3 meters, if there are no obstructions between the device and the beacon which might cause distractions of the beacon.

- Far

The Far proximity state indicates a detected beacon without much confidence in the accuracy that is to low to determine wether it is Near or Immediate. The Far proximity state relies on the accuracy property to determine the potential proximity to the beacon.

- Unknown

The Unknown proximity state indicates a state where beacons are can not be determined. This my happen if the ranging has just begun or that the accuracy level is insufficient for measurements to determine a state that is either Far, Near or Immediate.

2.4.4 Estimote Beacon Drawbacks

For the implementation of Estimote Indoor Navigation for the project we decided to use Monitoring on the devices in order to detect the beacons, specifiyng a certain region. However, the devices still take at least about 30 seconds to recognize the fact that a beacon is out of range. This is a built-in and non adjustable delay in order to prevent “false” exit events.[\[developerDocsEstimote\]](#) This is a major drawback regarding a use-case of the project, that a user might pass multiple beacons of a location and constantly update new locations without the need to stop at each beacon.

This is solved by in additional applied ranging of all beacons that are monitored in order to process the “nearest” beacon as a users location. The devices report then the beacons in an order that is best guess of their proximity regarding issues of signal attenuation. This order however may still not be correct [\[appleIBeacon\]](#).

2.5 Apple Core Location Framework

As firstly presented on WWDC2015 [\[wwdc15\]](#) Apple presented new functionalities to the already existing Core Location Framework which is the major Framework on iOS devices for location service [\[CLlocation\]](#). This Framework takes the user in charge of wether the app can use locations services on the device or not.

The Core Location Framework uses Cellular data to provide an aproximity in which area in a city a user is. Additionally it uses GPS based on Satellite signals to improve the position of the user as well as sorounding Wi-Fi signals.

As soon as the user enters an indoor venue, the iOS system turns down GPS and Cellular sensors and enlightens Motion and Wi-Fi sensors. These sensors are used in combination with the remaining GPS signals, comming through the windows to locate the user indoors. The Motion sensor hereby gives information to the system that a person is moving and how fast the user is moving, while wi-fi signals are feeded to the CLLocation Framework to calculate the exact position.

Apple also added the altitude and floor attributes to the CLLocation Framework in order to

provide the user with the correct floor attributes [CLlocation].

For the Project the new CLLocation Framework was used to show the users Positon in the Mensa. After the first indoor tests in the mensa revealed precise positioning of a user indoors. However the Framework also revealed large susceptibility in areas where the wi-fi signal was apparently weak. i.e. in the upper left corner of the Mensa which is surrounded by walls that are affecting the router wi-fi signals.

The Frameworks started in an early beta when this project started and is constantly improved. To enable the full abilities of CLLocation Framework indoors, the venue needs to be registered and enabled by apple, using Maps Connect program [mapsConnect] in order to unlock CLLocation Frameworks full potential.

2.6 Evaluation of Available Technologies

2.7 CISCO MSE API Wrapper Tests

In order to determine whether the CISCO MSE API wrapper provided by tubIT would be sufficient for the project's requirements we were asked to perform tests on it. Especially it was asked for details on how the API worked where, so what values could be retrieved via the API wrapper in which locations on campus and how precise the values would turn out to be.

Concerning use cases our project was focused on the mensa and the library, therefore we initially planned to be conducting tests in only these two locations. As the provided wrapper around the CISCO MSE API we had access to delivered back one short, simple XML line we decided to invest some time into developing a small tool which routinely queried the API for its current status and saved the result into an easily readable JSON file for later investigation. The source code of the developed tool you can find in appendix 1.

```
<Info changedOn="2015-10-30T15:32:27.185+0100"
confidenceFactor="40.0" building="MAR" floor="Erdgeschoss"
WLAN-Status="EDUROAM" username="[REDACTED]"
lon="13.323857725896488" lat="52.516801767504241"/>
```

What the CISCO MSE API wrapper response looks like.

It was planned to be conducting the tests on one day in the mensa and on another day in the library. On the first day we started around noon and ran the test tool on one of our notebooks connected to university WiFi, "eduroam". We started in the south-western corner near the windows, walked towards the south-eastern corner, went to the stairs in the northern part and upstairs and again at the window front to the south-western corner on the first floor. As it turned out, the results we got back were definitely not what we had hoped for, most importantly because longitude and latitude of the requesting user were missing completely. The following listing shows the first ten results logged in two second periods from the API wrapper:

¹ {

```
2 "signal": [
3     {"timestamp": 1445344391, "latitude": "0.0000000000000000", "longitude": "0.0000000000000000", "building": "Mensa", "floor": "Mensa 1. OG"},  
4     {"timestamp": 1445344393, "latitude": "0.0000000000000000", "longitude": "0.0000000000000000", "building": "Mensa", "floor": "Mensa 1. OG"},  
5     {"timestamp": 1445344395, "latitude": "0.0000000000000000", "longitude": "0.0000000000000000", "building": "Mensa", "floor": "Mensa 1. OG"},  
6     {"timestamp": 1445344397, "latitude": "0.0000000000000000", "longitude": "0.0000000000000000", "building": "Mensa", "floor": "Mensa 1. OG"},  
7     {"timestamp": 1445344399, "latitude": "0.0000000000000000", "longitude": "0.0000000000000000", "building": "Mensa", "floor": "Mensa 1. OG"},  
8     {"timestamp": 1445344401, "latitude": "0.0000000000000000", "longitude": "0.0000000000000000", "building": "Mensa", "floor": "Mensa 1. OG"},  
9     {"timestamp": 1445344404, "latitude": "0.0000000000000000", "longitude": "0.0000000000000000", "building": "Mensa", "floor": "Mensa 1. OG"},  
10    {"timestamp": 1445344406, "latitude": "0.0000000000000000", "longitude": "0.0000000000000000", "building": "Mensa", "floor": "Mensa 1. OG"},  
11    {"timestamp": 1445344408, "latitude": "0.0000000000000000", "longitude": "0.0000000000000000", "building": "Mensa", "floor": "Mensa 1. OG"},  
12    {"timestamp": 1445344410, "latitude": "0.0000000000000000", "longitude": "0.0000000000000000", "building": "Mensa", "floor": "Mensa 1. OG"},  
13    ...
14  ]  
15 }
```

Clearly it can be observed that the longitude and latitude values are unusable. Another take away was that the floor change during our test did not reflect into our captured results. Therefore we decided to directly test the library for comparable results. Inside the library, we started on ground floor and went upstairs in "circles" through the different levels. From the fourth floor we went back down straight forward. During that second test ten of the first fifteen responses from the API looked like:

```
1 {
2     "signal": [
3         {"timestamp": 1445345843, "latitude": "0.0000000000000000", "longitude": "0.0000000000000000", "building": "BIB", "floor": "Erdgeschoss"},  
4         {"timestamp": 1445345845, "latitude": "0.0000000000000000", "longitude": "0.0000000000000000", "building": "BIB", "floor": "Erdgeschoss"},  
5         {"timestamp": 1445345848, "latitude": "0.0000000000000000", "longitude": "0.0000000000000000", "building": "BIB", "floor": "Erdgeschoss"}]
```

```

6     Erdgeschoss"},  

7     {"timestamp": 1445345850, "latitude": "0.0000000000000000", "  

8         longitude": "0.0000000000000000", "building": "BIB", "floor": "  

9             Erdgeschoss"},  

10    {"timestamp": 1445345852, "latitude": "0.0000000000000000", "  

11        longitude": "0.0000000000000000", "building": "BIB", "floor": "  

12            Erdgeschoss"},  

13    {"timestamp": 1445345854, "latitude": "0.0000000000000000", "  

14        longitude": "0.0000000000000000", "building": "BIB", "floor": "  

15            Erdgeschoss"},  

16    ...  

17    {"timestamp": 1445345870, "latitude": "0.0000000000000000", "  

18        longitude": "0.0000000000000000", "building": "BIB", "floor": "1.  

19            Obergeschoss"},  

20    {"timestamp": 1445345872, "latitude": "0.0000000000000000", "  

21        longitude": "0.0000000000000000", "building": "BIB", "floor": "1.  

22            Obergeschoss"},  

23    {"timestamp": 1445345874, "latitude": "0.0000000000000000", "  

24        longitude": "0.0000000000000000", "building": "BIB", "floor": "1.  

25            Obergeschoss"},  

26    {"timestamp": 1445345876, "latitude": "0.0000000000000000", "  

27        longitude": "0.0000000000000000", "building": "BIB", "floor": "1.  

28            Obergeschoss"},  

29    ...  

30    ]  

31 }

```

First, longitude and latitude were again unusable. This time though the floor information worked quite reliably and indicated very fast on which floor we currently measured. After that we were wondering whether eventually we would get back longitude and latitude values anywhere on campus and decided to give it one last try by taking one more measurement in the MAR building (Marchstraße).

One more measurement turned into three as during the first two attempts we got sudden disconnects and therefore unusable results. We walked the whole foyer from north to south side and somewhere near the entrance we suspect the wireless signal got bad and our notebook conducting the tests disconnected. In the third try though we finally were able to get back usable results, in which chosen ten results logged looked like this:

```

1 {  

2     "signal": [  

3         {"timestamp": 1445350550, "latitude": "52.516903688639005", "  

4             longitude": "13.323958376544699", "building": "MAR", "floor": "  

5                 Erdgeschoss"},  

6         {"timestamp": 1445350552, "latitude": "52.516903688639005", "  

7             longitude": "13.323958376544699", "building": "MAR", "floor": "  

8                 Erdgeschoss"},  

9         {"timestamp": 1445350554, "latitude": "52.516903688639005", "  

10            longitude": "13.323958376544699", "building": "MAR", "floor": "  

11                Erdgeschoss"},  

12         {"timestamp": 1445350557, "latitude": "52.516903688639005", "  

13             longitude": "13.323958376544699", "building": "MAR", "floor": "  

14                 Erdgeschoss"},  

15     ]

```

```

7     Erdgeschoss"},  

8     {"timestamp": 1445350686, "latitude": "52.516864921942748", "  

9         longitude": "13.323953890659670", "building": "MAR", "floor": "  

10        Erdgeschoss"},  

11    {"timestamp": 1445350688, "latitude": "52.516864921942748", "  

12        longitude": "13.323953890659670", "building": "MAR", "floor": "  

13        Erdgeschoss"},  

14    {"timestamp": 1445350690, "latitude": "52.516864921942748", "  

15        longitude": "13.323953890659670", "building": "MAR", "floor": "  

16        Erdgeschoss"},  

17    ...
]
}

```

Finally we received some longitude and latitude values. Unfortunately the three different pairs of longitude and latitude above were the only ones we could observe during the whole walk from north end to south end of the foyer, thus still rather unusable values.

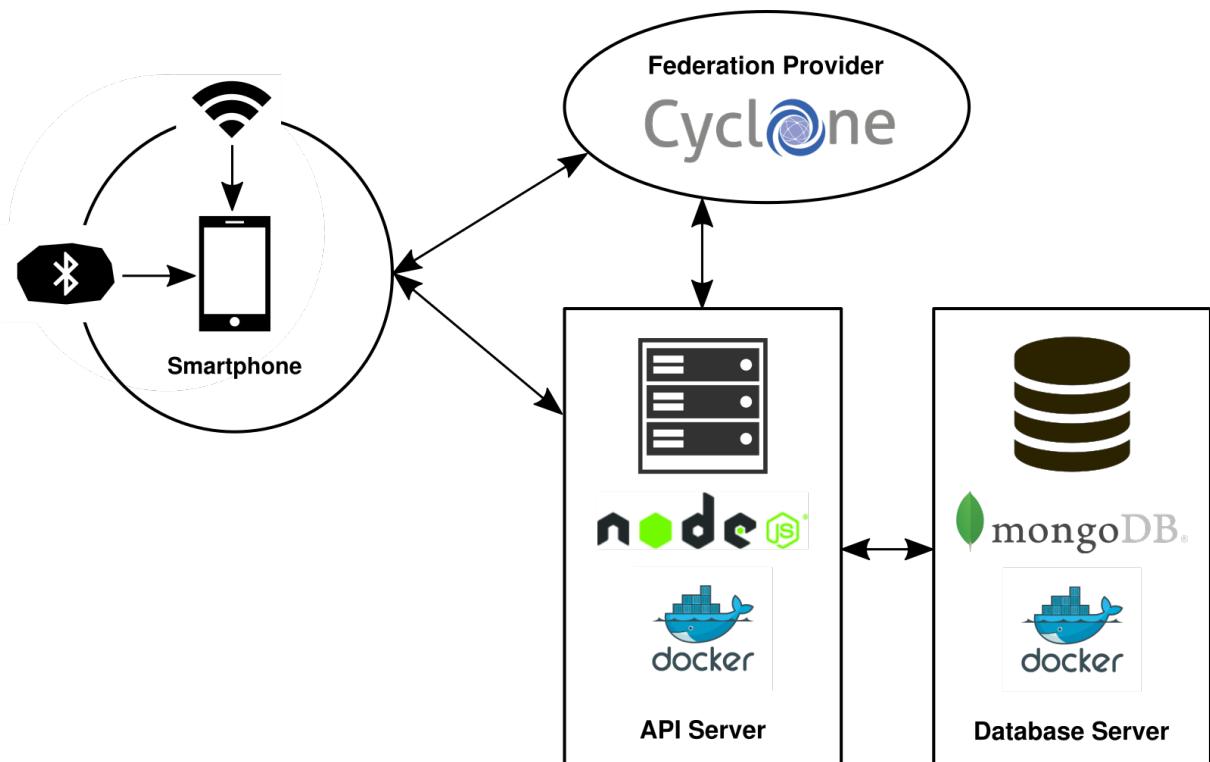
In the end, our conclusion at that point in the project progress was to use the CISCO MSE API wrapper provided by the tubIT in order to retrieve the rough position of a user. This means the building and floor the request originated from. We recommended back to our supervisors not to use this API for receiving longitude and latitude as these values were either quite imprecise or not available at all. The full result files of all our three measurements can be found at [\[ioslINavGitHub\]](#).

3 Concept and Design

3.1 Big Picture

As already mentioned our system consisted of two parts, the mobile clients side and the backend side which interconnectedly exchanged data. The backend part itself was split up again in three parts of which one was an “external” (means: SNET) resource, the CYCLONE Federation Provider. This entity provided us with user management and session handling functionalities so that we were able to out source these tedious and error-prone tasks to them. On the other side CYCLONE profited from our experiences with its rather young service. Concerning our part of the backend side, the task was to serve two machines independently, the API server and the database server. This was expected to be achieved with Docker.

To gain an understanding of what system we were trying to build, the following image should be at help:



All components of our envisioned system working together.

To sum it up, the idea was to gather position information aligned to the user's preferences locally on the smartphone, perform some processing steps on it, contact our backend for which authentication via the CYCLONE Federation Provider is needed and after that create, read, update or delete information (CRUD principle¹) in the backend and therefore in the database.

3.2 API Considerations

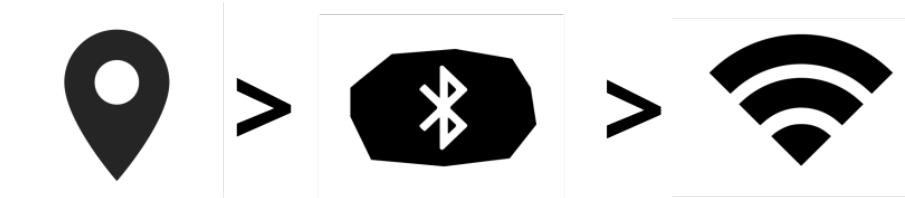
We had to decide on which paradigm our API should be based on. With the client-server and stateless nature of our envisioned system setup in combination with the just mentioned approach of relying on HTTP verbs such as POST, GET, PUT and DELETE for the CRUD operations, the decision to go for a RESTful API was quite clear ([fielding2000architectural], chapter 5).

Furthermore a widely supported and light-weight format for interchanging data between clients and backend was needed, and it was decided to use JSON². This also played nicely together with our requirement to implement the API server in Node.js³, using the Express web framework⁴ which provided us a convenient way to define the RESTful resources of our API.

3.3 Workload Split Between Clients and Backend

Our system setup required a certain workload split between the two logical parties, the client side and the backend side. The modification of data on backend side was designed to be very transparent to clients but still some values submitted on the endpoints needed preprocessing to be done on mobile application side.

The general idea is to have the clients handle all direct user interaction with the service in a consistent, user-friendly manner. The other important task on client side is to gather the position information of the requesting user and performing prioritization logic on all available information. This was needed to deliver only the most important position to backend where it would replace the old value. The following prioritization order was chosen:



Position prioritization order.

On backend side received data would then be sanitized, error and security checked and processed in its according request handler function. Therefore backend would handle everything concerning long term user state management, data persistence and API response aggregation.

¹ https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

² <http://json.org/>

³ <https://nodejs.org/en/>

⁴ <http://expressjs.com/>

3.4 Authentication and Session Management

The CYCLONE Federation Provider provided us with the ability to integrate a well-tested user authentication method with only little effort into our service. CYCLONE is based on the JBoss Keycloak⁵ project which in turn is based on the OpenID Connect standard⁶, OAuth 2.0⁷, JSON Web Token⁸ and SAML 2.0. In the following a short overview of the involved technologies will be given.

The OpenID Connect standard provides two important functionalities in focus of our project. The first one is the addition of an identity layer on top of the OAuth 2.0 Authorization Framework that enables developers to reliable verify what person is using the authenticated service, no matter the used client, be it web or native applications. OpenID Connect does this without the need to maintain password storages on developers' side. Specifically, the CYCLONE Federation Provider uses the Authorization Code Flow⁹ of the OpenID Connect standard. The other feature OpenID Connect brings into the project is that it already is built as a RESTful HTTP API based on JSON as the transport format and therefore perfectly integrated into the implementation and also provides the functionality to extend the specification in order to, for example, encrypt the transported identity data.

The OAuth 2.0 Authorization Framework is an IETF RFC [[hardt2012oauth](#)] which introduces an abstraction layer, the authentication layer, in distributed web application environments. Therefore the standard is designed for the HTTP protocol and does not specify other protocols. OAuth 2.0 provides the ability to differentiate between resource owners (e.g. end users on a service, the resource server) and requesting third parties that need access to some or all resources of the resource owner. Third parties will never need to be authenticated by the resource owner's own credentials but will instead request an access token issued for specific scope, access duration and further attributes from an authorization server. This flow lets the resource owner be in full control of all allowed access requests from third parties, enabling her/him to eventually revoke granted access. Furthermore the attack vector on each involved participants in the service is isolated by separating all authorizations via the access tokens. An extensive threat model analysis on OAuth 2.0 can be found in [[lodderstedt2013oauth](#)].

JWTs (pronounced: "jots") are another IETF RFC [[bradley2015json](#)] standardizing the transfer of a set of claims as a JSON object in a compact and URL-safe manner. The standard defines three parts of an JWT with the first one being the algorithm and token part, the second being the payload and the third used to verify the transported JWT. Encoding and concatenating these fields with dots yields the mentioned URL-safe representation predestined to be transported in HTTP Authorization headers or URI query parameters.

This authentication environment helped us to achieve multiple goals in our implementation. First of all users were able to authenticate via their university account as it is part of the edugain collaboration for which CYCLONE acts as an federation provider. Therefore we had a single-sign-on mechanism available by simply integrating CYCLONE. The other major advantage was that we never saw the user's credentials used to authenticate. We were able to

⁵ <http://keycloak.jboss.org/>

⁶ <https://openid.net/connect/>

⁷ <http://oauth.net/2/>

⁸ <https://jwt.io/>

⁹ https://openid.net/specs/openid-connect-core-1_0.html#CodeFlowAuth

rely on the well-tested user authentication code base provided implicitly through the layered mechanisms the CYCLONE Federation Provider contained.

3.5 Database Design

Part of the template we received at the beginning of the project from our supervisors at the SNET department was the integration of a MongoDB¹⁰ as the database for our service. The communication with the database was not done directly via the officially supported Node.JS MongoDB drivers but instead via another layer inbetween, an object mapping library called Mongoose¹¹.

MongoDB is a representative of the class of the NoSQL databases. These kind of databases reject the traditional database approach of laying out the whole data landscape in a relational, table-based fashion prior to implementing an architecture. In the class of NoSQL databases again multiple different data models are used, including columns, documents, key-value stores, graphs or multi-modal databases. MongoDB itself is a document-oriented data store based on the “Binary JSON” (BSON, pronounced “bison”) data format.

The NoSQL schema provides multiple features favourable for distributed, scalable environments such as big web applications are. NoSQL databases are generally simpler in design, are able to scale horizontally through sharding and provide greater operation speed in some settings. Also MongoDB incorporates a replication mechanism for high availability in which sets of at least two representations of data exist, one being the primary, the other ones being secondary replicas. In case of a failure of the primary replica, the remaining members perform a leader election in order to retain availability.

In our case we defined models consistent to the representation of data in our architecture. The models included an `user`, a `location`, a `companionrequest` and a `hotspot` model which in turn got converted into MongoDB collections. Via Mongoose we were able to perform operations on these collections by invoking methods on such a specified model. Further discussion of our model structure will be given in chapter 4.1.4.

Of course this setup did not only provide advantages but also yielded quite a lot of headaches during the project. We will be reviewing our issues with MongoDB, or more specifically Mongoose, in our chapter about the evaluation.

¹⁰ <https://www.mongodb.org/>

¹¹ <http://mongoosejs.com/>

4 Implementation

4.1 Backend

4.1.1 Architecture

In this section we will focus on the architecture and our decisions on backend side and its communication. We built a RESTful API which allows clients to communicate with the backend via the JSON format. The figure 3.1 shows what the architecture consisted of. We had two different client applications which were communicating with the backend and a federation provider, further backend specific information about the federation provider will follow in chapter 4.1.7. The backend persisted the data into a mongoDB database. The SNET department provided us a virtual machine at the address <http://piazza.snet.tu-berlin.de>, where we could setup our environment.

We developed the backend with Node.JS¹ with the purpose that the SNET department could use or integrate it later into other projects which are mostly written in Node.JS. For our web application we used the Express web framework which influenced us in the structure of how our application is built. Our file structure consisted of five main directories:

- **Controllers** are the specific implementation for an API endpoint request.
- **Routes** link an endpoint to a controller.
- **Models** are schemas for the persistent data in the database.
- **Middleware** takes care of our authentication.
- **Tests** are provided for some of the important controllers.

4.1.2 Controllers

We designed four controllers which are also part of our API endpoints to handle all requests to our API. `companionrequests` handles the creation and modification of a companion request. This means that a user wants to add a colleague to her/his friends list and she/he asks her/his friend for permission. If she/he accepts it, both parties are added to each others friends list. If she/he denies it, the requesting user will be notified about the changed status. The controller `hotspot` gives back all information about the defined hotspots like mensa or library. This includes GPS coordinates, companyUUID, major and minor of the Estimote Bluetooth Beacons contained in the area. For retrieving or modifying user specific information like her/his loca-

¹ <https://nodejs.org/en/>

tion, groups and settings the `users` controller is the handler for these kind of requests. The last controller is the `login` controller which on successful login returns some information about the logged in user.

4.1.3 Routes

Routes specifies a specific endpoint for a defined URI². Our routes link mostly with the same name to the controllers which we defined above. Since we have an RESTful API the expressjs framework also allows us to define GET,POST,PUT and UPDATE functionality. All endpoints are secured via our middleware which we will describe in section 4.1.5.

4.1.4 Models

Mongoose³ is a powerful plugin for nodejs which allows us to have an easier connection to the mongoDB database. For this we have to define schema's for our models. So we definded four schemas companionrequest, hotspot, location and user. But not every information have to be an extra collection in the mongoDB. We definded subdocuments for those information which dont need an extra collection. For example the beacons are always in a hotspot so there is no need for an extra collection.

4.1.5 Middleware

The middleware is provided by the cyclone projekt which handels the authentication and session management to the federation provider. If the user doesn't exist in our database the authentication middleware will add it with the informations which he gets from the federation provider. This includes the name, email and the ID which is also given from the authentication token of the user. We overloaded this middleware for the test section below. Which allows us to have a valid authentication against the database.

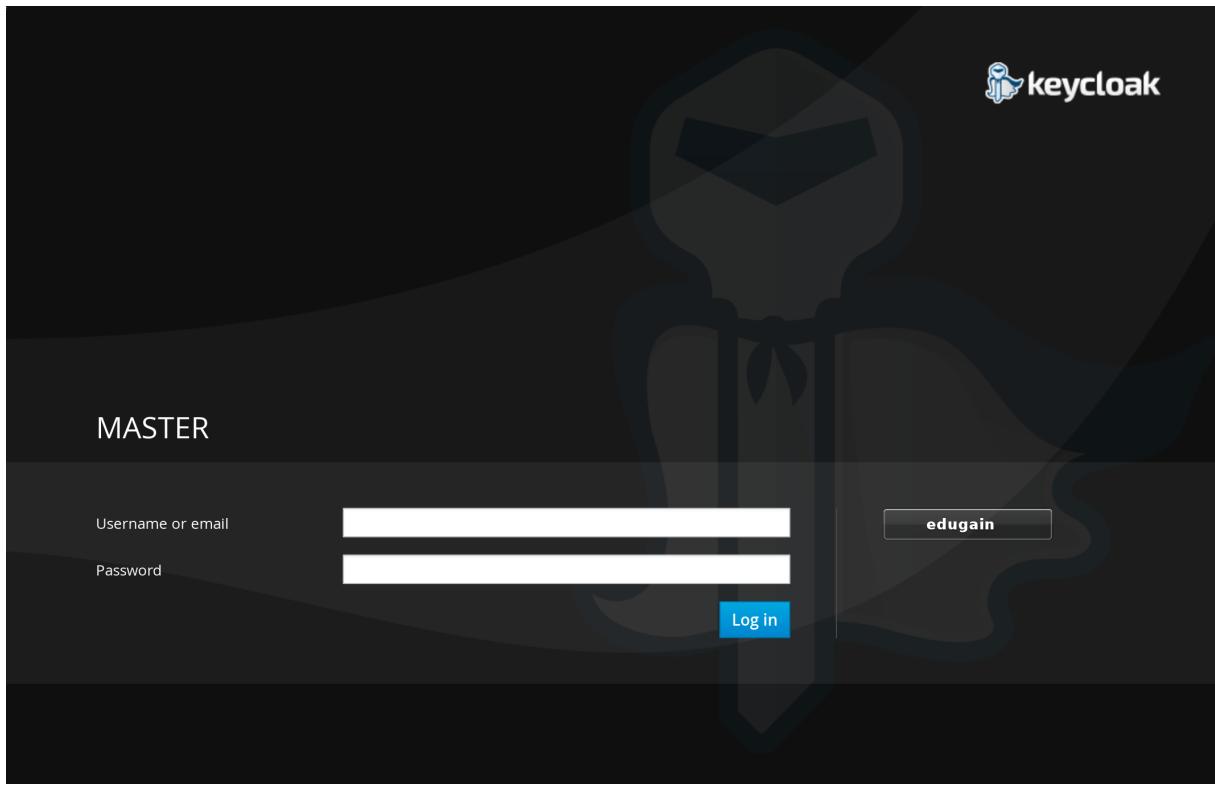
4.1.6 Test

- Server setup
- Docker deployment
- List of important API endpoints
- Tests

²URI=Uniform Resource Identifier

³<http://mongoosejs.com/>

4.1.7 CYCLONE Federation Provider



Login screen to CYCLONE Federation Provider.

4.2 Android

4.3 iOS

The iOS application provides the user with a map of the specific hotspot. For this project the example of indoor navigation inside the mensa has been implemented. The app shows two floors fo the mensa, together with additional informations about the Location.

4.3.1 Floorplans

In order to show positions of the user and friends with the exact coordinates on the map, the floorplan material is added as MKOverlay on top of the MKMapView provided by Apple. We use the PDF files provided by Studentenwerk Berlin for this project.

Apple also provides registered iOS developers with a framework to manage the mapping of x/y coordinates of the PDF file with the exact Latitude/Longitude real world coordinates of the map 4.1.

This framework has been used for this project to integrate the floorplan of the mensa. The GeoAnchor class provided by apple converts each corner of a PDF rectangle into an MKMap-

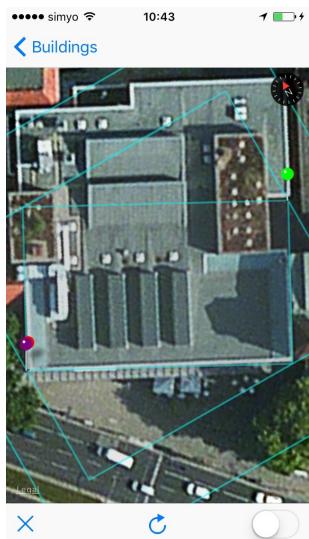


Figure 4.1: Share Position Mode



Figure 4.2: Synchronized Position Mode

Point. The collection of MKMapPoints is combined to MKPolygon which. MKPolygons are used to draw elements and annotations on top of a map. This technology is used to divide the map in multiple maptiles that can be added to the map 4.2.

4.3.2 Login

In order to use the application, the user has to login via Federation Provider, in order to get a security token which grants access to the application server.

The app handles the login process using a UIWebView. The login request will then be redirected to the Tubit login page 4.3, where the user finally can type in Tubit username and password. If the user credentials are correct, the webview gets redirected to the piazza application server.

If the WebView was able to access the application server successfully, the WebView closes and grants the user access to the application. The WebView can not be bypassed without a successful login via Federation Provider 4.4. Each request to the application server reopens the WebView again, if the used security token is invalid or if the user has been logged out.

4.3.3 Hotspots / Buildings

The Buildings View 4.5 is the first view the user sees after login. This view enlists all available Hotspots delivered by the Backend.

1. The reload button, calls the Backend to get a list of available hotspots. The list contains the hotspots and the available beacon and floor information.
2. The Tableview lists all available hotspots from the Backend. Each cell contains the name of a Hotspot.

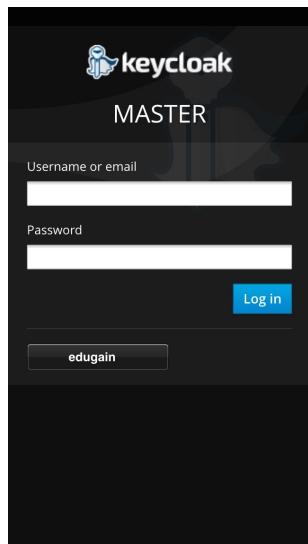


Figure 4.3: Federation Provider

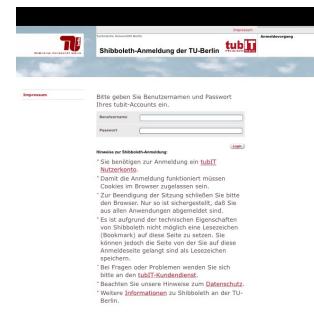


Figure 4.4: Tubit Login

3. The Application is divided into three contextual distinct parts: Buildings, Friends and Groups. This subdivision is implemented in a UITabBar which is the leading element of the application architecture.

4.3.4 Indoor Positioning

If the user taps on one of the Buildings enlisted in the Buildings View, the app opens respectively the corresponding floorplan in an MKMapView. Figure 4.7 shows the mensa floorplan on a large scale. This functionality was added additionally in order to increase the usability and accuracy of manual pin-pointing.

1. The the MapView shows the first floor of the mensa by default, if the Tubit-MSE api which provides the floor and building information is not available. In cases of availability of the Tubit-MSE api, the MapView shows the corresponding floor as soon as the MapView appears on the screen.
2. The manual pin pointing of the users position is triggered by a so called long press on the map which was additionally implemented. The long press is a distinct user-interaction mode. It is applied as solution for manual pin pointing on iOS in order to prevent the user from mistakenly share positions by inadvertently touching the map. After a long press is registered by the system, it drops a red pin on the position where the user pressed. This switches the MapView into a share position mode. In this mode the user can touch on any place on the map to easily edit or remove the pin, if the position to share is not correct.
3. The information panel is used to give the user the possibility to change the floor by pushing the plus (up) or minus(down) buttons. The Informationpanel also flips with an animation, signaling the user of the changed mode of the MapView. The flip animation is triggered as soon as the system successfully registeres a long press. In Share Position Mode the Information-Panel prompts the user to share the location.



Figure 4.5: List of available Hotspots



Figure 4.6: Access Location Request

4. The File-Button modally opens a settings view.

Figure 4.8 shows the MapView after a user has successfully shared the position either via Bluetooth Beacon, automatically using the Apple CLLocation Framework or manually via pin-pointing.

1. As soon as the backend has received the new location of the user successfully, the red pin is exchanged with a marker annotation. This marker shows the provided synchronized location of the user. The switch of the marker gives the user a hint, that his position is now synchronized with the server state. This marker is also used for the positions of friends, however the user position is highlighted with a star.
2. If the user taps on his own marker, it reveals an annotation view that includes the user name as well as a remove button that deletes the shared position from the server permanently.
3. The Information-Panel is now flipped back to Normal mode (if the user previously shared the position manually).

The Settings view (see Figure 4.9 and 4.10) provides additional options to indoor positioning and Information Annotations that are shown on the floorplan.

1. **Show all Friends** activates annotations for Friends in the same hotspot. (Note: Friends will only be shown on the map if the user has shared his own position before)
2. **Automatic Update** activates a routine that constantly requests the server for location updates of friends.
3. **Wi-Fi Indoor-Positioning** activates the CLLocation positioning. This will automatically update the user position while moving through the Building.
4. **Bluetooth-Positioning** activates location sharing via bluetooth beacons
5. **Show Beacons/Regions** activates annotations showing Beacons placed in the Hotspot

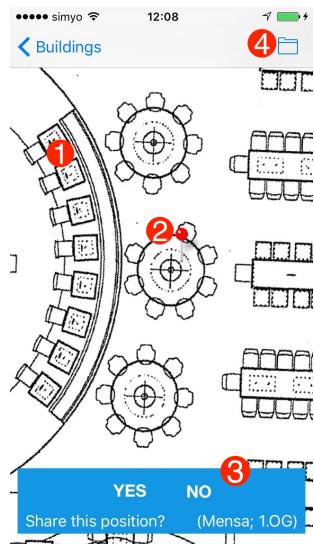


Figure 4.7: Share Position Mode



Figure 4.8: Synchronized Position Mode

6. Show Specific Hotspot Informations

activates annotations showing specific hotspot informations.

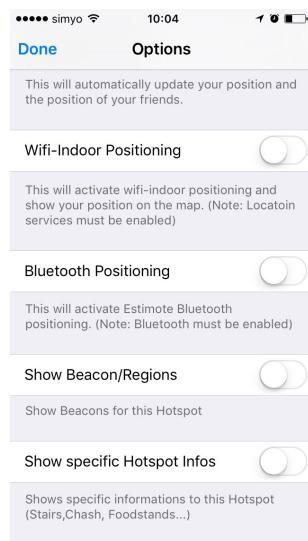
If the user activates **Show all Friends** on settings and has already shared the own position, the MapView adds Markers for each friend located in the same hotspot. The friend markers are equipped with Accuracy-Circles placed underneath the marker. The radius of the Accuracy-Circles decreases with the accuracy of the shared position. If a position is manually shared, no Accuracy-Circle will be shown. However, if the position is shared via Bluetooth or Wi-Fi, the Accuracy-Circle increases to show the region the friends position (see Figure 4.12).

If the user activates **Show Specific Hotspot Informations** on settings, the MapView will show the user specific Informations regarding the Hotspot such as Cashregister or Foodstands. This functionality has been added in order to improve the usability of the Floorplan and helps the user to orientate in the building (see Figure 4.11).

4.3.5 Friends

As soon as the user tabs on the All Friends button of the TabBar, the application shows the All my Friends List 4.13. This View shows either the friends of the user or opens companion requests that the user needs to accept in order to add a person into his friends list. Note that all available friends will be shown in this list regardless of their group affiliation.

1. The Reloadbutton is implemented in order to update the list of friends or new companion requests. In future work the server should be able to send remote notifications in order to inform the user of new companion requests.
2. The Plus button is implemented to open a new view 4.14 where the user can add new friends and send new companion requests.
3. The list view is managed by a UISegmentControl. The content is divided into available friends and available Companionrequests. These are shown by numbers. A tap on one

**Figure 4.9:** Share Position Mode**Figure 4.10:** Synchronized Position Mode

of the segments will either change the content of the TableView to friends or companion requests. The entry of the list contains the provided name of a friend.

As seen on 4.14, the app opens modally a new view that enables the user to add new friends.

1. The UITextField is used in order to enable the user to type in the email address of a friend. This request will be handed by the backend. The response of the backend about the success or fail of the request will be directly shown under the textField.

4.3.6 Groups

The backend enables the user to manage groups. Each user has a group named All friends, containing all of the users friends. Additionally the user may have different friends that can be added to own groups. These groups will be enlisted in this TableView as shown in 4.15.

1. The plus-button will open a new view modally that enables the user to create a new group (see 4.16).
2. This tableview shows all available groups of the user. Each cell contains of the Name of the group together with the GroupID as subtitle. For future work if the user taps a group, it will open a new list showing the members of this group. The user should also be able to add specific friends to a group as well as edit or delete a group.

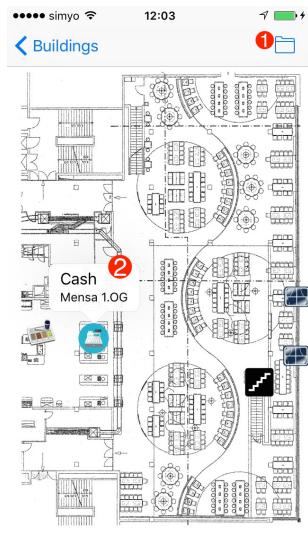


Figure 4.11: Share Position Mode

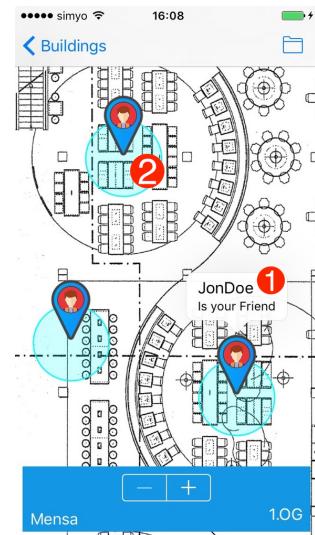


Figure 4.12: Synchronized Position Mode

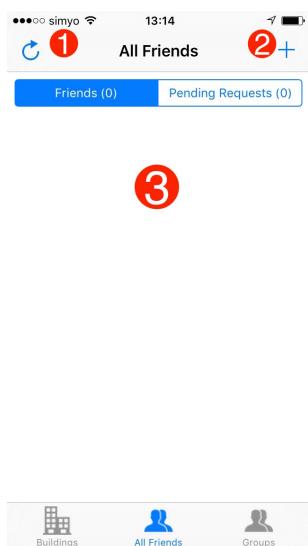


Figure 4.13: Available Friends and Open Requests

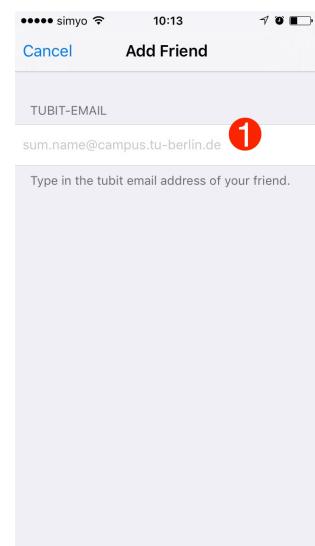


Figure 4.14: Add Friends



Figure 4.15: Available Groups

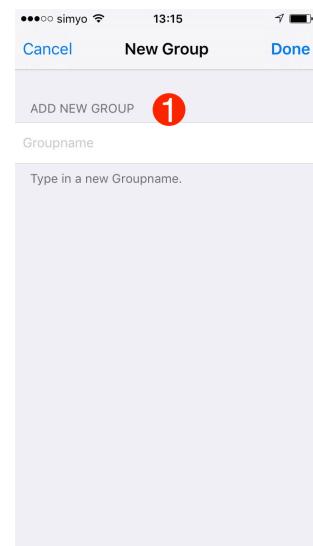


Figure 4.16: Add Group

5 Evaluation

Evaluate:

- What does work so far? What does not?
- What were the observed issues (see final presentation)?
- → MSE API, beacons, interplay server with clients, Node.JS, Mongoose (MongoDB), etc.

6 Conclusion

Conclusion:

- What did we do, how did it work out?
- Mention “team work issues”?
- Future work

6.1 Future Work

List of Tables

List of Figures

4.1	Share Position Mode	24
4.2	Synchronized Position Mode	24
4.3	Federation Provider	25
4.4	Tubit Login	25
4.5	List of available Hotspots	26
4.6	Access Location Request	26
4.7	Share Position Mode	27
4.8	Synchronized Position Mode	27
4.9	Share Position Mode	28
4.10	Synchronized Position Mode	28
4.11	Share Position Mode	29
4.12	Synchronized Position Mode	29
4.13	Available Friends and Open Requests	29
4.14	Add Friends	29
4.15	Available Groups	30
4.16	Add Group	30

Appendices

1 Tool for performing tests on the CISCO MSE API wrapper

```

1 package main
2
3 import (
4     "fmt"
5     "log"
6     "os"
7     "syscall"
8     "time"
9
10    "encoding/xml"
11    "io/ioutil"
12    "net/http"
13    "os/signal"
14 )
15
16 type WiFiInfo struct {
17     XMLName      xml.Name `xml:"Info"`
18     ChangedOn    string   `xml:"changedOn,attr"`
19     ConfidenceFactor float32 `xml:"confidenceFactor,attr"`
20     Building     string   `xml:"building,attr"`
21     Floor        string   `xml:"floor,attr"`
22     Network      string   `xml:"WLAN-Status,attr"`
23     UserName     string   `xml:"username,attr"`
24     Longitude    float64  `xml:"lon,attr"`
25     Latitude     float64  `xml:"lat,attr"`
26 }
27
28 type LogWiFiStruct struct {
29     Timestamp int    `json:"timestamp"`
30     Latitude  float64 `json:"latitude"`
31     Longitude float64 `json:"longitude"`
32     Building   string  `json:"building"`
33     Floor      string  `json:"floor"`
34 }
35
36 func handleUserExit(signalChannel chan os.Signal) {
37
38     for _ = range signalChannel {
39
40         // Define a useful result file name (format: "wifi-measurement-year-
41         // month-day-hour-minute-seconds.json")
42         fileName := fmt.Sprintf("wifi-measurement-%s.json", fileNameTime)
43
44         // Open measurement result file
45         fileResult, fileError := os.OpenFile(fileName, os.O_CREATE|os.O_RDWR|os.
46             O_APPEND, 0666)
47
48         if fileError != nil {
49             log.Fatal(fileError)
50         }
51
52         defer fileResult.Close()
53
54         // Write WiFiInfo struct to file
55         err := xml.NewEncoder(fileResult).Encode(wifiInfo)
56         if err != nil {
57             log.Fatal(err)
58         }
59
60         // Close file
61         fileResult.Close()
62
63         // Print message to log
64         log.Println("Measurement completed successfully!")
65     }
66 }

```

```

49
50     // Close open logging file
51     defer fileResult.Close()
52
53     log.Printf("\nWritten measurement values to file %s. Good bye.\n",
54             fileName)
55
56     os.Exit(0)
57 }
58
59 func main() {
60
61     // Put in here the API endpoint to the wifi location service.
62     const apiURL = "PUT THE API ENDPOINT IN HERE"
63
64     // Get time data
65     startTime := time.Now()
66     fileNameTime := startTime.Format("2006-1-2-3-4-5")
67
68     // Define a clean up channel
69     signalChannel := make(chan os.Signal)
70     signal.Notify(signalChannel, os.Interrupt, syscall.SIGTERM)
71     go handleUserExit(signalChannel)
72
73     log.Printf("Starting to measure WiFi API.\n")
74
75     for {
76
77         // Make a GET call on that URL
78         apiResp, apiError := http.Get(apiURL)
79
80         if apiError != nil {
81             log.Fatal(apiError)
82         }
83
84         // Read in all body content we got and close connection
85         xmlData, ioError := ioutil.ReadAll(apiResp.Body)
86         apiResp.Body.Close()
87
88         if ioError != nil {
89             log.Fatal(ioError)
90         }
91
92         // Our go struct representation of the tubIT XML
93         wInfo := WiFiInfo{}
94
95         // Parse received XML into struct
96         xmlError := xml.Unmarshal([]byte(xmlData), &wInfo)
97
98         if xmlError != nil {
99             log.Fatal("XML parsing error: %v.\n", xmlError)

```

```
100         }
101
102         // Get the current UNIX epoch timestamp for logging
103         timeResult := time.Now().Unix()
104
105         // Build up the log string
106         logResult := fmt.Sprintf("\t\t{\"timestamp\": %d, \"latitude\": \"%.\n15f\n\", \"longitude\": \"%.\n15f\", \"building\": \"%s\", \"floor\": \"%s\n\"},\n", timeResult, wInfo.Latitude, wInfo.Longitude, wInfo.Building,
107             wInfo.Floor)
108
109         // Write log string to opened file
110         fileResult.WriteString(logResult)
111
112         log.Printf("Received: \"long\": %v, \"lat\": %v.\n", wInfo.Longitude,
113             wInfo.Latitude)
114
115         // Let the execution wait for 2 seconds
116         time.Sleep(2 * time.Second)
117     }
118 }
```