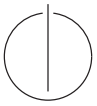


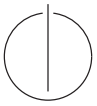
Technische Universität München

Department of Informatics

Master's Thesis in Informatics

A Self–Adapting User Interface for Smart Spaces

Andreas Hubel



Technische Universität München

Department of Informatics

Master's Thesis in Informatics

A Self–Adapting User Interface for Smart Spaces

Eine selbstanpassende Benutzerschnittstelle für Smart Spaces

Author	Andreas Hubel
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Dr. rer. nat. Marc-Oliver Pahl
Date	September 15, 2017

Informatik VIII
Chair of Network Architectures and Services

Abstract

This thesis provides the foundation for adaptive, state of the art, graphical user interfaces within the DS2OS project. The DS2OS project provides a platform independent middleware (VSL) for modelling, accessing and processing information of (smart) devices. It connects previous incompatible devices through one unified namespace with reusable services, distributed via a global exchange platform, with security in mind.

The UI prototype created within this thesis, is a floor plan based web app. Research shows that current state of the art JavaScript web application frameworks are Angular, Ember and React. To build this UI as flexible and extensible as the VSL itself, Ember and its 'Components' are chosen. The floor plan is implemented using the Leaflet web map library.

To model the spatial relationships ("Which device is in which room?"), set theory in 3D space is used via PostgreSQL/PostGIS as geodatabase. Thus, the spatial model is flexible enough to cover special cases, without becoming too complex for the default case. This model is exposed in the VSL via new geo service using GeoJSON as exchange format.

To introduce the reader to the problem domain and to gather requirements, the document is opened by an interview series with occupants of four different physical spaces. It's closed with quantitative and qualitative evaluations, including a six participant user study checking usability.

Further artefacts are: analysis of existing smart space UI using floor plans, requirements for modern smart space UIs, and a corresponding use case model.

Zusammenfassung

Diese Arbeit legt die Grundlagen für anpassbare, State of the Art, grafische Benutzeroberflächen im DS2OS-Projekt. Das DS2OS-Projekt stellt eine plattformunabhängige Middleware zu Verfügung (VSL); die Modellierung, Zugriff, und Verarbeitung von Informationen (intelligenter) Geräte ermöglicht. Es verbindet vorher inkompatible Geräte über einen einheitlichen Namensraum mit wiederverwendbaren Diensten, die über eine globale Austauschplattform verteilt werden, wobei trotz allem auf optimale Sicherheit geachtet wird.

Der im Rahmen dieser Arbeit entstandene UI-Prototyp, ist eine Gebäudeplan-basierte Web-App. Die Recherche ergibt, dass die aktuellen State of the Art JavaScript Web-Application-Frameworks Angular, Ember, und React sind. Um die Benutzeroberfläche so flexibel und erweiterbar wie das VSL selbst zu konstruieren, wird Ember und dessen 'Components' genutzt. Der Gebäudeplan wird mit der Leaflet Webkarten-Bibliothek umgesetzt.

Um die räumlichen Beziehungen zu modellieren ("Welches Gerät ist in welchem Raum?"), wird über PostgreSQL/PostGIS als Geodatenbank auf Mengenlehre im 3D-Raum zurückgegriffen. Dadurch ist das räumliche Modell flexibel genug, um Spezialfälle abzudecken, ohne für den Standardfall zu komplex zu werden. Dieses Modell wird im VSL über einen neuen Geo-Service bereitgestellt, wobei GeoJSON als Austauschformat verwendet wird.

Um den Leser in die Problemdomäne einzuführen und Anforderungen zu sammeln, beginnt das Dokument mit einer Interviewserie mit den Nutzern von vier verschiedenen physikalischen Orten. Es endet mit quantitativen und qualitativen Evaluierungen, darunter eine Usability-Nutzerstudie mit sechs Teilnehmern.

Weitere Artefakte sind: eine ausführliche Analyse bestehender Smart Space UI's mit Gebäudeplan, Anforderungen an moderne Smart Space UI's, und ein Use Case Modell.

Contents

Chapter 1

Introduction

When I mentioned this thesis's title "A Self–Adapting User Interface for Smart Spaces" the first question I typically got was "*What is a smart space?*". At our research project, the term smart space is a generalisation of the 'smart home' to not only include residential homes like a flats, detached homes, or multifamily residences; but also commercial buildings like offices, meeting rooms or lecture halls.

Every human interface can be regarded as UI: The wall-switch for the lights, the push buttons for the blinds, the thermostat for heating/ventilation, or the infra-red remote control for a TV or a projector.

Problems with today's smart space UIs are:

1. Solutions implemented for one space are not simple reusable in an other space.
   - API is not abstract enough: services use fixed devices addresses and not generic terms like 'living-room lamps'.
   - No or only insufficient exchange platforms: no app store, but forums or blogs with instructions or snippets to copy&paste, rather than easily installable software packages (apps).
2. Semantic relationships are not modelled as such, e.g. they are often only represented by menu structures. Thus computers do not know, which device is in which room.
3. Some UI apps are not platforms in-depended, e.g. ActiveX 'web UI' requiring a Windows desktop PC; or the devices UI app only exist for iOS but not for Android etc.
4. Users themselves can not adapt the UI, but need to call in (external) experts for changes.
5. Solutions are more expensive than they need to be: Instead of mounting Android tablets to the wall, five times as expensive touch screens with a severely restricted feature set are installed.

6. Vendors create silos for their devices:
    - UIs are distributed to different apps or mounted in different locations in a room.
    - System overlapping group actions (scenes) are not possible due to non-existent, proprietary or incompatible APIs.
7. New users have to find out which devices can be controlled and how these devices are labelled in the UI, e.g. “Where is lamp 3?”.

1.1 Document conventions

Following conventions are used throughout this document:

- Numbers in square brackets are source references, e.g. [1] refers to source 1; [3, p. 25] refers to page 25 of source 3; and [5, ch. 3] refers to chapter 3 of source 5. To dereference these source numbers, see the bibliography at the end of this document.
- Numbers in parentheses refer to others (sub)sections – for example (1.2.3). On odd pages, the current section number is printed in the top left corner.
- Angle brackets are used for requirements and use cases. <R.x> refers requirement x, c.f. requirement index in section 2.6. <U.y> refers Use case y, c.f. section 2.3.
- Starting with chapter 3, the requirements notation is extended as following: If a requirement is met its referenced with <R.x>✓, if not it is crossed out ~~<R.x>~~↯.
- Inside of VSL paths (2.2.1) angle brackets are used as parameter placeholders, e.g. *get /search/geometriesIn/<floor>*.
- Footnotes[1] are placed at the bottom of the page they are referenced from.

If you read a digital version of this document, all references mentioned above including URLs are hyper-links directing you to the corresponding pages.

[1] Like this example footnote.

1.2 Task definition and environment

Within this thesis, my task is to create the foundation and a prototype for a floor plan based user interface (UI) for smart spaces. The constraints are, that it is implemented as web app and that this UI should be as flexible as the underlying middle-ware. Its primary target platform are tablets, but it should be also usable on desktop PCs.

This work is part of the DS2OS (Distributed smart space orchestration system) research project[2]. The idea and theoretical background were introduced with the dissertation of Marc-Oliver Pahl [1]. Now a group of 5 to 20 students bring these concepts to life. The middle-ware (Virtual State Layer, VSL) provides an abstract representation for individual devices and their attributes, reusable services which can be exchanged between sites via a smart space store.

Figure 1.1 shows this work's area circled in red using the DS2OS context concept. This work includes creating an UI service and a 'geo' service. The geo service should provide the back-end features required to implement a floor plan based UI, like positioning of individual devices and reasoning in 3D space. Everything else is provided by other members of the project or is not part of this work.

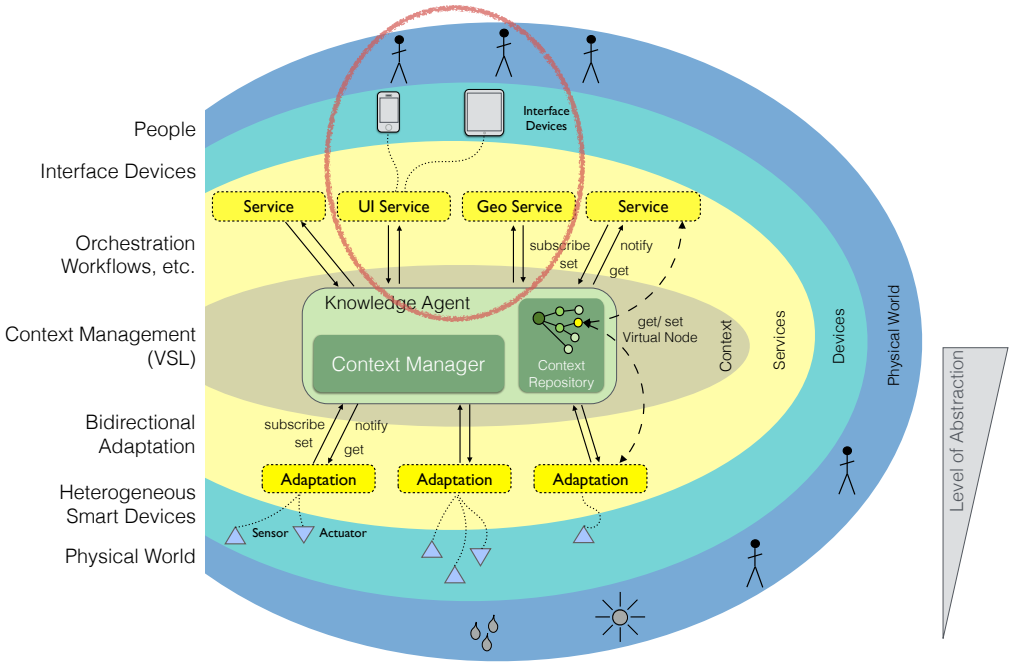


Figure 1.1: Context view of the VSL, with a circle around the area this work is focused at, based on [1, p. 183]

[2] http://ds2os.org

1.3 Thesis outline and methodology

The thesis is structured into seven chapters, the first of them being this introduction. The relations between the individual subsections were also put into a diagram, cf. Figure 1.2.

The next chapter – **Analysis** – presents challenges of the smart space UI domain, existing methodologies and available tools:

As the resulting system should work for every smart space, I analyse four existing spaces, their associated users and their current and future *usage scenarios* by performing interviews. The Virtual State Layer (*VSL*) concepts, terminology, data model and data access APIs are introduced. The scenarios from the interviews and VSL are combined, generalized, and *use case model* is derived.

Back-end: I analyse the previous approach to handle geolocations in the VSL, and draw first conclusions for the future geo model. I introduce geodatabases, their features for spatial data, as well as associated geometry types and serialisation formats.

Front-end: The terms user interface and usability are formally introduced and defined. Existing floor plan representations and UIs are investigated: Classic architecture floor plans; three web map services presenting indoor data; and three floor plan views actually designed for smart spaces. The origin and background of apps, tablets, and their operation systems are explained. Different ways to create and distribute apps for multiple platforms are introduced. I introduce human interface guidelines as way to build apps with good UI and usability. Different ways of optimizing UIs for spaces and its users are discussed. The subsection closes with building blocks: JavaScript as environment, the mindset of its developers, and the libraries and components for this project's use case.

Finally, all artefacts of the analysis chapter are compiled into a list of *requirements*.

Chapter 3 presents other products and prototypes (**Related work**), and compares their visualisation approaches, data model, and vendor independence. We take a closely look at the openHAB project, ABB-free@home, and Apple's HomeKit. The merged results are presented with an comparison table.

Chapter 4 presents this thesis's chosen approach. I develop a system architecture and **design** to resolve the challenges resulting from the requirements influenced by ideas from related work. Again, I divide into back-end and front-end. Back-end: Definition of the *geo data model* and the *geo service* API. Front-end: Selection of *libraries*, specifically the web application framework, as well as the web map library. Definition of the front end *software architecture* as well as the development of the *graphic UI design*.

Chapter 5 reports relevant **implementation** details and gives examples from the code base. The structure is similar to previous chapter: First the back-end with focus on the *database* schema and *geo service* features. For the front-end: Screen shots and textual description of the web app prototype, the interaction with the back-end, and implementation of UI extensions.

Chapter 6 **evaluates** how the goals and requirements stated during analysis are fulfilled by the implementation. Depending on requirement, different procedures are necessary: Some functional requirements and constraints can simply be *checked of*, others require *measurement* setups or tests with actual *users*.

Finally the document **concludes** with an round-up of the results and outlines *future work* in chapter 7.

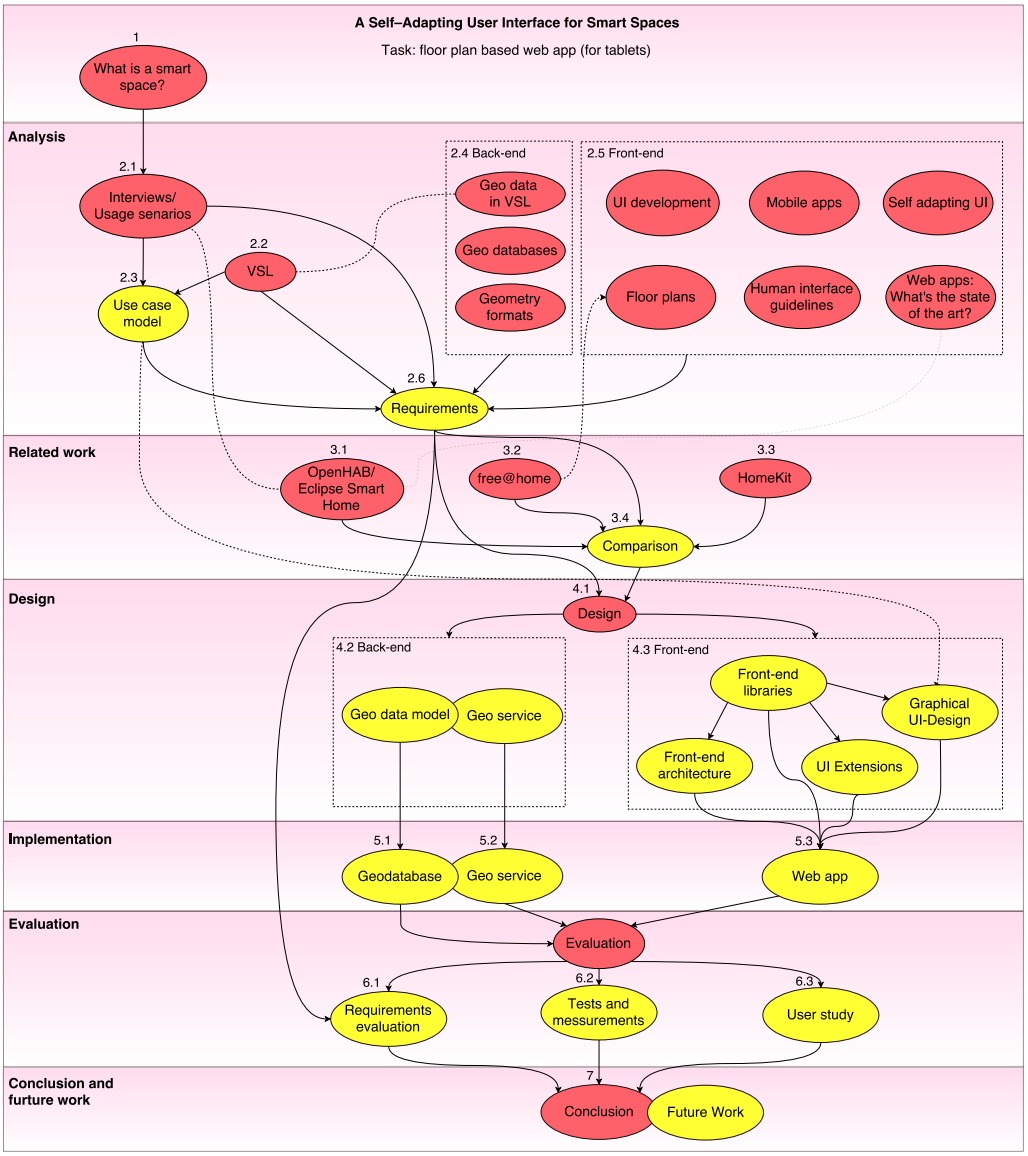


Figure 1.2: Structure of this document

Chapter 2

Analysis

Brief summary:

The goal of this work is to provide the foundation and a prototype of a floor plan based user interface for smart spaces implemented as web app. It's target usage devices are tablets and PCs.

This chapter introduces the reader to the problem domain. The first two sections are about a user survey and provide current and future usage scenarios for smart spaces. These scenarios are then used to build a use case model. In the other part of this chapter I work out the relevant issues and determine which problems exist, separated by back- and front-end. I look into the available methodologies and tools and search for state-of-the-art technology addressing the research problems. Finally the chapter closes with a list of requirements.

Extended summary:

The resulting system of this thesis should work for every smart space, whether it is an apartment, a house, a workplace or an other shared space. Therefore I analyse each of these spaces, the associated users and their current and future usage scenarios in section 2.1. I select four different spaces and do several formal and informal interviews. The interviewees use the systems eQ-3 FS20, openHAB, Crestron DigitalMedia, and KNX.

In section 2.2, the Virtual State Layer (VSL) concepts, terminology, data model and data access APIs are introduced. Through visionary scenarios, I introduce the main idea for this thesis: To build an user interface which is as dynamic as the VSL data model.

Section 2.3 combines and generalizes the scenarios and I derive a use case model. The use-cases are split into daily operation of devices (switch light on); meta data management (device locations, floor plan, users and their rights); service management and development; and system debugging.

Section 2.4 deduces the necessary adjustments to enable the VSL back-end to answer questions like “Which lamps are in the living room?”. To achieve this goal, the back-end needs to understand what a living room is and which entities are a lamp. I draw first conclusions for the building model from the spaces descried in section 2.1. To solve the resulting problems, it makes sense to rely on existing components. Therefore I introduce geodatabases and their features for spatial data.

The last big section (2.5) finally deals with the actual user interface. The terms user interface and usability are formally introduced, defined and explained. I discuss different ways of optimizing UIs for spaces and its users, explaining where apps, tablets, and their operation systems originate from. I further explain the different ways how to create and distribute apps for multiple platforms. I investigate if there are any rules one can follow to build a web app with good UI and usability and conclude that the DS2OS project might benefit from creating own guideline documents. The section closes with the building blocks for my web app, especially JavaScript as environment, the mindset of its developers, and the libraries and components for the project’s use case.

I close the chapter with a summary of the compiled requirements as a list (section 2.6). The individual requirements are referenced with <R.x> through the whole chapter.

2.1 Problem domain: smart spaces

This section further introduces the reader to the problem domain by collecting requirements and usage scenarios for smart spaces. Typical practices to collect requirements and usage scenarios include interviews, questionnaires, user observation, workshops, brainstorming, use cases, role playing and prototyping [2]. As I am to construct a research prototype, there are no real users yet. Nevertheless I can ask existing and potential smart space users what kind of requirements they come up with. As this prototype should work for every smart space, whether it is an apartment, a house, a workplace or a other shared space; I need to interview at least one person for each one of these space types. The next subsection describes the methodology of these interviews, the remaining ones are about the individual spaces. Table 2.1 gives an overview about the chosen spaces and interviewees.

	private		**commercial**		
	single apartment Munich	**single house Augsburg**	**Hackerspace Bamberg**	**University building Garching**	
interviewee	IT project manager	IT specialist	web developer	head of multimedia group	researcher
interviewee age	~40 years	~25 years	~25 years	~50 years	~30 years
number of space users	normally 1, sometimes guests	normally 1, maximal 5	association has 48 members, in average 8-10 present	~765 employees, with students ~5064 users	
number of rooms	4	~20	7	~700	
structure	living room with kitchen, work- and bedroom, bathroom, attic	two-storey house with basement. per floor: living room, kitchen, bathroom, bedroom, guest rooms, winter garden, foyer, laundry, garage	ground floor two main rooms: lounge and hackcenter, workshop, project room, CNC room, kitchen, WC	592 offices, 40 - 50 groups with meeting, seminar and lab rooms 4 bigger lecture rooms 3 lecture halls	
smart space implemented?	since 12 years eQ-3/ELV FS20,	in planning phase with custom hardware	OpenHAB 1 with custom hardware	partial (lecture halls, corridors) Crestron DigitalMedia, KNX	

Table 2.1: Overview about the chosen spaces and their related interviewees

2.1.1 Interview preparations

Before doing the actual interviews some preparation is needed. [2] [3]

To structure the individual interviews I prepared a mind map, which was then transformed to a table to be extended with the answers and edited with a spreadsheet program. The first three columns on the left are used for the areas, topics, questions together with expected and example answers. Each topic/question has it's own row. The other columns from left to right are for the individual spaces and interviewed people. The order of the areas and rows was changed after the first two interviews.

The first section is meta information: The name and occupation of the interviewees, the smart space type, and the kinds of devices they want to control in their space (desktop

pc, notebook, smartphone, tablet computer, smartwatch). The last two meta questions are about the number of users and the structure of the space, e.g. which rooms it has.

The next sections are about existing and planned systems, separated into sensors, UI, and others. The first two interviews showed that this separation did not really work. In the following interviews the structure is topic based.

The idea of these sections is to give the interviewee a sense of what kind of devices a smart space could actually consist of. For example, in a radio show[1] where the host and his guest talked for an hour about the guest's Wi-Fi presence-based heating automation system, the guest was somehow confused when the host asked if he also wanted to control his roller blinds.

The system inputs asked for included the opening state of windows (reed contact), motion- or presence- or brightness-sensors, cameras, smoke detectors, indoor temperature and humidity sensors, outdoor wind sensors, and brightness sensors. I also asked for the location of meters (power, water, gas) or dedicated heat cost allocator devices.

The systems we spoke about were lighting (intensity and colour changeable); heating, ventilating, and air conditioning (HVAC); audio (loudspeakers, stereos, radios with and without Wi-Fi) and video (TV sets, other screens, projectors); AV receivers, amplifiers, and controllers; communication (telephone, door intercom); and security (fire, burglary, access control).

I additional asked how the communication between individual users is performed, e.g. are there electronic mailing lists or is a chat or instant messaging system commonly used (IRC, Whats App, Facebook)? Questions asked regarding attendance detection: Is there a calender system? Does everybody have a smartphone? Is there an obligatory time and attendance system?

The UI list consists of traditional light switches, control panels (e.g. touch screens or fixed plates with building plan and hardware buttons), or desktop PC software.

The questions regarding UI in general were:

- Who is currently using which interface?
- Who creates the automation rules?
- Is there an existing building management system?
- Where is need for improvement?
- Is a combined all-in-one interface/app possible or meaningful?
- Is a building plan useful or not?
- Should there be a graphical automation rule editor, or would the interviewee prefer a scripting language to orchestrate everything?

All interviews were done in spring 2015 and recorded in separated audio tracks for

[1] `https://cocoapulver.de/2015/02/09/cp008-hausautomation-mit-intelligenz/`

interviewer and interviewee. In most cases they were done at the corresponding space. The post-processing included removing pauses and adding chapters, so individual statements can be located more easily. I did not create a full transcript, like it is done in social science [3]. Some recordings are available online at `http://andreas-hubel.de/serie-smart-spaces/`.

The rest of this section presents the results, grouped by each space.

2.1.2 Single apartment Munich

The interviewee lives alone in a typical urban three room apparent and has built an homogeneous automation system in his home over the last 12 years. His apartment consists of four areas: living room with integrated kitchen, combined work- and bedroom, bathroom, and a storage room in the attic. Compare Figure 2.1 – the entrance door is at bottom middle, the attic is not depicted.

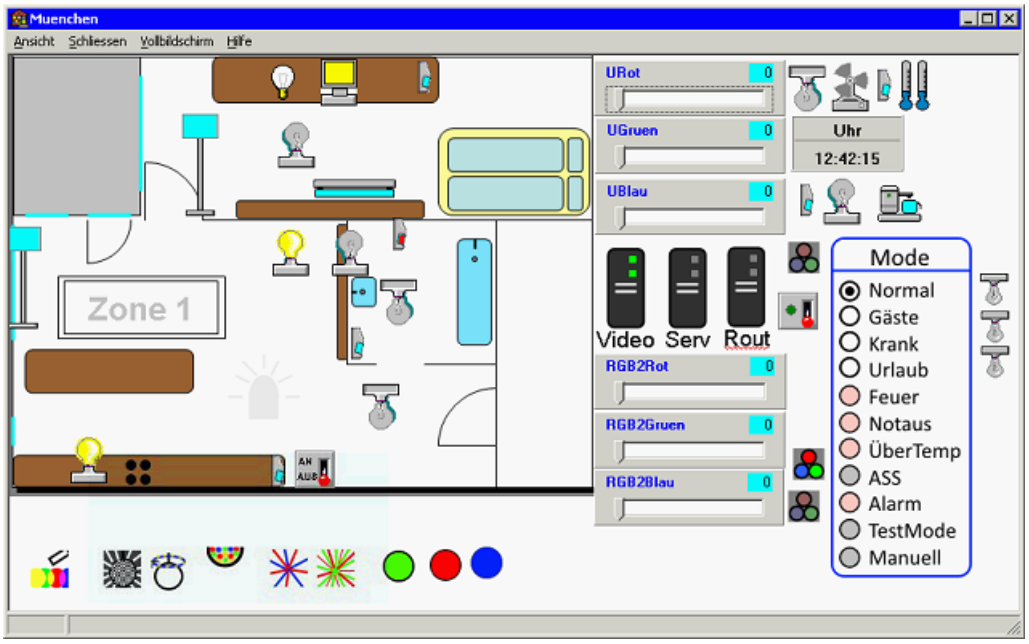


Figure 2.1: FS20 Windows UI with floor-plan customized and extended by owner

He uses only off-the-shelf components from one manufacturer. The unidirectional system is called FS20 and is developed by eQ-3. It is distributed in Germany by Conrad Electronic SE and ELV Elektronik AG.

His typical daily **scenario**:

> The owner comes home from work and opens his door by key. If it is dark outside (based on the time) his home turns the lights on. Then he has to

> authenticate himself by pressing buttons on a wall mounted remote control to disarm the burglar alarm.
>
> He goes to his desk in the second room and authenticates himself by password. This activates the power for the PC displays and other components, which would otherwise be on the whole time.
>
> When it is time, his home reminds him to go to bed by turning on a lava lamp.

When he gets up in the morning, the way to the bathroom passes through the living room. In summer the bathroom is lit by three lamps instead of just one, as the natural illumination in bathroom is darker than the living room lit by the sun.

Although present everywhere, motion detectors only manage the lighting in the hall area and in the bathroom. In the living room and bedroom, the lighting is intentionally controlled manually.

In the areas where the lights are not controlled automatically, he uses battery driven remote controls: either unlabelled four channel wall-mounted push buttons (Figure 2.2a) or one of his portable ones (Figure 2.2b and 2.2c).

The software GUI shown in Figure 2.1, is actually quite rarely used: “The apartment takes care of everything. You need the graphical interface only when something does not work or a special situation arises.” <R.13>

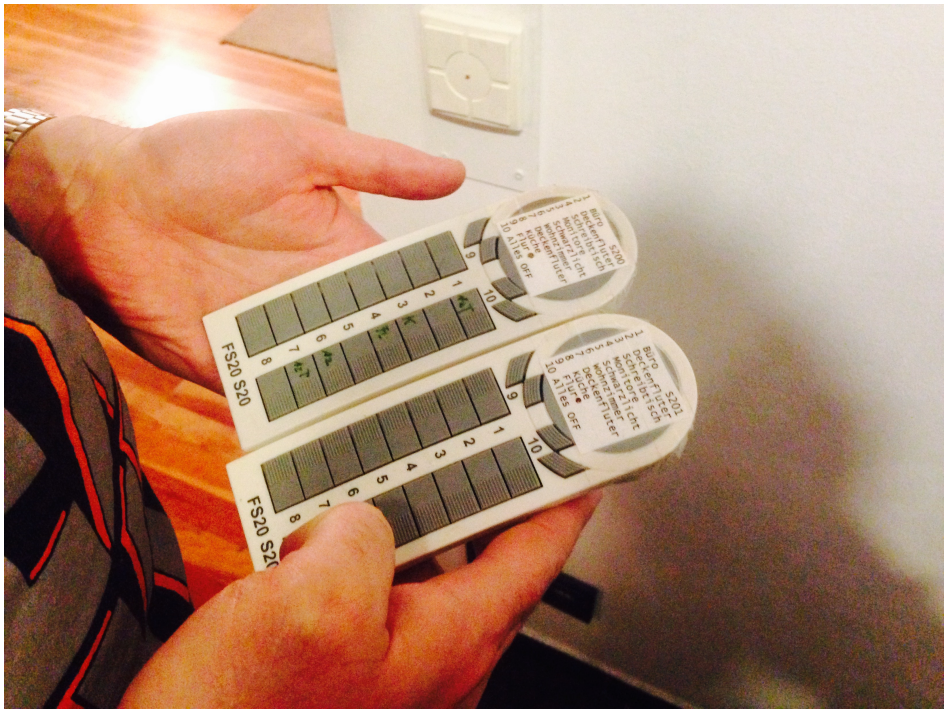
To use this GUI from a mobile device he uses the desktop sharing software TeamViewer because he does not want to manage and configure a second user interface. Since he switched from light bulbs to LEDs, the brightness is controlled by the number of switched-on lamps and no longer by dimming.

According to himself, his approach would not work when he would share his home with other people. When guests use the bathroom (see Figure 2.1) the light enables automatically when they step into the room. But instead they try to find out which of the four buttons beside the bathroom door (Figure 2.2a) they have to press. Unfortunately one of these buttons is central off. Quote: “I tell my guests: ‘Do not push anything. The system will get mixed up, you get mixed up.’ ”

All overnight guests get an own bedside lamp having absolute control via a cord switch.

(a) wall-mounted remote at bathroom door

(b) fixed remote at entrance



(c) portable remotes with labels

Figure 2.2: FS20 Hardware UI devices

2.1.3 Single house Augsburg

The second interview revolved around a two family house, which currently only has one inhabitant. The home automation system is still in the planning phase. In the default case, when he comes home, he does not want to touch any light switch as illustrated by following visionary **scenario**:

> Because he visited a customer in a distant city, the only inhabitant of the house was a few days out. On the way from the airport to his home – drive duration 2 hours, the oil heater turns on and warms up the ground floor radiators. The upper floor apartment is currently not in use and therefore remains cold.
>
> When the car is one kilometre away, the driveway gate opens. When he approaches his property, the garage door opens just in time, no waiting time occurs. When the ignition is turned off, the garage door closes again. He opens the front door with his house key and the hallway light activates automatically. Upon entering the dining room, the light is automatically turned on and the hallway light is turned off. Together with the light, the music or radio show he was listening to in the car travels with him from room to room.

Today, his home does not know in which room he is and the transmitting range of the gate remote control is not far enough.

At the time of the interview, he was planing to design and build an custom printed circuit board (PCB) which should contain a passive infra-red sensor for motion detection, temperature, relays and several general purpose input/output (GPIO) pins. This PCBs are then deployed in every room above the door. The main light and the light switch are directly connected to it. For security reasons all communication should go over wires and not via radio. The system should be hosted locally in the house and no data should be stored in an internet cloud <R.24>. He explicitly does not want an electronic lock to ensure that he can always get into his apartment, even if he has totally messed up his smart home configuration.

Main **motivations**:

- cost and risk reduction (e.g. forgotten basement lamp: energy and fire safety)
- increased comfort: controlling radiators manually currently makes up quite a lot of work
- to not rely on feeling but concrete data. Example: Should the blinds stay open to let the sun heat up the building as it is a sunny day? Close the roller blinds in the evening to prevent fast cool down?

As he wants to control everything automatically, why not remove all UI like light

switches? He answered that light switches are still needed, because there are always exceptions: For example, when he is playing a board game together with guests and they would turn on a film in parallel. When the projector has been turned on, the light would dim and roller blinds would be closed. But on the table the game taking place, there is still a need for light. Therefore some UI is needed to turn on these spots above the table. In the best way, without a walk to the light switches on the wall.

Would it make sense to display a floor plan, or would a hierarchical menu be more useful? He said he personally would manage a hierarchical menu better, presumably. But only because we would know where to search for specific items. His visitors would probably cope better with a spatial illustration on a building plan.

Regarding the **orchestration** rule editor, he can not imagine that a graphical editor would work for him. His orchestration ideas:

- House should wake him on the basis of important dates on his calendar, e.g. by opening the shutters, enabling all lights and playing music in the room in which he fell asleep, whether this was his bedroom or the living room, until he finally awakes.
- Washing machine starts one hour before arrival, so the laundry is not laying wet in the machine for several days until he returns. Of course, the laundry has to be placed in the machine before departure.

Other notable pieces: Each person, who stays longer than one day, will get an own account in his wireless guest network. He plans to order a digital electricity meter from his power utility.

2.1.4 Hackerspace Bamberg

This subsection is about a community-operated workspace with about 50 members. In the evenings, in average 8–10 persons are present in the space at the same time. The starting point to build their space automation was the problem of turning off all devices when the last member leaves the building. Quoting the interviewee:

> "With the growing number of projects, the switching on/off options were becoming more exciting and varying:
>
> When you where the last one to leave the space, you had to turn off fuse A, but not fuse B, as that's the refrigerator. [...] There were switchable power strips everywhere: Some had to be switched off, to turn down the audio amplifier, others – like the one the Raspberry Pi was connected to – must not be switched off by no means. In the end, even for those who created these projects, it was so confusing what they had to switch off and what not.
>
> All of our 50 members have closing authority and each of them could potentially be the last person to leave the building.
>
> Our first solution was to write a manual. It was three pages long, causing it barely read.
>
> Eventually we realized: We need a 'shutdown button' per room. When you leave a room today, you can switch off all lights with one button. Although there is a global 'shutdown button', users prefer to check visually if everything has been turned off per room."

The interviewee made me aware of smartwatches as possible UI, sunrise and sunset are easier obtained via internet data sources and not via outdoor brightness sensors. They have an AV receiver with Ethernet port and plan to build a decentralized audio routing setup. They use internet relay chat (IRC) and mailing lists for communication. The number of persons present is detected with a voluntary to use system, based on MAC addresses of personal laptops and smartphones. A signal when the dishwasher is done and can be emptied would be nice in future.

The **motivation** to introduce a home automation system was the central off switch (per room), but also the prospect to reduce power consumption and to reduce the fire likelihood by disabling soldering irons.

The most used interface in the two main rooms is a dedicated android tablet fixed to the wall of each room, see Figure 2.3. This tablet shows the menu of this room, implemented with OpenHAB 1.0. Nearly nobody uses their own smartphone to control the room, instead the tablet is preferred <R.8> <R.10>.

Figure 2.3: HABDroid on a wall mounted Android tablet in Backspace Bamberg [https://www.hackerspace-bamberg.de/Datei:Tablet.jpg]

To the question whether separate apps compared to an all-in-one app would be preferable the interviewee emphasized vendor independent solutions and did not like the idea of switching apps for different devices. He pointed out that a 2D building plan would have some problems with their shelf lights, where each bay can be set to a different colour <R.5>. For **orchestration** he personally preferred a scripting language over a graphical editor, but assumed that end users would need a graphical version. Their most advanced rule-set is a 'scavenging alarm': It reminds the users of the space to clean up the space 8 to 14 days after the last alarm, when the door is not locked and no one is watching a film. The alarm is only issued between 16:00 and 23:00 when there are at least four members present for more than one hour, see Listing 1. Another rule decreases the audio volume level when somebody rings on the door.

```
var Number alarmNotBeforeHour = 16, var Number alarmNotAfterHour = 23
var Number minimumIntervalDays = 8, var Number maximumIntervalDays = 14
var Number minimumMembersPresent = 4

rule "Check for cleanup alarm"
when
    Time cron "0 0/23 * * * ?" /* Check every 23 minutes */
then
    // We need to convert it to yodatime before using it with historicState or changedSince
    val lastCleanupJodaTime = new DateTime((lastCleanupAlarm.state as DateTimeType).calendar.timeInMillis)
    val randomValue = Math::random
    logInfo("cleanupAlarm", "Checking for conditions for cleanup")
    if(cleanupAlarmEnable.state == ON &&
        // Only if the door is not locked (which may indicate sleeping members) and
        doorLock.state == OPEN &&
        // no one is watching a movie (do not want to interrupt people)
        projector.state == OFF &&
        // The memberCount should be above a sane value, because no one want's to clean up alone.
        memberCount.state >= minimumMembersPresent &&
        // Also check if the members are at least an hour here, so one can just sit and relax for a while.
        memberCount.historicState(now.minusHours(1)).state >= minimumMembersPresent &&
        now.getHourOfDay() >= alarmNotBeforeHour && now.getHourOfDay() <= alarmNotAfterHour {
        logInfo("cleanupAlarm", "Criterias met, checking for last alarm + random")

        // Also check if there are different members than the last time
        if(!cleanupAlarm.changedSince(now.minusDays(minimumIntervalDays)) &&
            memberNames.state != memberNames.historicState(lastCleanupJodaTime).state &&
            randomValue > 0.9) {
              sendCommand(cleanupAlarm, ON);
        // No matter what, maximumIntervalDays is enough. We have to clean up. Seriously
        } else if(!cleanupAlarm.changedSince(now.minusDays(maximumIntervalDays))) {
            sendCommand(cleanupAlarm, ON);
        }
    }
end
```

Listing 1: Orchestration in Hackerspace Bamberg via openHAB rule: scavenging alarm

2.1.5 University building Garching

The last building in question is the home of the faculties of mathematics and informatics of the Technische Universität München. There work about 765 employees with an office (head count, without student research assistants), and up to 5000 students[2]. The building is made up of four to five floors, has 592 rooms classified as office and a usable floor space of about 28.000 m^2.

The 'smartness' of individual building areas is inhomogeneous. When moving into the building in 2004 all was uniform, but with time extensions and upgrades were made.

[2] Official numbers as of end of 2013, source: https://www.tum.de/fileadmin/w00bfo/www/TUM_in_Zahlen/TUM_in_Zahlen__2013_WEB.pdf

Figure 2.4: Home of the faculties of mathematics and informatics of the Technische Universität München from Uli Benz (TUM) https://mediatum.ub.tum.de/node?id=614549

The main auditoriums are quite automated and were last updated in 2014, whereas the seminar/meeting room of a single chair/group is comparatively simple equipped. The KNX field bus terminates at the utility room of each section per floor and is not available in the individual offices. Thus only the large lecture halls, and master control functions such as lighting in the hallways, blinds per floor and façade are controllable via KNX. Besides there are two other building wide control networks: one for the air cooling units and another one for the RFID locking system (based on RS-422). Both are typically deployed in rooms larger than an office.

This building distinguishes significantly from the spaces presented in the other sections, both in size and number of users. Therefore a single person can not describe all requirements for all stakeholders. It's necessary to cluster the various user groups and consult them separately. However, there are not always hard boundaries between groups. I split the responsibilities into different roles, where of course an individual can hold multiple ones:

- Occupant: People having a key for at least one room, e.g. staff, students who work paid part-time or writing their thesis (like myself).
- Lecturer: in the auditorium: researchers, including professors, in seminar room additionally students – whether as a tutor or presenting their work.
- Support/Operator: People which are called e.g. by phone when something is not working.

The rest of this section is structured as follows: First I describe **today's** daily routine of each role in form of scenarios, then **future** usage scenarios. The section closes with background information and a few anecdotes.

Occupant

Occupant enters the building by one of the entrances. He goes through the inside courtyard (local alias 'Magistrale') to the stairwell to his group's area ('Finger'). Passing the kitchen he finally opens his office door. If it is too dark or no colleagues are there yet, he uses the light switch. He sits down at his desk and starts with his work.

Lecturer (in lecture hall)

Through a motion detector in the desk area the system determines that the lecturer has arrived. She touches the screen on the speaker's desk and the audio equipment in the room is activated. The lecturer typically either uses the blackboard or the projectors. Let's assume she presents with her laptop: She connects the computer via VGA or HDMI to the outlets on the desk. Now she has to decide which of the projectors she wants to use. As explained by a leaflet fixed to the desk, she has to select the outlet she uses as source and selects one or multiple sources by touching the corresponding projector buttons within five seconds, compare Figure 2.5. The selected projectors are switched on and the corresponding projection screens descend. If the projection is difficult to read, she has to close the window shutters or regulate the lighting – but only in the front seating rows so that students in the back do not fall asleep. The lecture recording is automatically started based on the schedule of the room.

When the lecture is finished, she shuts down the system with the off button in the top right (Figure 2.5). All equipment is switched off, shutters, projection screens and blackboards return to their initial position. Lights are turned down so only the passageways are lit. When she leaves the room, she can fully darken the room with a classic wall push-switch.

Lecturer (in seminar/meeting room)

When the lecturer arrives he has to power on the projector manually with a tethered infra-red remote and select VGA or HDMI as input. To control the lights, he has to go to the entrance door on the other side of the room. To reduce the audio noise level, it sometimes makes sense to turn off the air conditioning/circulation unit during the lecture. While there is a built-in mechanical ventilation, during a longer usage period a forced ventilation by opening windows in a break can be helpful.

It can take 3-10 minutes, until a new computer communicates correctly with the projector. Therefore in sessions with multiple lectures, presentation slides are collected in advance and played from a dedicated computer of the chair.

Operator, Technical support staff

The daily routine is like a regular Occupant, but they have to login/logout to time recording system with the locking system RFID token as they are non-academic staff.

Room 0.08.15 calls via phone: The notebook is connected but the projector only shows a black screen. Operator connects to rooms media system, sees lecturer has occidentally muted the projector, unmutes it and tells lecturer how to handle this in future.

Other lecturer calls from off-site campus extension: He needs access to a seminar room. Operator opens floor plan, searches corresponding door and activates release buzzer.

Receives mail from projector: air filter or lamp has to be exchanged. Schedules cleaning/replacement for the next morning.

2.1.5.1 Affordable vision seminar/meeting room

When the lecturer arrives he connects the presentation notebook via VGA or HDMI. The notebook recognises in which room it is via the projectors serial number (communicated via the EDID back-channel part of the VGA/HDMI connector). The software opens a UI window allowing the lecturer to select which kind of presentation he wants to hold. The blinds, lighting and ventilation is set to the corresponding state. He can intervene manually via the same UI without the need to search for the remote or to go to the entrance door on the other side of the room. When there is a break during the session, the UI reminds the lecturer to open the windows to refresh the air via a desktop notification.

2.1.5.2 Vision auditorium

Through a motion detector in the desk area the system determined that the lecturer has arrived. The touch display lights up and welcomes the lecturer by her name. (Through the Campus Management System the auditorium is aware which lecturer and which lecture is here today.)

She unlocks the touch screen with her university employee RFID card and the auditorium adapts itself based on their preferences: the projection screens or blackboard are positioned, the audio EQ profile optimized for her voice is loaded to the audio processing unit / amplifier and the corresponding projectors are turned on.

45 minutes after the lecture started the speaker is reminded via a flashing message on the touch display to take a break. Depending on the room the system turns up ventilation or reminds to open the windows.

At the end the system asks the lecturer how many participants actually were there today. This number is correlated with the Wi-Fi users and course registrations from the Campus Management System. This allows the operator to move future lectures of this course to another room if it is too large for the actual number of participants or another lecture in a smaller room is surprisingly crowded.

2.1.5.3 Affordable vision office

The office optimizes the lighting for people and plants by controlling lamps and blinds automatically, depending if someone is in the room and how bright it is outside. The system alerts the occupant to open the window when the air is stale, as detected via a humidity or CO2 sensor.

2.1.5.4 Background information and anecdotes

In summer, every day at two o'clock the outdoor vents in my office close. I can not stop them, my local control buttons do not work any more. After 60 seconds the blinds rotate about 30 degrees and the sunlight gets through again. The local control buttons can be used again to undo this useless automation. <R.27>

The media control systems in the main auditoriums were upgraded in 2014. The previous system was installed during construction of the building in 2004. The new Crestron "DigitalMedia 2-series" system controls not only lighting and blinds but additionally projectors and audio via a uniform user interface (compare Figure 2.5). According to the local technician, managing this new system is still kind of a hassle: Every change to the interface has to be compiled to a new binary which needs to be uploaded onto the system. A full re-upload of the configuration of one system takes about 15-25 minutes and requires usage of multiple Windows programs – SIMPL Windows for logic/processor configuration, VisionTools Pro-e for UI design, Crestron Toolbox for uploading firmware and configuration. For more insight into the process see Crestron 2-Series Control Systems Reference Guide[3] or video tutorial from third-party programmer[4]. This tool chain can also generate Windows executables and ActiveX components which only work with Microsoft's web browser. (ActiveX is outdated technology.) According to the Crestron web pages the newer generation systems also have an web interface, so this two/three year old system is already outdated again. As the local technicians have no system to test and there is no "processor simulator", all changes on the system are done by an external contractor. <R.21>

[3] http://www.crestron.com/downloads/pdf/product_misc/rg_2-series_control_systems.pdf
[4] https://www.youtube.com/watch?v=-mu3MmNkr_Y

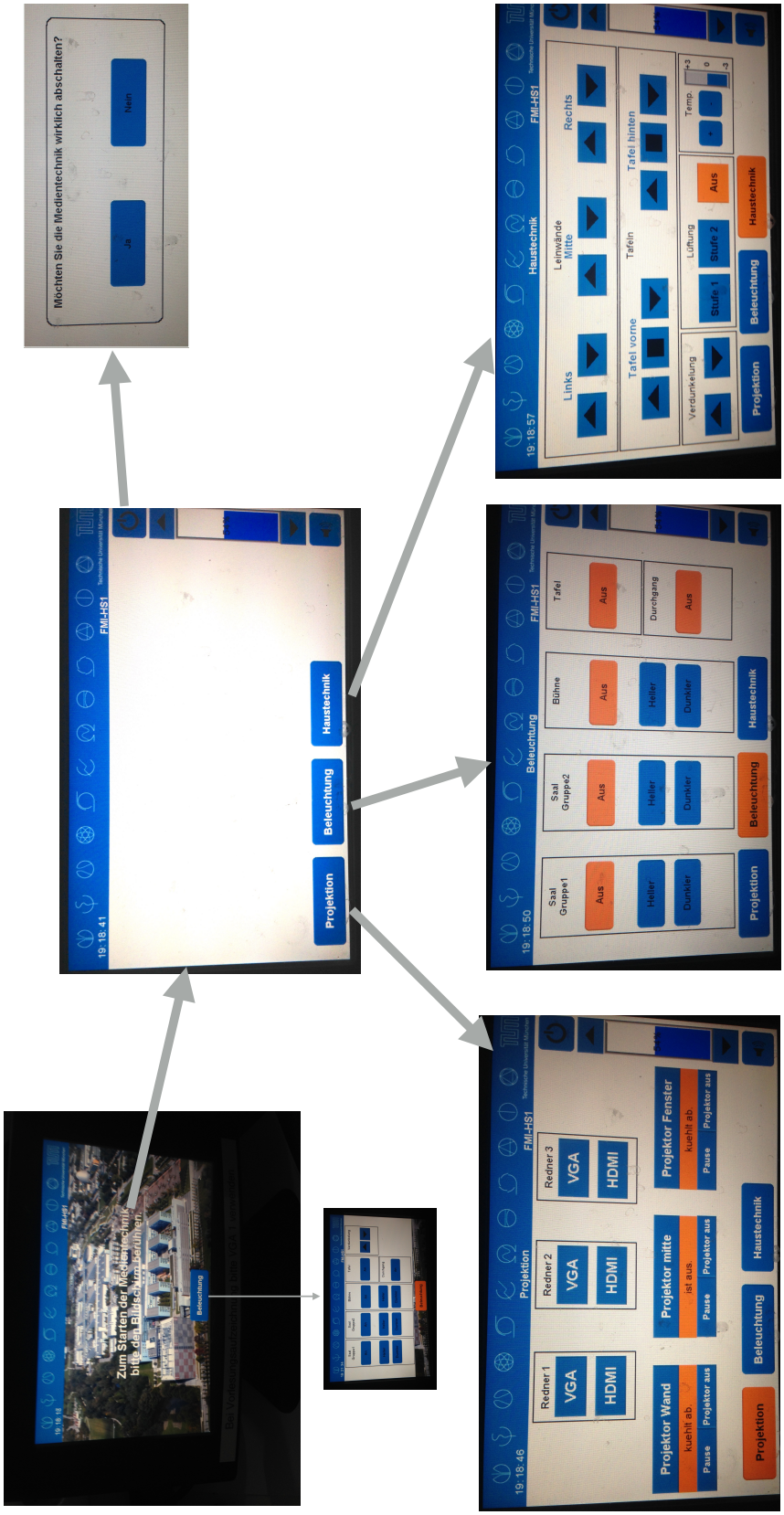


Figure 2.5: Usage flow of touch screen on main speaker's desk in big auditorium

2.2 VSL and DS2OS

In the first section of this chapter I presented different spaces, their associated users and their current and future usage scenarios. I now introduce the system, I am to design a graphical user interface for. This system is called Virtual State Layer (VSL) and is a set of concepts for Smart Space Orchestration published by Marc-Oliver Pahl in 2014 [1]. Together with it's foundation services, the VSL provides a middle-ware for modelling, accessing and processing information about a smart space and it's devices.

As described in Chapter 1 (Introduction) the VSL's goals are to connect previous incompatible devices through one unified namespace, reusable services distributed via a global exchange platform, and good usability for users and developers, with security in mind.

To explain the different concepts I introduce an excerpt of the TUM Autonomic Home Networking (AHN) lab setup, as shown in Figure 2.6. In this setup we use one *Knowledge Agent* (KA) with three VSL gateway *services*. Gateway 1 connects lamps via simple radio controlled outlets and a Raspberry PI based 433 MHz transceiver, gateway 2 connects the blinds of the room via an Arduino with Ethernet shield and relays, and gateway 3 a 8-way web controllable 19-inch power plug array.

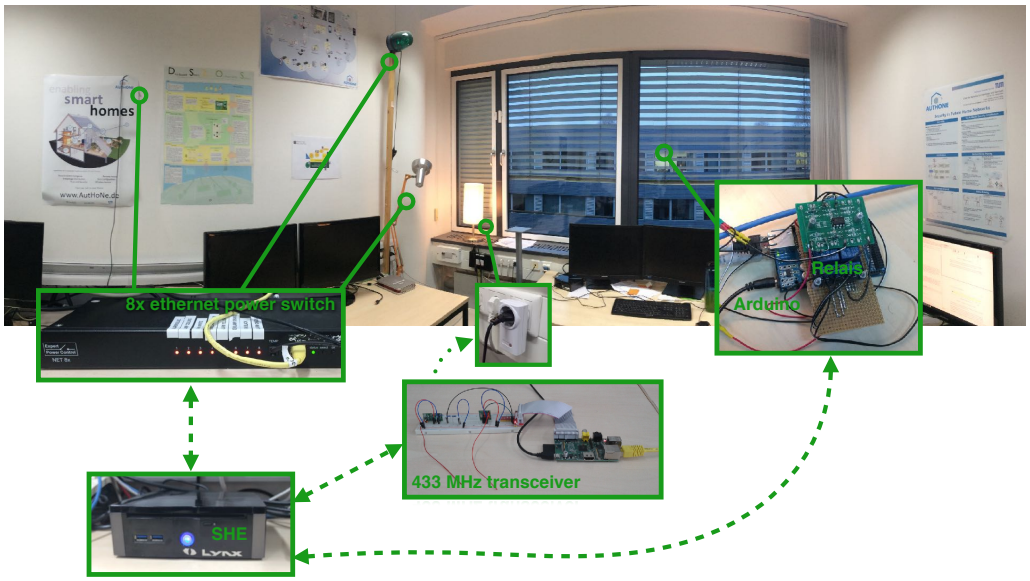


Figure 2.6: TUM Autonomic Home Networking lab in December 2016

- Lamp ← simple radio controlled outlet ← Raspberry PI with 433 MHz transceiver ↔ VSL service via HTTP ↔ KA
- Blinds ← Arduino with Ethernet shield and relays ↔ VSL service via Telnet ↔ KA
- 8-way web controllable 19" power plug array ↔ VSL service via HTTP ↔ KA

2.2.1 Data model

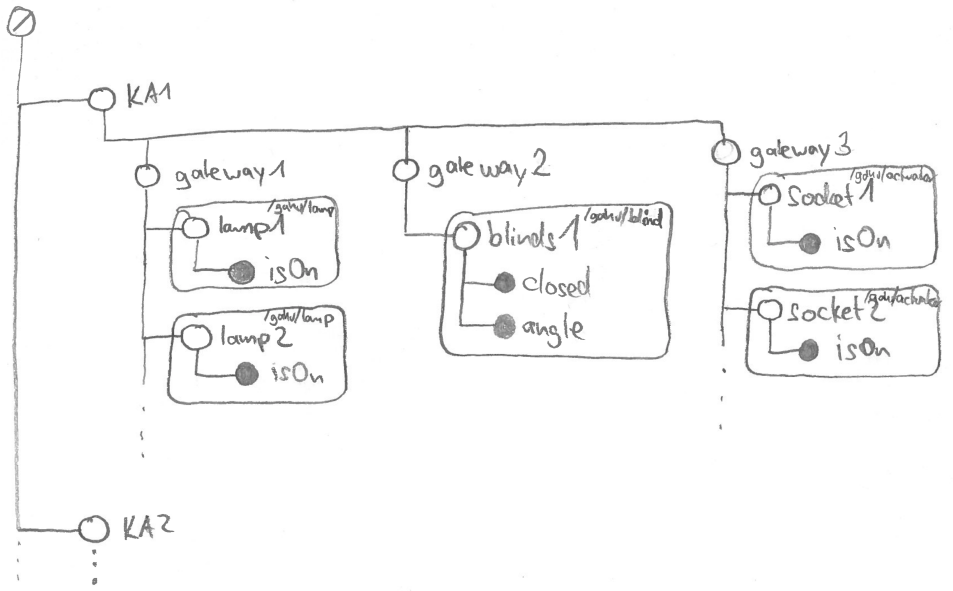


Figure 2.7: VSL context for AHN lab

The data is called VSL context and is a hierarchical namespace distributed between all KAs at a site [4]. Each *node* has one or multiple *type*s, a *value* – and optional – named *children*. The AHN lab context is shown in Figure 2.7 – one might recognize the three gateway nodes with there assigned devices (lamp, blinds, socket) as children. The attributes (isOn, closed, angle) of the devices are in turn children of the device nodes.

A node can be addressed via the names of its parents. For the example shown in Figure 2.7 the path of the isOn attribute of lamp1 is */KA1/gateway1/lamp1/isOn*.

The different types of nodes are provided via the *Model Repository* (CMR, see 2.2.2). All types are derived from three base types: *text*, *number* and *list* and a fourth *composed* type. The type system supports multiple inheritance. The relationships between the types used at the AHN lab is shown in Figure 2.8. Types have a double role: classification of the value (boolean, number) and defining the semantic meaning (lamp, brightness). The format of the value might be further defined with *restrictions*, e.g. minimal and maximal values for numbers or regular expressions for text.

The VSL also provides an authorization system based on client certificates and reader/writer groups. This allows read-only nodes or hiding of whole sub-trees. As this mechanism works transparently, this thesis does not go into more detail on the authorization topic: I have to assume that the VSL context can change (e.g. when user authenticates) and that there are read-only nodes.

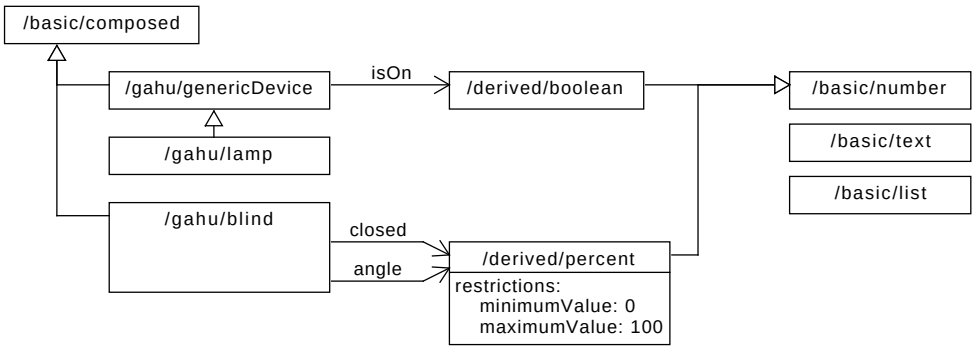


Figure 2.8: Relationships between types used in AHN lab

Services can add virtual nodes to the tree. The content of virtual nodes is dynamically created by the owning service.

The information which push button or switch controls which lamp is not part of this model, as it is stored inside an orchestration service.

2.2.2 Deployment and services inter-exchange

KAs typically run on every computer with enough resources taking part at a DS2OS site. Such a computer does not have to be a dedicated server or desktop PC: Single board computers like a Raspberry Pi or home network routers like an AVM Fritz!Box are also possible KA hosts.

As of December 2016, in the AHN lab only one KA runs permanently. If necessary, students can connect their notebook to the LAN and start additional KA instances on their own. KAs discover themselves automatically and allow their registered services to intercommunicate. For example, the speech recognition service on the students notebook can access the a gateway service controlling the lamps in the lab. The KA runs as a Java process in a GNU Screen session, together with the gateway services, as well as a Web UI server process – compare Figure 2.9. In the VSL concept, this session is the Service Hosting Environment (SHE), which would also be controllable via the VSL. The usage of GNU Screen is only a workaround till a real SHE implementation is available.

VSL services are small programs focussing on one problem, similar to apps on mobile platforms like Android or iOS. To be reusable in other sites, VSL services should be generic and modular.

A gateway service is an adapter to a concrete device like a lamp, a washing machine, a gas heating controller or a building automation bus like KNX. Besides gateways services, the

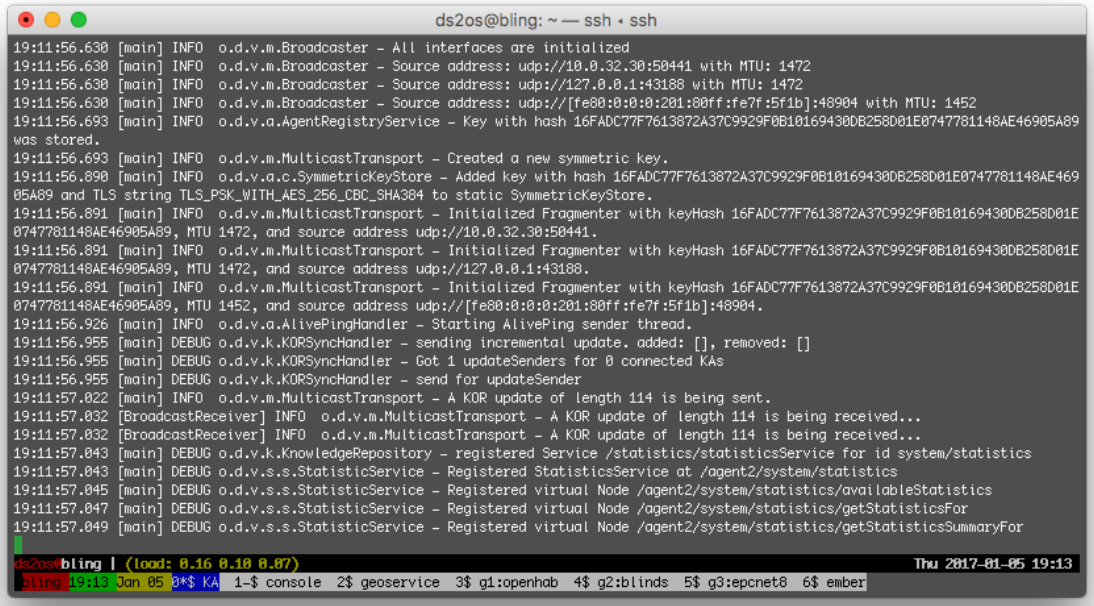


Figure 2.9: SHE in AHN lab: GNU Screen session with KA, KA console and other services

VSL includes orchestration, advanced reasoning, and user interface services. Advanced reasoning services draw conclusions (“Currently it is daytime in winter”) from values they can access via the VSL (e.g. brightness, temperature). Orchestration services connect sensors with actors. In the simplest case two devices which can not directly communicate – like a hardware switch with a lamp from different vendors. The more complex cases involve save and recall of device settings like a ‘film scene’ in a living room or user preferences regarding the light colour based on current daytime.

Users can share their newly developed and improved services between themselves via a global repository, the Smart Space Store (S2Store). This store includes the Model Repository (CMR, see 2.2.1), and other global repositories and services. Each space where the VSL is deployed is called *site*. All sites together with the central store form the *DS2OS* (Distributed smart space orchestration system).

2.2.3 VSL API

The VSL context can be accessed by a closed set of methods: get and set. To get a notification on value changes via WebSocket, one can subscribe (and unsubscribe) to a node.

As of January 2017, the KA has a HTTP REST API and uses JSON for its responses. Therefore a JavaScript client can communicate directly with the KA. The authentication is exclusively done via client certificates and acts transparently: Sub-trees to which the certificate has no access to are not returned by the API. Example responses for get requests are shown in Listings 2 and 3, an example set request in Listing 4.

To reproduce the requests listed in this section, the client might require an entry in /etc/hosts for KA1, so requests to `https://KA1:8080` do not fail. Each request needs to use a client certificate (e.g. service1 from `http://dev.ds2os.org/certificates/`) and a `Content-Type: application/json` header.

GET /KA1/gateway1/lamp1?depth=1&scope=complete

```
{
  "types": ["/gahu/lamp", "/gahu/genericDevice", "/basic/composed"],
  "children": {
    "isOn": {
      "types": ["/derived/boolean", "/basic/number"],
      "restrictions": {"minimumValue": "0", "maximumValue": "1"},
      "value": "0"
     }
}}
```

Listing 2: reduced JSON response for a lamp

GET /KA1/gateway2/blinds1?depth=1&scope=complete

```
{
  "types": ["/gahu/blind", "/basic/composed"],
  "children": {
    "closed": {
      "types": ["/derived/percent", "/basic/number"],
      "restrictions": {"minimumValue": "0", "maximumValue": "100"},
      "value": "40"
    },
    "angle": {
      "types": ["/derived/percent", "/basic/number"],
      "restrictions": {"minimumValue": "0", "maximumValue": "100"},
      "value": "50"
    }
}}
```

Listing 3: reduced JSON response for blinds

PUT /KA1/gateway1/lamp1/isOn

```
{
  "value": "1"
}
```

Response: *HTTP/1.1 204 No Content*

Listing 4: JSON request to switch lamp1 on[5]

[5] In the current AHN lab setup, one would have to use a desired child node to change the value. This node is part of MAPE concept, which is not explained in this document. See [1, page 320f] for more information.

To get notifications the client has connect to `wss://KA1:8080/callbacks` with WebSocket protocol "v1.vsl.ds2os.org". The client has to generate a random uuid and subscribe via an HTTP POST request as shown in Listing 5.

POST /KA1/gateway1?depth=-1

```
{
  "operation": "SUBSCRIBE",
  "callbackId": "cf255f45-c442-4af8-95f7-1c054ad0093a"
}
```

Response: *HTTP/1.1 204 No Content*

Listing 5: Example JSON request to register for notifications of whole sub-tree of gateway1, connected to KA1

The last WebSocket opened with the same client certificate, now receives messages with this callbackId, the changed node's address and a serial number:

```
{
  "callbackId": "cf255f45-c442-4af8-95f7-1c054ad0093a",
  "serial": 33,
  "address": "/KA1/gateway1/lamp1/isOn",
  "invoke": "NOTIFY"
}
```

The successful receipt of this message must be confirmed by the client via a return message including the callbackId and serial:

```
{
  "callbackId": "cf255f45-c442-4af8-95f7-1c054ad0093a",
  "serial": 33
}
```

When the client terminates, it should destroy the subscription as shown in Listing 6.

POST /KA1/gateway1?depth=-1

```
{
  "operation": "UNSUBSCRIBE",
  "callbackId": "cf255f45-c442-4af8-95f7-1c054ad0093a"
}
```

Response: *HTTP/1.1 204 No Content*

Listing 6: Example JSON request to unsubscribe

As of January 2017, each client requires an own certificate – even a second browser tab of the same web app can steal the other tab's notifications.

2.2.4 Programmatic UI, extensible by third party

Every item in the VSL data model can be traced back to a base type, cf. Figure 2.8. When rules how to create the UI for these base types are defined, programmatic creation of a basic user interface is possible. Let's revisit the AHN lab as described at the start of this section (2.2) and Figure 2.7. `lamp1` is of type */gahu/lamp* which is traced back to */basic/composed/*. The rule for composed is to iterate over all children. `lamp1` has only one child node: `isOn`, which is of type */derived/boolean*. The rule for boolean defines a switch as UI element. The other case in the AHN lab are the blinds: `blinds1` is of type */gahu/blind*, which also traces back to */basic/composed*, so the UI again iterates over all children. This time these are `closed` and `angle`, both of type */derived/percent*. As the UI does not know this type, it falls back to the parent type */basic/number*. The rule for number has a case distinction: As the restrictions *minimumValue* and *maximumValue* exist, the corresponding UI element is a slider.

2.2.4.1 Scenario: Extensible UI

The thesis's task description requires the UI to be extensible, so that third party developers can easily inject new components to the interface renderer – in other words: Add new rules to the UI. Let me illustrate that extensibility requirement with following scenario:

The user has radio controllable LED bulbs in his living room. The software control element is build with three sliders, adjusting the red, green and blue value of the light (compare left part of Figure 2.10). When he wants to dim the light, all three sliders have to be adjusted, and he always needs some time to find the right colour. As he is dissatisfied with this situation, he opens the "Extension Store". There he finds an extension that has only one slider for the colour, a second slider for the intensity and a toggle switch to turn the light off (compare right part of Figure 2.10). He hits the install button and the initial control element is replaced with the new one from the extension. Now he can switch off or dim the lights without destroying the colour settings.

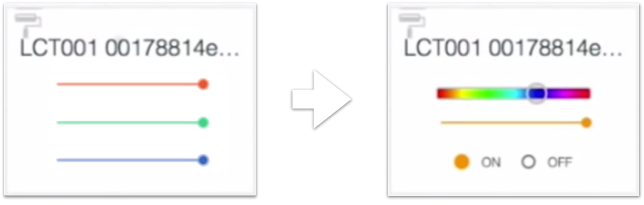


Figure 2.10: Simple control for colour lamp; left: simple from developer perspective, right: simple from user perspective

2.2.4.2 Scenario: Self-adaptive UI

Preconditions: The overall system (KA including UI) is never interrupted or reloaded.

The user has bought a washing machine with Wi-Fi, connected it already to water and electricity, and entered the Wi-Fi credentials. The system recognizes the new machine via auto-discovery methods (e.g. via Bonjour/mDNS) and adds it to an inbox view. The User visits the inbox view and installs the corresponding gateway service, which includes a set of washing machine specific UI extensions. The icon in the inbox transforms from a generic to a more specific one, titled with the machine's manufacturer and type. The user moves this icon from the inbox to the floor plan. Other instances of the UI get updated automatically: They now display the new machine in the floor plan, and use the accompanying UI extensions when users access the washing machine.

2.3 Use case model

In this section I create a use case model through combining the scenarios gathered during the interviews (2.1), the VSL concept (2.2) and other related work (3). A use case model consists of the actual use cases and their actors. Actors can be user roles, internal processes or other external systems [5, sec. 4.4]. The resulting top level use cases with their actors are shown in Figure 2.11.

The use cases are split into groups: daily operation of devices (switch light on <U.1>); meta data management (device locations <U.2>, floor plan <U.5>, users and their rights <U.6>); service/extension management <U.3> <U.4> and development <U.7> <U.8>; and system debugging.

<U.1> operate/control devices
<U.2> manage device location: add, update, delete device positions
<U.3> configure service
<U.4> install service
<U.5> manage floor plan
<U.6> manage user, permissions
<U.7> publish service
<U.8> implement service

Occupant is the most common user role of the system – basically everybody who works or lives in the smart space holds that role. Occupants are able to control the devices in the rooms they have access to <U.1>. They can switch the lights on (operate), change the heating schedule (control), modify the orchestration rules (control), or create/change a scene (control). When they move a standard lamp in their room, they also should be able to correct the device location inside of the system <U.2>. Inside their room they are able to change the mapping of sensors to actuators, e.g. which devices the push button besides their room door should affect. Occupants should be able to quickly learn the basic concepts of the system and are not expected to have programming knowledge.

The **Admin** role is responsible for the more complex or critical tasks: Modelling the building inside the system by importing a floor plan <U.5>. Assigning occupants the corresponding rights <U.6>. Giving them access to their devices <U.6>. Installing and configuring new services which concern multiple occupants, e.g. bringing the central heating into the smart space orchestration system <U.3> <U.4>.

Developers introduce new ideas into the system or improve existing ones by implementing reusable services in a programming language <U.8>. They publish these implementations at a global Store <U.7>. Other users can find them at the Store, install a service to their local smart space and give the developer feedback. This feedback and other statistics help other users to find interesting services and developers to improve (their) services.

External systems: The **Store** is a global off-site system, providing services to users and feedback to developers. **Devices** are other systems inside the smart space (actuators: lights; sensors: switches/buttons; and third-party gateways)

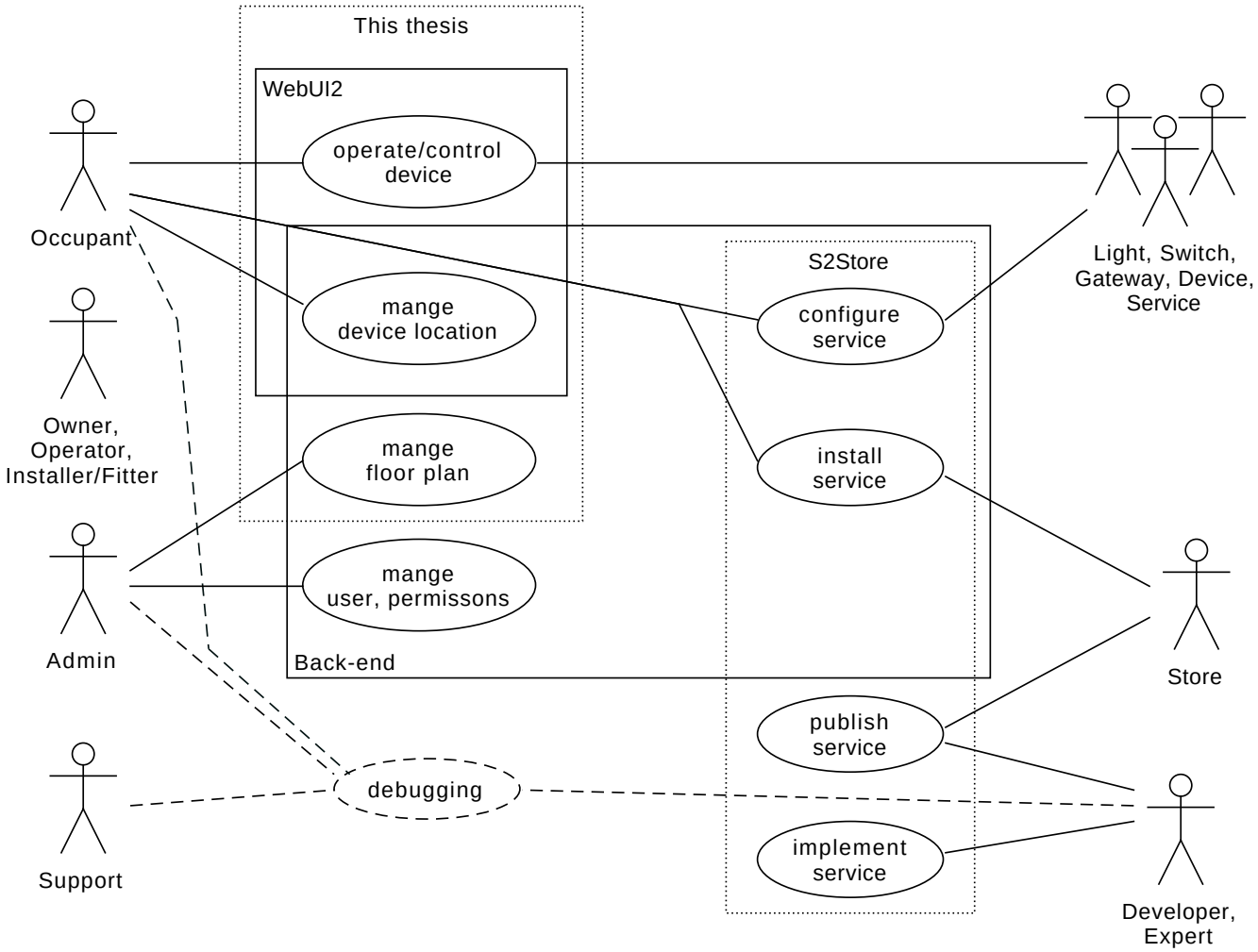


Figure 2.11: top level use cases with their actors

The different areas in Figure 2.11 are defined in sections 4.1, 4.2, and 4.3.4.

2.4 Back-end Analysis

The previous sections of this chapter presented visions for smart spaces via scenarios (2.1 and 2.2) which are used to build a use case model (2.3). This section deduces requirements for the necessary back-end adjustments. The actual user interface (Web UI 2) is treat in section 2.5.

In this section, I analyse the previous approach to handle geolocations in the VSL, and draw first conclusions for the future geo model (2.4.1).

The back-end should be able to answer questions like "*Which lamps are in the living room?*". To achieve this goal, the back-end needs to understand what this 'living room' thing is and which devices are a lamp. Literature often calls this capability 'semantics'. To solve this problem, it makes sense to rely on existing components such as geodatabases: The spatial data addons for PostgreSQL and SQLite are introduced (2.4.2).

The components in the back-end and front-end need to exchange geo-data with each other. As they both support a variety of geometry serializations, a common denominator has to be found (2.4.3).

2.4.1 Geo data and the VSL

The DS2OS (2.2) has no real specification for geo data yet. The first UI prototype added a *geolocation* node to the device models in form of an single 3D coordinate with latitude, longitude and altitude (see Listing 7). Since this approach is too inflexible, the concept has been changed: Instead of *get /KA1/oven1/geolocation* the location is now queried via an own sub-tree implemented via a virtual node by a service, e.g. with *get /KA1/geoservice/locationOf/oven1* or short *get /search/locationOf/oven1*.

models/demo/oven.xml

```
<model>
  <temperature type="/basic/number" />
  <threshold type="/basic/number" />
  <isOn type="/derived/boolean" />
  <isAlarm type="/derived/boolean" />
  <devicename type="/basic/text" />
  <geolocation type="/demo/geolocation" />
</model>
```

models/demo/geolocation.xml

```
<model>
  <lat type="/derived/decimal" />
  <lon type="/derived/decimal" />
  <alt type="/derived/decimal" />
</model>
```

Listing 7: VSL geolocations in 2014, in VSL types XML notation

Due to the DS2OS project's goals, the building model should work for three room flats, small houses, as well as for big office buildings, or conference centres. To illustrate the differences, Figures 2.12 and 2.13 shows side views of an exemplary home and the university building from section 2.1.5.

In general, challenges for indoor building models are:

- rooms which go over multiple floors e.g. auditoriums, stairs or elevators.
- different floor heights in some sections of a building
- intermediate storeys
- sub-dividable rooms

The model must be flexible enough to cover these special cases, without becoming too complex for the default case.

As designer of a building model, you have to decide how fine-grained you want to map the world. Does everything have to be to the nearest millimetre? Does every wall surface have to be a separate object in your model?

At OpenStreetMap – a world wide open-data geo wiki database where I'm involved during spare time – we agreed to allow several modelling variants at once [7], depending on which level of detail one needs:

For the lowest level of detail, a point of interest (POI) such as a clothes shop is as extended by the 3rd dimension, the *level*. The value of this attribute is the European floor numbering: 0 is the ground floor, 1 the floor over ground floor (British English: first floor, American English: second floor), 2 etc. and -1 the first basement floor.

However, if the human data editor wants to represent more details, the model provides additional principles: Each room is mapped as an area, compare Figure 2.14. Doors are simply defined as a point, if necessary, with optional width. The area types are divided into three categories: *Area* is a special case of a *room*, with the difference that an area does not has walls. To highlight their importance, corridors where given an own category. It's also specified how to model single walls, mainly for graphical reasons.

For a DS2OS geo-data model, the challenge is to find the right balance: From the implementation point of view it makes sense to only allow a single type of coordinate. The default case should be covered in a simple way, but the complex cases should still be reasonably possible. It should be possible to annotate each node within the VSL tree with a geo-location. However, a policy defining what this location represents exactly for each node type is required – e.g. for an 8-channel heating actuator: the installation location of the actuator, the control valve or the corresponding floor heating area. For the occupants probably only the latter is interesting, but a maintenance worker is also interested in the others.

The other question is how to model relations between individual objects. The naive approach is to explicitly state individual relations and thus establishing a hierarchy: Device A is in room B, room B is part of floor C, floor C is part of building D. This approach would not address the challenges listed at the beginning of this subsection: Think about a mobile device like a projector or overlapping/sub-dividable rooms. I prefer a flexible approach based on set theory: Every entity is represented by a three-dimensional (3D) geometry: For devices a point in 3D space might be enough. For rooms, floors, buildings, etc. one would use cubes or polygons. If we now want to turn off all lights on one floor, the system only has to intersect geometries with each other and gets a list of the appropriate devices.

The chosen structure for this work is presented in subsection 4.2.1.

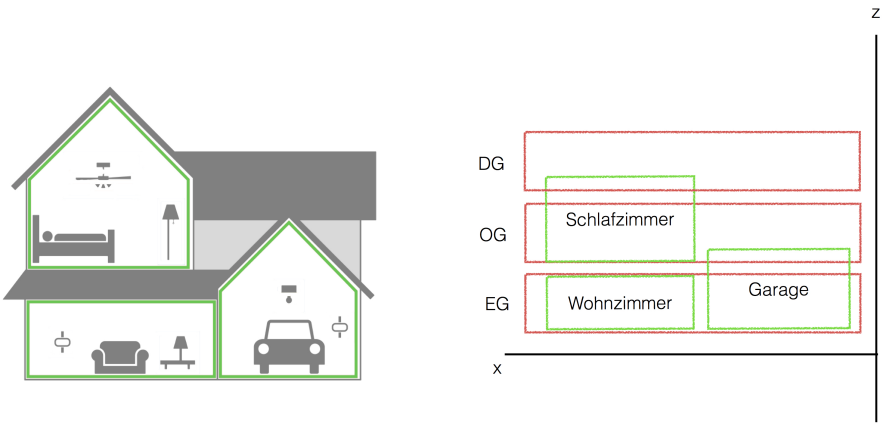


Figure 2.12: Sketch and side view of an example home. Sketch derived from [6].

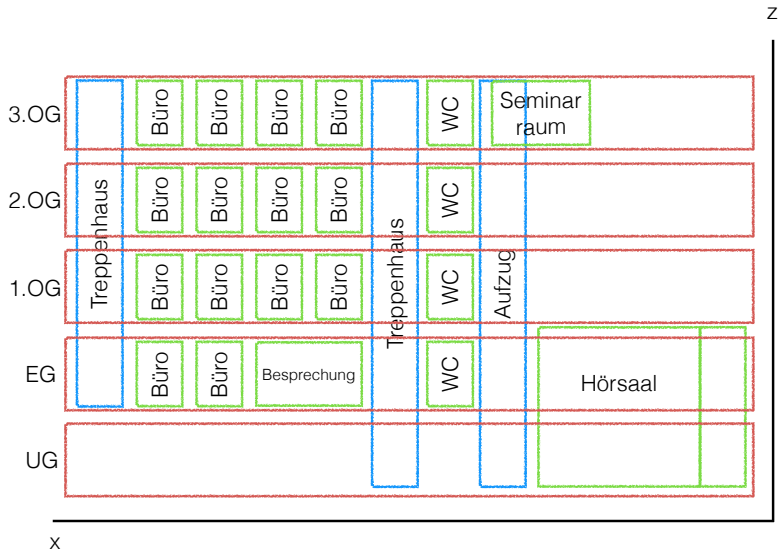


Figure 2.13: Side view of the university building

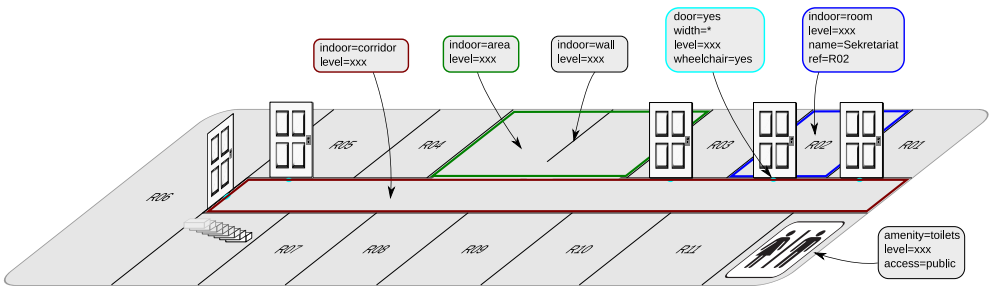


Figure 2.14: Indoor elements from [7]

2.4.2 Geo databases

As the DS2OS should work for small houses as well as for complex buildings a flexible way of managing geospatial data has advantages. As introduced in the previous section the position of devices should be specified as point in the 3D space. Rooms and other extends in the spatial world, are defined by an more complex geometry – in most cases a simple cube.

To find all devices in a room the system has to implement set theory, e.g. intersection of a point with an other geometry. Set theory can either be implemented in an additional programming layer, or by the database itself. Databases which have special data types and methods to calculate spatial relationships are called geodatabases. In the open source database world, there are two serious implementations: An extension called PostGIS for the heavy-duty database PostgreSQL, and an extension called SpatiaLite for the lightweight database SQLite. Of course commercial databases have similar components, e.g. Oracle Spatial and Graph. For further information on the subject please refer to [8].

Currently the VSL KA (2.2) uses the Java database HSQL. In the future content from this KA DB could also be moved to the same database as the geo data.

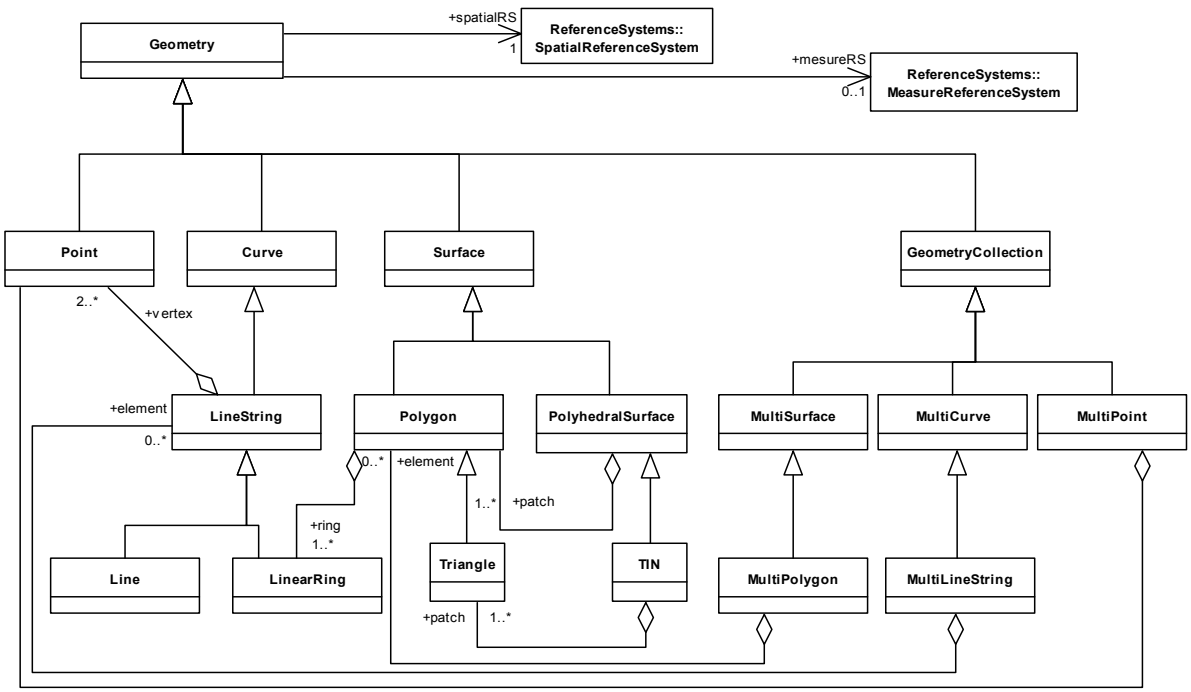


Figure 2.15: OpenGIS SQL Geometry Type hierarchy from [9]

2.4.3 Geometries

The components to be used in the back-end (geodatabase, 2.4.2) and front-end (web map library, 2.5.6.3) support a variety of geometry serializations. Since both exchange data with each other, a common denominator has to be found. Therefore this section presents different geometry types and their serialisation formats. It closes with Table 2.3 comparing the relevant capabilities of these components with each other.

For this work I need polytopes in dimensions 0, 2, and 3 – points, polygons and polyhedrons. All in 3D space. A point consists of a single coordinate, so three float numbers. Polygons define an area (in one plane). Three-dimensional areas like a building require a cube or generalized a polyhedron. To describe geometries in computers, there exist different models – relevant for this work are WKT and GeoJSON:

Well Known Text (WKT) is a markup language that extends the SQL standard to add geometries. It is the "human-readable version" of its binary cousin (Well Known Binary, WKB) in which the geometries are stored in the database's memory. The underlying data models and hierarchy are shown in Figure 2.15.

GeoJSON is an open standard format designed for representing simple geographical features, along with their non-spatial attributes, based on JavaScript Object Notation (JSON). Its current specification is RFC 7946 [10].

The various WKT specifications and implementations differ in their feature set: As of 2016, PostGIS's WKT implements all geometry types shown in Figure 2.15, SpatiaLite only polytopes up to dimension 2 – especially not PolyhedralSurface, c.f. Table 2.2.

	PostGIS WKT	SpatiaLite WKT	GeoJSON
Point	yes	yes	yes
Polygon	yes	yes	yes
PolyhedralSurface	yes	no	no
Feature	no	no	yes

Table 2.2: Comparison of geometry data formats

The specifications mentioned above also define geometry functions. However, in the case of PostGIS and SpatiaLite, the function names differ slightly. Some of these functions are only defined in 2D.

With GeoJSON the situation is similar to SpatiaLite's WKT implementation: The 'simple geographical features' do not include polyhedra. But points and polygons with 3D coordinates as well as 3D bounding boxes are supported. GeoJSON, as specified by [10, Ch. 4], only allows WGS84 long lat as spatial reference system – whereby earlier versions of the specification allowed other coordinate systems.

Even the most recent OpenGIS specification [9] does not provide a own data type for closed objects also known as *solids*: After saving to the database, the original geom-

etry of a simple cuboid is not easily recognizable as such. For example with PostGIS 2.3: `BOX3D(11 48 2, 12 49 3)` becomes `POLYHEDRALSURFACE Z (((11 48 2,11 49 2,12 49 2,12 48 2,11 48 2)), ((11 48 3,11 49 3,12 49 3,12 48 3,11 48 3)), ((11 48 2,11 49 2,11 49 3,11 48 3,11 48 2)), ((12 48 2,12 49 2,12 49 3,12 48 3,12 48 2)), ((11 48 2,12 48 2,12 48 3,11 48 3,11 48 2)), ((11 49 2,12 49 2,12 49 3,11 49 3,11 49 2))`. For solid 3D functions like `ST_Volume()` one has to cast closed WKT geometries to a solid with `ST_MakeSolid()`, as shown in Listing 8.

```
SELECT ST_Volume(ST_MakeSolid(geom))
  FROM (SELECT 'POLYHEDRALSURFACE( ((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)),
    ((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)),
    ((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
    ((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)),
    ((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)),
    ((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1)) )'::geometry) as f(geom);
```

Listing 8: PostGIS SQL query to calculate the volume of a closed polyhedron [adapted from example at `http://postgis.net/docs/ST_Volume.html`]

3D data format	web map libraries			geodatabases	
	OpenLayers 3	Leaflet	Caesium	PostGIS	SpatialLite
WKT	yes	via plugins		yes	yes
GeoJSON	yes	yes	yes	yes	yes
glTF			yes		
X3D				yes	
GML	yes	via plugins		yes	yes
KML	yes	via plugins	yes	yes	yes

Table 2.3: 3D data formats and their support by front-end (2.5.6.3) and back-end software

Further ‘full-3D’ data formats are X3D, glTF, GML and KML. Except for the comparison in Table 2.3, they are not dealt with in more detail in this work. The chosen geometry types for this work are presented in subsection 4.2.1.

2.5 Front-end Analysis

The first two sections of this chapter presented visions for smart spaces via scenarios (2.1 and 2.2) which are used to build a use case model (2.3). This section finally deals with the actual user interface. I discuss the relevant challenges and which bricks I require to reach the goals of this thesis.

The first subsection (2.5.1) presents a model for user interaction in smart spaces and further defines the area of this work. The “Usability Engineering Lifecycle” – a development process to ensure optimal usability – is introduced and correlated to this thesis’s structure.

As this work’s project definition dictates that floor plans are a large part of the app, I have to investigate (2.5.2) what kind of floor plan I need. I review existing solutions:

Classic architecture floor plans, web map services presenting indoor data, and floor plan views actually designed for smart spaces. As applicable, I iterate over these test subjects three times: First analysing the graphical floor plan representation itself, then indoor map UI elements and in the last run the smart space UI elements. The subsection concludes with a summary for each subject and derivation of concrete requirements.

After introducing the origin of apps, tablets, and their operation systems, subsection 2.5.3 explains the different ways to create apps, and how to distribute these for multiple platforms at the same time.

Subsection 2.5.4 investigates, if there are any rules one can follow to build a web app with good UI and usability. It introduces human interface guideline documents, that platform vendors provide to third party developers. I conclude that the DS2OS project is also creating some kind of platform and therefore might benefit from own guideline documents.

Subsection 2.5.5 discusses the different ways of dynamically optimizing UI for spaces and its users. I split the approaches into programmatic UI, customization by end user, optimization through machine learning and (semi) automatic generation of new extensions.

The final subsection (2.5.6) introduces the brick types I need for construction of my web UI prototype. As web apps are typically written in JavaScript dialects, I give a short introduction into the environment and mindset of this scripting language and its developers. I talk about libraries and frameworks providing toolboxes and additional guidelines for web app development. I come to the conclusion that all three selected web application frameworks of the current generation are equivalent. The decision which one to use depends on the associated ecosystem. Therefore I provide a short overview over the associated libraries especially regarding UI elements and floor plans.

2.5.1 Development of user interfaces for smart spaces

This section characterises the term user interface (UI) and the UI development process in the context of the smart spaces. These foundations are extended in section 2.5.4 and assumed as given in the further course of this document.

If one takes the term “user interface” literally it’s the cut surface between human and machine, so input and output devices. In other words, the things allowing machines to talk to us humans, and vice versa. A user communicates with another system by performing action on an input device (for example a microphone) . This system processes the action and gives the user a reaction via an output device (for example a speaker). The user perceive this reaction, processes it, and possibly performs other actions. This creates a kind of closed control loop. In the case of a smartphone or tablet, input and output coincide in the same physical device.

Now let me introduce a model for user interaction in smart spaces from [11] [12]: The authors of this VDE guideline split the overall system into three actors, see Figure 2.16: user, smart home and service.

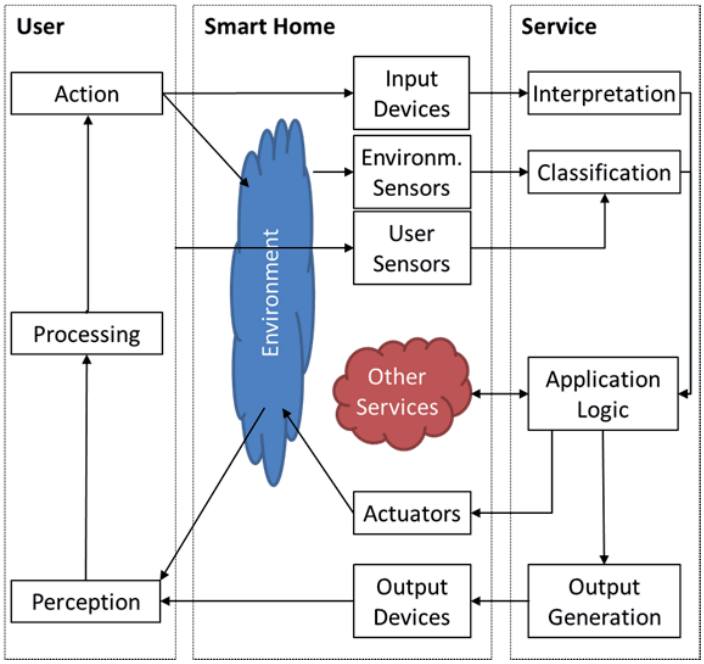


Figure 2.16: Simplified and generalized representation of user interaction with services in the Smart Home from [12]

In a smart space the user no longer only affects the input devices, but also the environment as services can perceive it through environmental sensors (temperature, brightness). There are user sensors which might provide pulse, location etc. The reaction from service to user no longer only happens thought devices, but through changes to the environment via actuators (heating or light switches on).

This thesis uses the term "user interface" only for the in-/output devices and not for any sensors[6] or actuators. In the words of [11]: "The interaction with a user interface would be described as explicit interaction, whereas data from sensors and actuators are classified as implicit interaction."

Having defined the term user interface for the context of this work, I now discuss the design and development process of user interfaces.

When searching for UI papers, I recommend to use "smart home usability" as search terms and not "smart home user interface". Besides usability engineering there is also the term user experience design (UX):

> "ISO 9241-11 defines **usability** as 'the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use'." [12]
>
> "In order to complement this traditional view on usability with non-functional aspects, the concept of **User Experience** (UX) has been brought up. ISO

[6] excluding input devices like wall switches which some vendors (3.2) also categorize as sensors.

> 9241-210 takes a comprehensive view and defines user experience as ‘a person’s perceptions and responses that result from the use or anticipated use of a product, system or service’. Although this view might be very broad, it shows that classical perspectives might be too limited in covering all aspects which are considered to be decisive for a user to successfully use a service offered in a smart home context. ” [12]

To summarize: The packaging in which the product is delivered plays a role in UX, but not in usability engineering. This work tries to optimize usability and does not deal with UX.

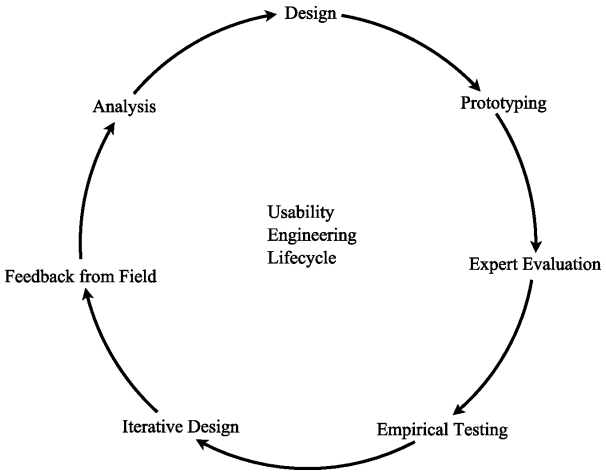


Figure 2.17: Usability engineering lifecycle from [11] based on Nielsen 1993

Section 1.5 of the VDE guideline “Messung und Bewertung der Usability in Smart Home-Umgebungen” [11] describes the user interface development process as follows:

> “To ensure optimal usability, precautions must be taken even before the system design. Thus, the characteristics and capabilities of users, requirements regarding the tasks performed with the UI and typical processes executing these, and the usage context have to be analysed. First designs for the interface should consider these findings. Based on the results of the task analysis usability metrics and target values for these metrics are determined. [...]
>
> After the completion of the task analysis, the system design follows. The system design is crucial, because here the foundations for a good man-machine interaction are placed. One or more design proposals are implemented in prototypes that can be evaluated with respect to the usability. For this there are two possibilities: On the one hand different methods of expert-based evaluation, on the other hand tests with users. Both methods are typically used in parallel or alternated by turn. They provide leads for usability problems and opportunities for improvement. In particular, before implementation of the designs as software, these practices are very inexpensive

> to use, since changes, required to resolve found usability problems, are not changes to an existing implementation.
>
> Further development of the prototype is usually done in several cycles of re-design and evaluation [see Figure 2.17], since changes can uncover or create new problems.
>
> Once the targets for usability metrics are achieved, the new user interface can be tested or used in the field. User feedback allows to further optimize and provide ideas for future generations of this and other services."

To some degree, this document is one run of this cycle – the labels of Figure 2.17 map to following chapters and sections:

- *Analysis* to chapter 2 and 3,
- *Design* to chapter 4,
- *Prototyping* to subsubsection 4.3.4.1 (Paper prototype) and chapter 5,
- *Empirical Testing* to section 6.2, and
- *Feedback from Field* to section 6.3.

A future thesis builds on my work (cf. section 7.2) and the cycle begins anew.

2.5.2 Floor plans as smart space UI

Due to the project's definition, the visually largest part of the front-end is an interactive floor plan. Therefore I look into different existing solutions. My investigation subjects are: Classic architecture floor plans; three web map services presenting indoor data: Google Maps, OpenLevelUp, and Munimap; and three floor plan views actually designed for smart spaces: Two different map views from the free@home system (3.2) and a mockup from Apple. The remainder of this section structured as follows:

The first part is about the graphical floor plan representation itself: Does the map present a suitable amount of details, and are these details zoom depended? I investigate how room types like corridors, toilets, and others are distinguishable, e.g. by colour. I further explore the underlying data model: Are rooms an own semantic entity? Are doors, windows and furniture depicted? Is the presentation's view angle a classic top view or were the creators more creative?
In the part 'indoor map UIE', I analyse if the subject is zoomable and has support for multi-storey buildings. I study the UI elements (UIE) used to change floors (dropdown, buttons, etc.) and their screen position (fixed to screen or relative to building). I further look at the points of interests (POI) marker behaviour: When are which icons shown and how do they interact?
The third part is similar to the previous one, but this time I set the focus on Smart Space UIE. I connect to the POI markers from the previous part and examine whether this marker reflects device state as well as which user interactions are specifically supported.

Each part has it's own comparison table, c.f. Tables 2.4, 2.5, and 2.6.

Finally, I conclude with a summary for each subject and derive concrete requirements.

2.5.2.1 Graphical floor plan presentation

Architecture floor plans, as shown in Figure 2.18, are originally drawn for construction of buildings. Therefore they reveal too many details for raw usage in Smart Space UI (−−). The representation is independent of the scale to which the plan is printed. Typically used are coloured lines (c.f. plotters) and not coloured surfaces. Rooms can be inferred indirectly via the surrounding walls and the descriptive text in the middle. Doors can be recognized by quarter circles describing the door opening direction. Windows are plotted but difficult to recognize. Furniture is shown in some areas.

To my current knowledge, Architecture file formats explicitly modelling rooms as semantic entity are not very common in Germany, yet. Although new buildings are increasingly planned in programs that are capable of exporting IFC data (Industry Foundation Classes, standardized by buildingSMART Interational), older buildings are not so often digitally remodelled during modernization processes. But at least in 2020, Building Information Modelling (BIM) and therefore IFC becomes mandatory for some public building projects [13] [14]. Till then, the pragmatic option to gather semantic building data (like rooms as own semantic entity) is redrawing and manual assignment of meta data like room numbers.

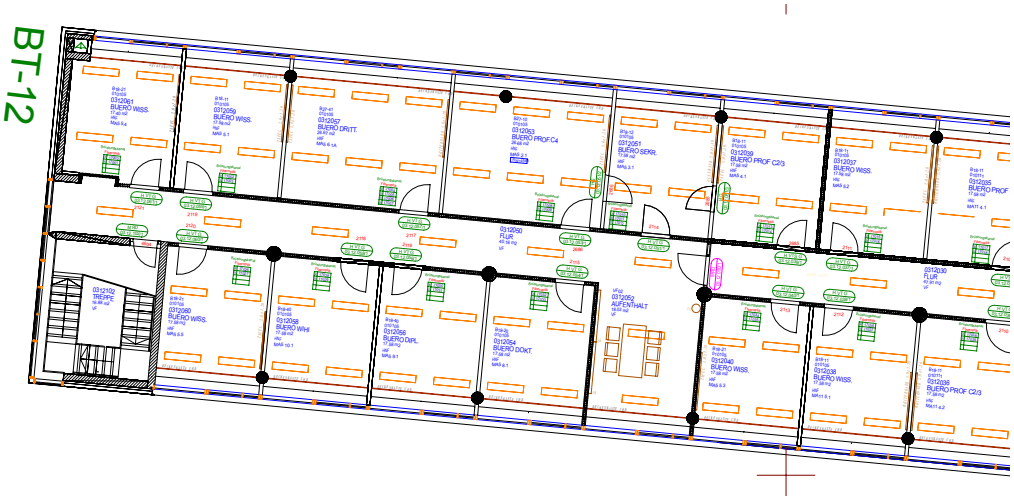


Figure 2.18: Architecture floor plan from `http://wwwrbgalt.in.tum.de/Garching/data/pdf/plaene/3OG-gesammt.pdf`

The **Google Maps** web view uses as suitable degree of details (+) to present indoor data. They use different colours for corridors, toilets, stairs and the other rooms. Rooms are (supposedly) explicitly modelled. At my test building no doors, windows, or furniture are visible, compare Figure 2.19.

OpenLevelUp has the suitable degree of details (+), too – compare Figure 2.20. The corridor is highlighted by a different colour, but the colour selection on Google Maps is more appealing. Rooms are modelled as the same. Doors are displayed on higher zoom levels as icons. In my test building, windows and furniture are not modelled.

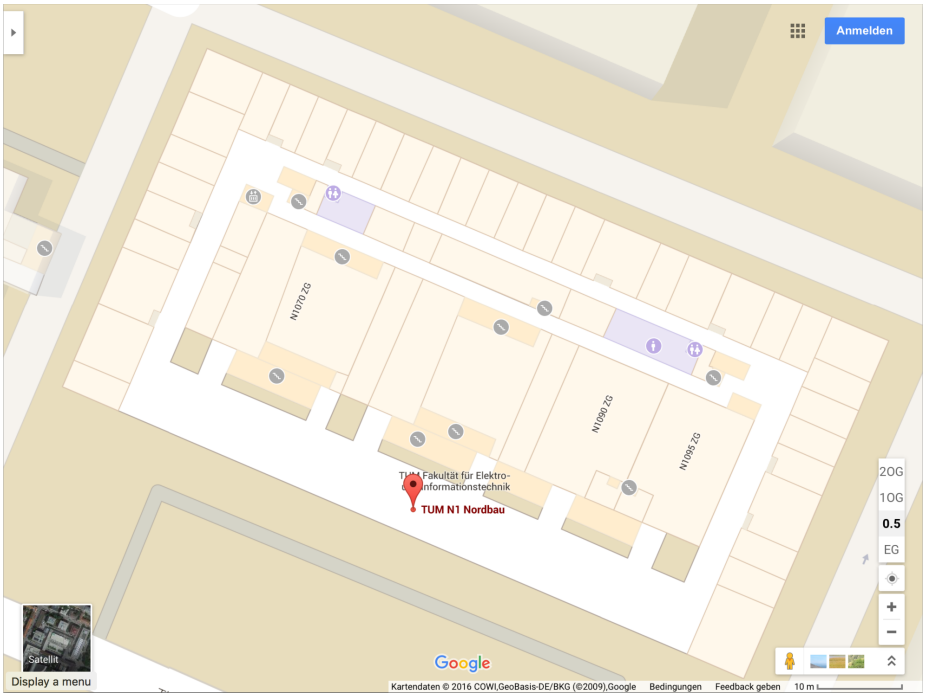


Figure 2.19: Screenshot from Goole Maps with indoor floor plan `https://goo.gl/maps/FgR4LexKAzp`

Figure 2.20: Screenshoot from OpenLevelUp indoor view using OpenStreetMap data [from `http://wiki.osm.org/File:OpenLevelUp_shopping_mall.png`] `http://openlevelup.net/?lat=48.136858&lon=-1.695054&z=18&lvl=0`

The "**Munimap**" project of Masaryk University is a solution in between: It is based on 2D geometries from architectural plans, broken down into doors, windows, rooms and floors [15]. The optical distinction between various room types is given, but in comparison to the continuous areas from Google Maps or OpenLevelUp they look cluttered. The used patterns are difficult to differentiate, compare Figure 2.21. Again, I have no access to the concrete data, but the statements in [15, 0:02:11] suggest that rooms are explicitly modelled as own entity – "Every indoor feature is polygon [...], georeferenced and related to **one** floor". They represent the "walkable surface". The doors at the test building (MU Rectorat) are partially marked by dark grey boxes. Windows can be recognized indirectly through indentations in the walls, the windows themselves are not plotted.

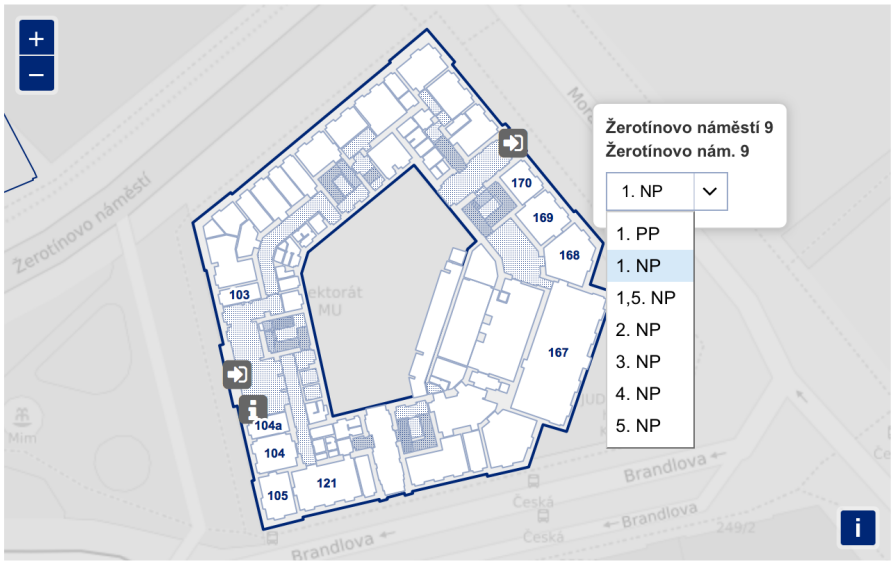


Figure 2.21: Screenshot from interactive mini map example at `http://maps.muni.cz/munimap/latest/example/helloworld.html` showing building related floor selector [15]

free@home has two different views for the building plan <R.12>: On the one hand the configuration mode (Figure 2.22), on the other the operation mode (Figures 2.23, 2.25). The developers have taken the right degree of detail (++). In the configuration floor plan, the detail view is not zoom dependent. It is not abstracted, i.e. the system always tries to represent the same amount of data. Although the symbols become smaller, I still recognize whether it is a lamp, switch or blind. Within the operation floor plan, the icons become smaller and I can no longer recognize what type a device is. All rooms have the same colour. The user must derive from the name of the room whether its a corridor or toilet.

When setting up the system, the owner draws a rectangle[7] per room, representing the walkable surface. The gaps between the rectangles implicitly form the walls. There is no possibility to model doors, windows and furniture.

[7] simplification, see section 3.2 for details

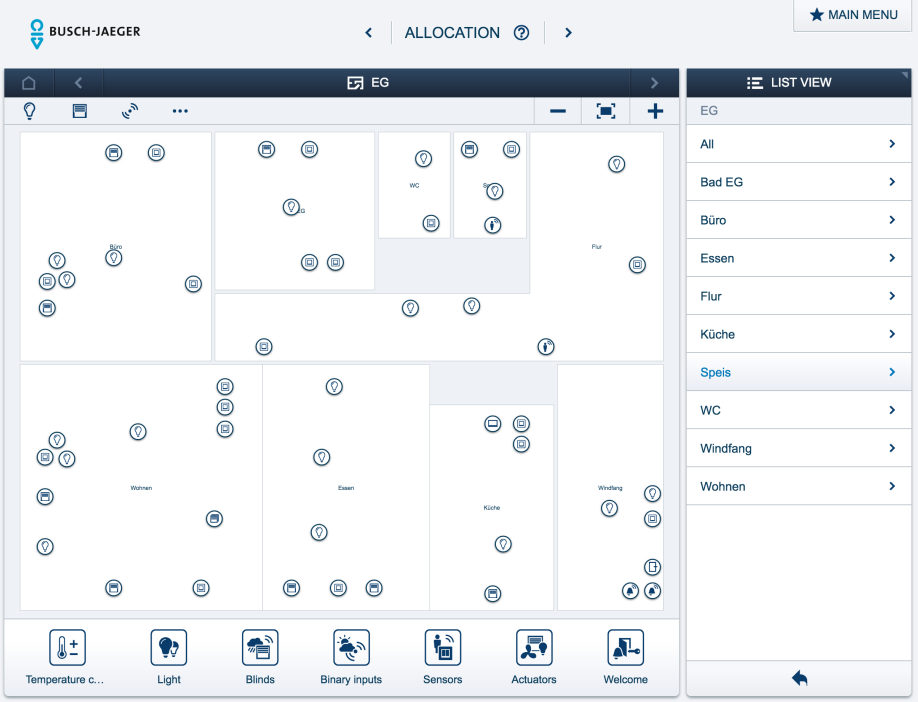


Figure 2.22: Screenshot free@home SysAP 2.0.4 configuration mode: allocation

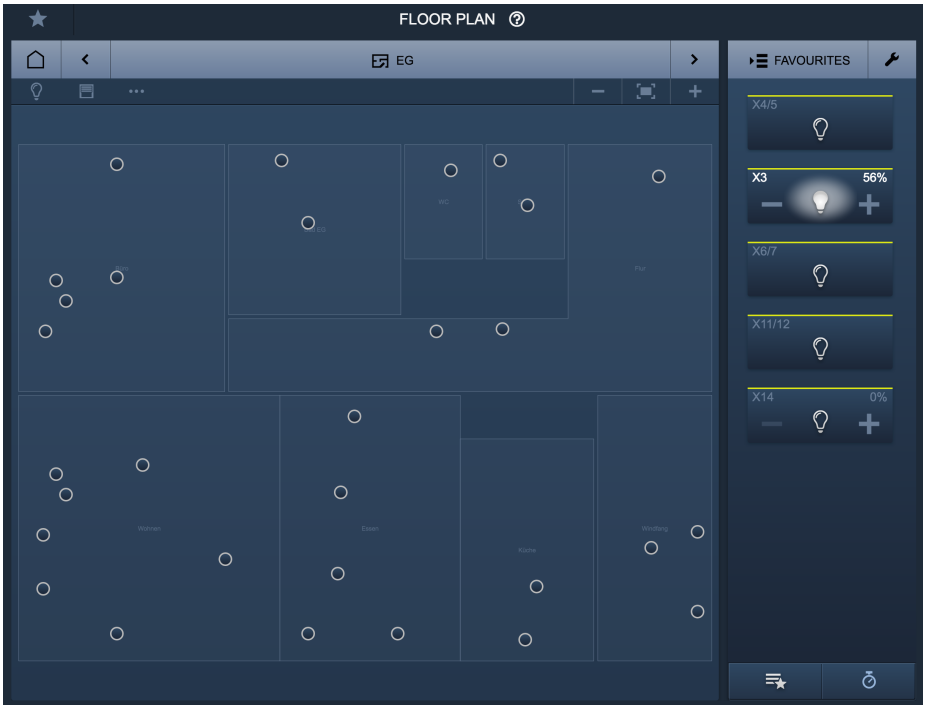


Figure 2.23: Screenshot free@home SysAP 2.0.4 operation mode: Floor plan in low zoom level

The last graphical representation (Figure 2.26) is a screenshot from **Apple**'s WWDC 2014 Keynote, during introducing Homekit: Although it is not a real product, this **mockup** provides interesting ideas: The level of detail is very good (++). Apple's designers did not use different colours for corridor, toilet, or similar – instead they placed furniture: Although the individual areas are not labelled with text, I can clearly recognize living room, dining table, kitchen and bedroom. In this comparison, it is the only representation which is not realised as top view, but in a perspective view – the plan is depicted as a 2D plane in 3D space.

Graphical floor plan representation	Architecture floor plan	Google Maps	OpenLevelUp	Munimap	free@home configuration	usage	Apple Mockup
suitable amount of details	no – –	yes +	yes +	yes 0	yes ++		yes ++
details zoom depended	no	yes	yes	yes	no	yes	no
different colours for corridors, toilets	no –	corridor ++ distinguish through colour	corridor ++ distinguish through colour	corridors 0 dotted	no –		no –
rooms are semantic entity	don't exist – semantically (if not IFC based)	yes +	yes +	yes, + walkable area	yes, + walkable area		possibly yes 0
doors	yes, with opening direction 0	no – –	yes, as icons 0	yes, as dark grey boxes +	no – –		yes, gaps in wall ++
windows	yes 0	no	no	wall indentions 0	no		grey gaps in wall ++
furniture	to some amount (yes) 0	no	no	no	no		yes, as grey areas ++
view angle	top view	top view	top view	top view	top view		perspective view

Table 2.4: Comparison of graphical presentation aspects of floor plans

2.5.2.2 Indoor Map UIE

After having discussed the graphic aspects of building plans, let us now look at the individual solutions in regard of indoor map user interface elements (UIE) – see Table 2.5 for summary.

The **architecture plan** has no UIE because paper is not interactive.

The **web maps** (Google Maps, OpenLevelUp and Munimap) are zoomable via gestures, mouse wheel, or buttons. They all support multi-storey buildings. To represent interesting places (POI short for 'points of interest') they use icons in fixed size relative to the screen resolution. Typically not all POIs are shown at once, they start to appear piece for piece depending on the current zoom level.

In the Google Maps web view (Figure 2.19), the floor selector is at the bottom right – near the zoom controls, aside are buttons for 'show current position', 'display images' and 'switch to Street View'. The floor selector, a vertical set of buttons for the individual floors, is fixed at the edge of the screen and always refers to the complete view. When the building plan is enlarged, the icons for staircases, toilets etc. have a higher priority and appear earlier.

At OpenLevelUp (Figure 2.20) the floor level selector is placed on the opposing side of the other controls. The developer used a drop-down box with up/down arrows. The

drop-down box lists all level values referenced in the current view area. At the example building, there are icons for each amenity (like shops, restaurants and banks); icons for toilets, stairs, elevators; or – at higher zoom levels – icons for doors. When zooming out, the POIs get combined into clusters <R.6>.

For Munimap (Figure 2.21), the floor selector is attached to the selected building and not to the edge of the screen – When the user moves the building/map the floor selector popup follows her movements. Despite this different concept, the selector itself is a simple drop-down field. In the low zoom levels there are only icons for building entrance and an info point. The next zoom level adds stairs and elevators, the next one toilets.

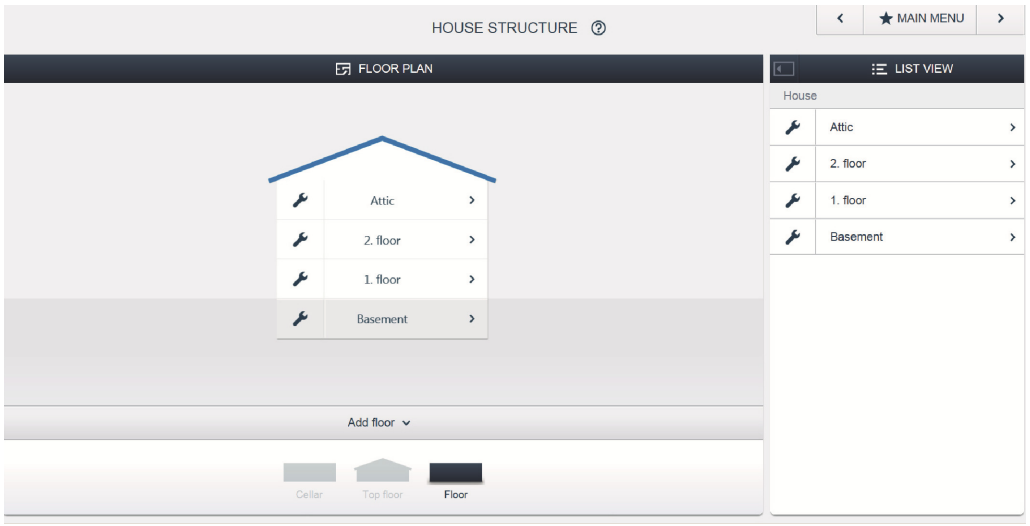


Figure 2.24: free@home SysAP 1.3 configuration mode: floor selection via side view from [16]

The building plans in **free@home** are zoomable and designed for a single multi-storey building. The floor selector is implemented as a side view. As shown in Figure 2.24: There is one cellar, multiple intermediate stories like ground floor or upper floor, and one top floor nicely symbolized with a roof.

By clicking on the arrow behind the floor name, I get to the actual floor view, see Figures 2.22 and 2.23. This view has a second floor selector: A dedicated bar at the top of the map, consisting of a house symbol, an arrow pointing to the left, the floor name, and an arrow pointing to the right. I can use the arrows to go one floor up or down, with a click on the house or the floor name I can return to the overall side view.

Since this second floor selector spans over the entire map width (c.f. Figure 2.25), the important parts are not close to the zoom control. The floor selector is fixed to the screen and not to the building. For free@home this is actually not an issue: There is only a single building, and not multiple ones with different floor numbering systems, like web maps have to cope with. There is no built-in icon clustering - multiple devices in the same XY location have to be misplaced: The location at free@home is not always the exact installation location of the device, but an approximate hint. This also applies

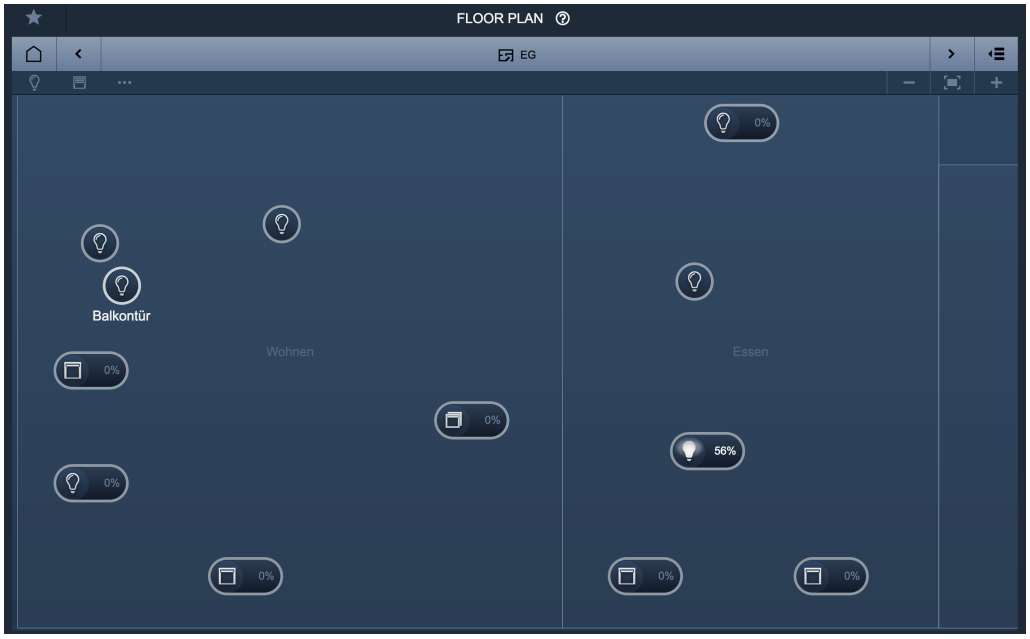


Figure 2.25: Screenshot free@home SysAP 2.0.4 operation mode: Floor plan in high zoom level

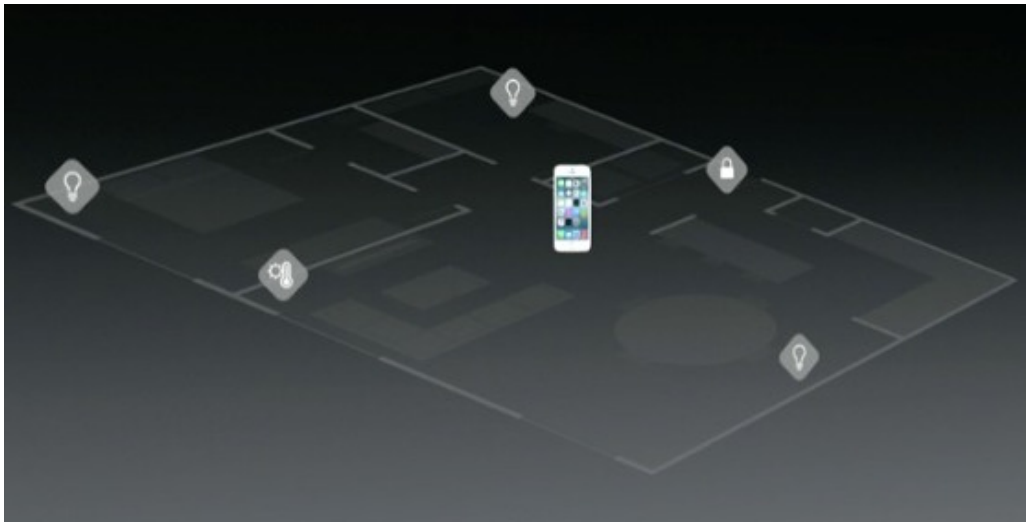


Figure 2.26: Screenshot from Apple WWDC Keynote 2014, announcing HomeKit from [17]

to the example configuration floor plan in Figure 2.22: The three switches in the room “Wohnen” near the door to “Flur” (corridor) mounted on the wall below each other. A similar problem exists with the outdoor lamps at the left edge of the building plan.

In contrast to all other solutions examined, at the free@home configuration view the icon size is dependent on the zoom level: When I enlarge the plan, the icons become larger. However, in the operation view the devices are only a small point, cf. Figure 2.23. When a section is enlarged, the view switches to a individual UI control element e.g. to turn on a lamp, as shown in Figure 2.25.

Indoor Map UIE	Architecture floor plan	Google Maps	OpenLevelUp	Munimap	free@home configuration	usage	Apple Mockup
zoomable	n/a	yes	yes	yes	yes		n/a
multiple floors	n/a	yes	yes	yes	yes		rather not
floor selector	n/a	listing on bottom right corner	top left, dropdown with arrows	floating, per building	own view + shortcut		
—near zoom controls	n/a	yes +	no –	no –	50% 0		
—per building	n/a	no 0	no 0	yes 0	no 0		
icon size zoom dependent	n/a	no, fixed	no, fixed	no, fixed	yes, icons get sized down	yes, UIE becomes small dot	n/a

Table 2.5: Comparison of indoor map UI elements

2.5.2.3 Smart Space UIE

This final iteration is about Smart Space specific user interface elements, c.f. Table 2.6.

As explained in the previous sub-subsections, the **free@home** developers divided configuration and operation into two separate web apps and floor plans. When the user enlarges the floor plan, the small dots indicating actuators become large (half) round icons acting as UIE, compare Figure 2.25. In the case of a simple lamp, a click on this icon leads to a state change from off to on or vice versa. This state is reflected by the icon itself: When lamps are switched on, the icons is bright with white glow. For dimmable lamps the icon brightness depends on the current state, also displayed as percentage. It can be increased or decreased by horizontal dragging of the symbol. The same is true for blinds, where the percentage and icon describes the opening degree. For heating actors, the temperature is expressed in centigrades.

In the operation view, the user can hide device groups - categorized by heating, light, and blinds. To display only a specific category, she has to hide all other categories manually. The configuration floor plan does not display any devices state, but the device type independent of the zoom level.

The **Apple Mockup** (Figure 2.26) represents state by icon colour. During the presentation [17] the grey lamp icon is animated to yellow, the thermostat to blue, and the door lock to green.

Smart Space UIE	Architecture floor plan	Google Maps	OpenLevelUp	Munimap	free@home configuration	usage	Apple Mockup
device icons represent state	n/a	n/a	n/a	n/a	no	yes	yes
direct action with one click	n/a	n/a	n/a	n/a	no	yes	n/a

Table 2.6: Comparison of Smart Space specific UI elements

2.5.2.4 Conclusion (Floor plans as smart space UI)

First a few concluding sentences per investigation subject:

The architecture floor plans typical presentation is not suitable for a smart space control app. Google Maps' presentation provides a better overview, has the most pragmatic floor selector and implements zoom dependent details. From the technical side, OpenLevelUp and Munimap show how rendering indoor maps via vector data is best implemented. Munimap introduces the concept of attaching the floor selector to a building instead of the map view. The free@home system shows that separation of a smart space app into configuration and usage should be considered: In the configuration view the icons resize, while still presenting the same information. The free@home usage view has very good Smart Space UI elements. However, there is only one value per device that can be changed. The graphical representation from the Apple Mockup is nice, but probably does not work for all to-be smart spaces, also Apple itself has decided to not use a building plan in their iOS 10 Home App.

To derive concrete requirements, let me reiterate the individual features:

In my point of view, the suitable amount of details for smart space UI floor plans are thick lines symbolizing walls with gaps at the door passages, grey areas symbolizing furniture, and optional grey lines for windows in exterior walls. Door opening directions, socket-outlets and other labels are irrelevant and only cause confusion. Optionally suggest fixed furniture such as tables. The web maps show that colours are useful to highlight high frequented areas like corridors, staircases, elevators, or toilets.

A North-oriented floor plan is suitable for Admins, where Occupants cope better with a rectangular representation (The roles Admin and Occupant are defined in section 2.3).

From web maps, users are used to enlarge the map via mouse wheel, gestures or plus/minus buttons <R.3>.

Switching between the floors <R.4> is best implemented by listing the individual floors near the zoom buttons (cf. Google Maps).

The device marker might display the current state via different colours or icons as shown by free@home. Multiple devices at the same XY coordinates <R.5> and flexible zoom levels, require self adjusting device representations, for example markers might shrink in size or merge <R.6>.

2.5.3 Mobile apps in smart spaces

The task definition dictates that I should create a web app for tablets and PCs. This sections analyses the origin of apps, tablets, and their operation systems. The section borders the term ‘web app’ from ‘native app’, and investigates how to distribute apps for multiple platforms.

The 1991 article “The Computer for the 21st Century” [18], Mark Weiser came to the conclusion that Ubiquitous Computing requires devices in different sizes:

1. Tabs: inch-scale machines (2 inch ≈ 5 cm) that approximate active Post-It notes
2. Pads: foot-scale devices (1 foot ≈ 30 cm) that behave something like a sheet of paper, book or magazine
3. Boards: yard-scale displays (1 yard ≈ 1 m) that are the equivalent of a blackboard or bulletin board

Pads became mainstream in 2010 when Apple introduced its first iPad. It was the most successful tablet at the time of its release [19]. Others followed, especially Samsung’s Galaxy Tab, which – in view of Weiser’s definition – actually is not a Tab but a Pad.

The iPad runs Apple iOS, the Galaxy Tab runs Google Android. Both operating systems (OS) were initially developed for smart phones and later extended to support tablet devices. Third party applications for these devices can be separated into three groups:

Native apps are written in an OS specific programming language. For Apple iOS the official supported programming languages are Objective C and Swift. Google’s Android is more open, but in general all UI specific code is written in Java. Both provide UI libraries, which provide ready to use layouts, input controls, settings, dialogues and navigation elements. A native app’s source code is compiled by it’s developer and shipped in binary form to the end user. This distribution is typically done via software repositories, the so called “app stores”. In case of iOS it requires a review by Apple for each version of the app to be listed in their repository. For companies exists the possibility to distribute in-house apps via a local server, but the broad way for the mass leads through Apple’s centralized repository. For Android exist multiple stores: Most common are the Google Play Store and F-Droid.

Web apps are build in totally different way. Interactive client software in the web is typically implemented in or transpiled to JavaScript (2.5.6.1) and executed by a web browser’s scripting engine. Browsers allow web apps to hide the browser UI chrome, so a naive user might not be aware he is actually using one. In contrast to classic native apps, it is not necessary to create a new code base for every platform. Android and iOS both provide a way to add web apps to a device’s home screen. Web apps can not make use of native UI libraries. Therefore several JavaScript libraries exist to reimplement native UI elements specified by the UI guidelines (2.5.4) of the corresponding platform.

In some cases a OS feature is not yet accessible for web apps, as APIs are not defined, available or consistent over different OS releases, e.g. background notifications on iOS. To overcome this limitation a developer can bundle their web app as **hybrid app**. This is typically done using Apache Cordova (a part of a project formerly known as PhoneGap).

Cordova is a build environment which includes a local web server and a minimal native app consisting of a single web view. This minimal app is extensible with additional plugins providing bridges between JavaScript and native libraries. These plugins can include even a full browser, as some manufactures of Android devices are known for not providing OS software updates for their products and the OS web rendering engine used by the web view would be probably outdated.

Reasons to distribute a web app as hybrid app are:

- ensure an up-to-date browser (Android)
- be listed and found in the app stores by the general user
- app requires huge[8] amount of data and should run offline
- display notifications from server process while app is closed/in background (iOS)
- add widget to home screen or today view

Besides classic hybrid apps, where web apps are encapsulated, there are also other approaches. There exist 'specialized browsers' having an own markup language, with own implementations per target platform. Apart from React Native these are (mostly) always commercial and not free software. Facebook's React (see section 2.5.6.2) allows to share code between a web, iOS and Android version.

2.5.4 Human interface guidelines

The previous three subsections introduced and defined UI engineering, Usability, Smart Space UI, mobile apps and their ecosystem. Now I investigate if there are rules one can follow to build a web app with good UI and usability.

> "From the earliest days of computing, interface designers have written down guidelines to record their insights and to try to guide the efforts of future designers. The early Apple and Microsoft guidelines [for desktop computers] have been followed by dozens of guideline documents for the Web and mobile devices." [20, p. 75]

Human interface guidelines (HIG) should not be confused with style guides. Besides pure graphics, they define behaviour, gestures and provide reasoning for their author's decisions. HIGs enable different developers to produce matching components acting in an uniform way. For example, HIGs define that the input box of for each option in a group, where the user has only a single choice are round (radio buttons); whereas in multiple choice groups, the boxes are quadratic (check boxes) . For a list of current guideline documents see `http://designguidelines.com`.

The relevant HIG documents for this work are the iOS Human interface guidelines [21] and the Android Material Design specification [22]. I could not find guideline documents for platform independent web apps. Each document is related to one particular OS or platform. I found conclusions for multiple platforms only in the form of articles like [23],

[8] The exact persistent storage limit depends on the used API, the browser and other things. See `http://www.html5rocks.com/en/tutorials/offline/quota-research/` for more details.

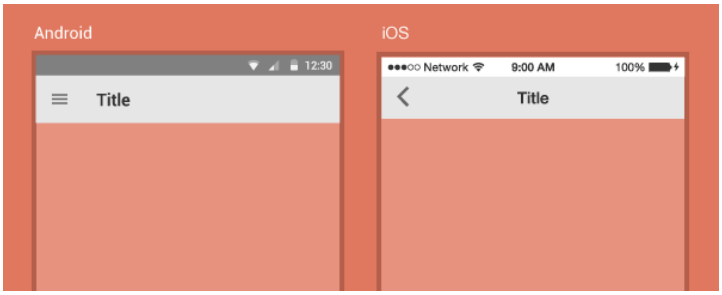


Figure 2.27: Comparison of Android (left) and iOS (right) global elements from [23]

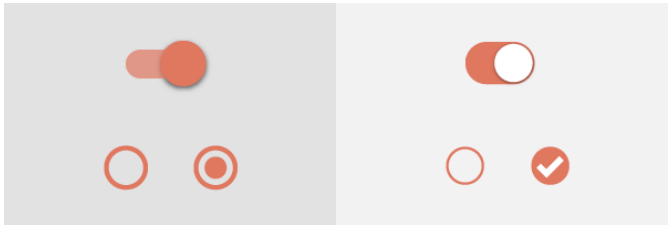


Figure 2.28: Comparison of Android (left) and iOS (right) input elements from [23]

where O'Sullivan compares OS mannerism and UI concepts: In iOS it is essential to provide a back software button in the top bar, whereas it is at the bottom of the screen for Android (compare Figure 2.27). For iOS this button should be labelled with the title of the previous view. On Android the button is not the app developers responsibility: On current devices the OS takes care of displaying a back button, previous Android hardware was even equipped an explicit hardware button. O'Sullivan recommends to give control and input elements a native feel: "As with alerts and dialogues, these controls and inputs are an area of trust and familiarity for the user. Use the native components as much as possible for these, so that people know how to use them [...]" (compare Figure 2.28). [23]

A web app developer has to decide if the app should look like a native app, or a totally different user experience should be used. Both choices have advantages and disadvantages: Particularly when HIGs are contradicting each other, an app might look totally different on one device then on another. With same UI on tablet and PC, users can help each other more easily and might get better along when switching devices (compare 2.5.5.2). Even small behaviour differences in animation and gestures stand out, when rebuilt UI elements in web apps look alike their native implementations too much. This effect is sometimes called the "uncanny valley". For example, I came across of switch implementations, reacting only to clicks/touches and not to swipes like their native counterpart.

In context of the DS2OS project, I would recommend to create an own set of guidelines, to which the UI extensions (see section 2.2.4) should conform to. For example: Components should behave similarly, whether the user controls a lamp or a heater. Users should recognize at first glance where they can perform actions or only state is displayed. Input elements must only be used, if the user actually has the rights to modify the value. For

a regular lamp, there must not be a disabled switch, but a text label with ‘on’ or ‘off’, or a corresponding image. A device icon colour represents the device’s category, e.g. yellow for light, blue for HVAC and green for security. Icons of powered off devices are grey. Further rules can be drawn from literature e.g. “Present digital values only when knowledge of numerical values is necessary and useful” [20, p. 77].

To return to the initial question: In my view there is currently no cookie-cutter approach to get web apps with good UI and usability. “Despite of scientists having tried to derive models, principles, and theories – UI design is a complex and highly creative process” [20, p. 107]. “Creative processes are notoriously difficult to study, but well-documented examples of success stories will inform and inspire.” [20, p. 143]

2.5.5 Self-adapting UI

The previous subsections introduced and defined UI engineering, Usability, Smart Space UI, mobile apps and their ecosystem. Now I investigate how to build self-adapting UI.

In an ideal world with infinite UI designers and money, every smart space would have an own app optimized for this space and its users, which was developed as described in the previous subsections. But in today’s reality this does not scale: There are not enough well educated UI designers and it would not be economically viable, as their wage is comparatively high. Thus to offer smart spaces with good UI to many people, a different, more scalable approach is needed. In the DS2OS project, we named this approach self-adapting UI.

The key results of this subsection are: Only programmatic UI can be self-adapting. The initial UI design must already be good, developers can not pass on their task to users. Customization is only easy when the user has previously designed options to choose from (which we named extensions). The following subsections state reasons for these results and provide further details.

2.5.5.1 Static UI versus Programmatic UI

The vertical labels at the left border of Figure 2.29 categorize the UIs introduced in section 2.1 and chapter 3 into static and programmatic UI.

Static UI can only be modified by replacing it with another UI or with some amount of manual work. For some, only (external) experts can change the configuration – e.g. a wall switch wired to the ceiling lamp by an electrician, or the touch panel introduced in 2.1.5.4. Others have a fixed layout of the buttons, but their configuration can be changed by it’s users, compare the remote controls and wall push buttons from subsection 2.1.2.

Programmatic UI, on the other hand, can use nested loops (while, for) and branches (if) to generate as may buttons as needed. Basically all products from chapter 3 fall into this category, including the UI prototype resulting from this work.

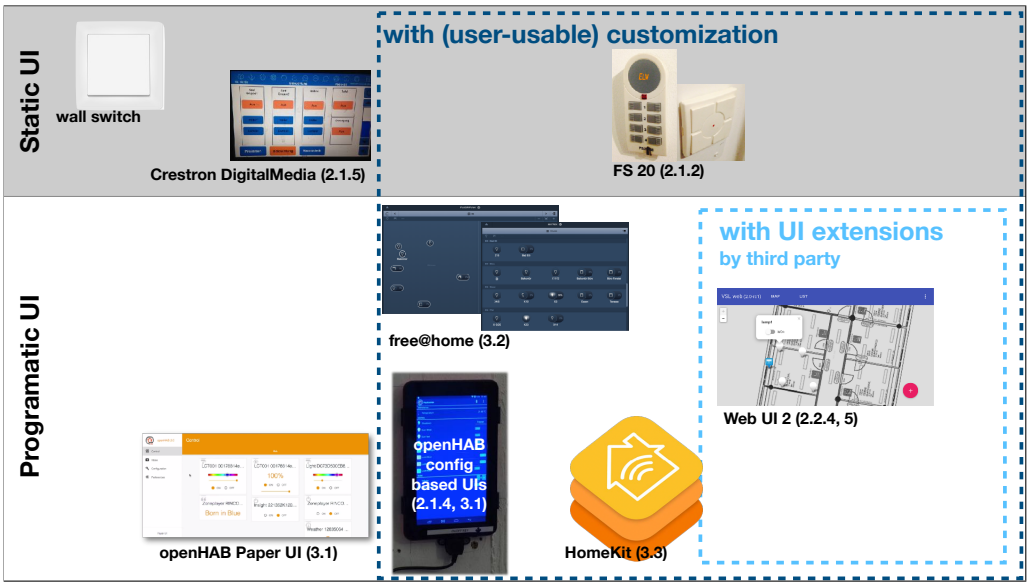


Figure 2.29: Categorization of UIs from section 2.1 and chapter 3

2.5.5.2 Customization by user

The fist subdivision of Figure 2.29 (in dark blue) contains only UI having (user-usable) customization features.

When adding customization features, former research [24, p. 12] observed:

- Users do not dare to use the customization features, because they are afraid to break something.
- The initial design has already to be good – one can not leave it to the users themselves
- Customization is easy only, when a pool of building blocks to choose from exist.

In the static UI section, only the remotes from subsection 2.1.2 are customizable, as users can reassign the buttons to other devices themselves.

The openHAB project provides multiple UIs: The Paper UI only lists the available devices and is not customizable by the user. To satisfy the users' customization needs, the project created the 'Basic UI' with the same web application framework as the Paper UI. The Basic UI, the older UIs (see subsection 2.1.4), and the native apps build there UI based on configuration files (aka 'sitemap'). There these UIs are part of the customization subsection.

free@home (3.2) and HomeKit (3.3) both allows the user to move devices between rooms via UI and create favourites. Due to the integrated floor plan, free@home even allows moving within a room.

For more details about the findings from the beginning of this sub-subsection, see following quotes from [24, p. 12]:

> "The ideal solution to the usability question might be to leave the design of the interface up to individual users. Just provide sufficient customization flexibility, and all users can have exactly the interface they like. Studies have shown, however, that users do not customize their interfaces even when such facilities are available [Jørgensen and Sauer 1990]. One novice user exclaimed, 'I did not dare touch them [the customization features] in case something went wrong.' Therefore, **a good initial interface is needed to support novice users**. Expert users (especially programmers) do use customization features, but there are still compelling reasons not to rely on user customization as the main element of user interface design. "
>
> "First, customization is easy only if it builds on a coherent design with good **previously designed options from which to choose.** Second, the customization feature itself will need a user interface and will thus add to the complexity of the system and to the users' learning load. Third, too much customization leads each user to have a **wildly different interface** from the interfaces used by other users. Such interface variety makes it **difficult to get help from colleagues**, even though that is the help method rated highest by novice and expert users [Mack and Nielsen 1987]. And fourth, users may not always make the most appropriate design decisions. "

The UI has to be good enough the first time a new user comes in contact with it, particularly in installations with multiple users. A new occupant should not have to gather information from colleagues for some days to get an usable interface.

For commercial buildings I could even go a step further and make the following case: When the users have the opportunity to customize the UI for themselves, it's bad for the overall system: The power users no longer use the default UI and therefore no longer know how bad it is. There is no more pressure on the operator/developer to improve the default configuration. When you give the power users the possibility to customize, how are you ensuring that the new solutions are flowing back into the community? How to notify power users about a new default configuration, which is potentially better than their own – without annoying them?

2.5.5.3 Extensions by third party from catalogue/store

The last subdivision Figure 2.29 contains only extensible UI. As far as known to the author all existing smart space UI lacks this feature – only the UI to be created as part of this work is part of this group.

As already quoted above: "customization is easy only if it builds on a coherent design with good previously designed options from which to choose." [24] When we apply the 'app economy' part of the VSL services concept also to this 'options catalogue', we get portable UI extensions distributed via a store. Nevertheless, designers and developers need guidelines so the extensions are of good quality and consistent with each other, see section 2.5.4. The extension store idea was already described in more detail by following scenario from subsubsection 2.2.4.1:

The user has radio controllable LED bulbs in his living room. The software control element is build with three sliders, adjusting the red, green and blue value of the light (compare left part of Figure 2.10). When he wants to dim the light, all three sliders have to be adjusted, and he always needs some time to find the right colour. As he is dissatisfied with this situation, he opens the "Extension Store". There he finds an extension that has only one slider for the colour, a second slider for the intensity and a toggle switch to turn the light off (compare right part of Figure 2.10). He hits the install button and the initial control element is replaced with the new one from the extension. Now he can switch off or dim the lights without destroying the colour settings.

2.5.5.4 Self-adapting

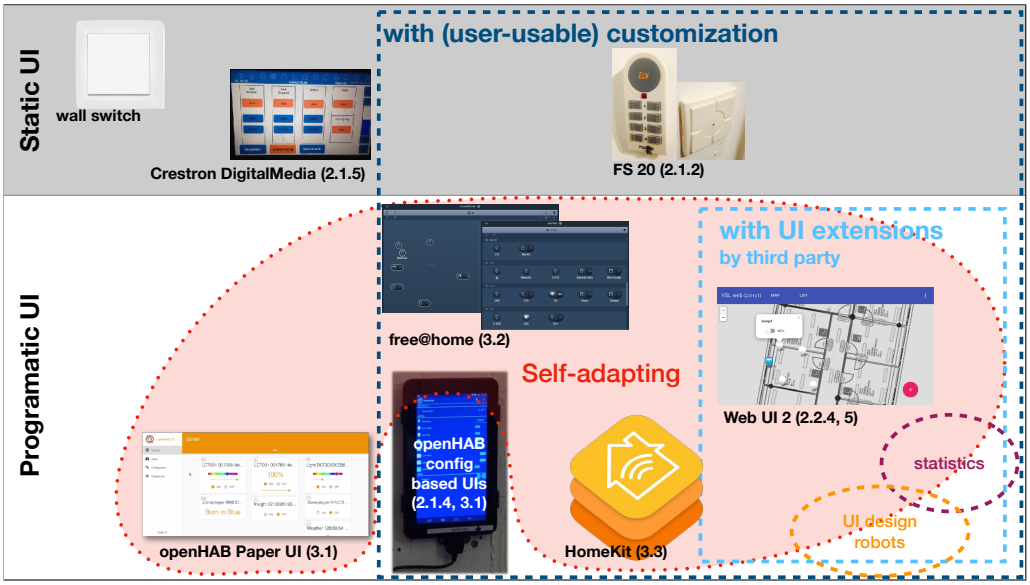


Figure 2.30: Categorization with self-adapting UI highlighted in red

So far the overall categorization - now to 'self-adapting UI'. Self-adapting combined with extensible UI was already described in more detail with the corresponding scenario in subsubsection 2.2.4.2. It is closely related to <R.16>. When interpreting self-adapting as 'A new device comes into the system and the user can handle it inside the GUI without having to manually change any configuration files', the red highlighted classification shown in Figure 2.30 emerges.

The Paper UI from the openHAB project (3.1), features an inbox view, where new found devices appear – after the corresponding binding was installed with in an other section of the Paper UI. Apart from dynamic group listings, the classic, configuration file based UIs are not self-adapting, and therefore mostly let out of the red area. Free@home (3.2) has a similar concept to the inbox: In lower area of the 'Placement' step (Figure 3.7) new connected devices pop up automatically. The user than moves the devices from

this lower bar to the actual location of the device in the floor plan. The further process is described in detail in section 3.2. HomeKit (3.3) also supports auto discovery of new devices, but requires the one user with administration rights, to scan or enter the new devices security code. After that the new devices is assigned to a room and configured accordingly.

The reminder of this section is about two small areas at the bottom right of Figure 2.30 involving machine learning.

2.5.5.5 Customization/Optimization by the UI itself

UI customizations can be done by either humans, e.g. the users (2.5.5.2) or third-party developers (2.5.5.3), but to some degree the UI might be able to optimize itself for the users. For example – the UI can track usage and use this statistics for automatic creation of favourites, offer short cuts for frequently used multi step processes, or change size of device icons in the map view based on (this user's) usage. Storing usage statistics in distributed applications is indeed a challenge. It might be a privacy issue when not stored decentralized.

An other automatic optimization could be based on the on the 7±2 rule. This 'rule' goes back to one of the most highly cited papers in psychology, published 1956 by George Miller and is actually misinterpreted most times:

> "The value 7±2 as a measure of short term memory is an urban legend. It only applies to speakers of English attempting to remember a sequence of digits. Actual human memory performance depends on many factors and cannot be approximated by a numeric value." [25]

It does not matter how many items there are, as long as they are grouped in 7±2 items per group. Even more flat menu hierarchies with many items per level work better than deep menu hierarchies. A piece of software could actually implement such a behaviour by implementing a set of heuristics. For example: Try to group devices by different attributes like type, location etc. and count the group members. If the number of each group is below 10 everything is fine, otherwise try different attributes or split the group in half.

2.5.5.6 (Semi) automatic generation of new UI extensions

Suppose that the way designers and developers work can be completely formalized into rules and models in Future: A kind of UI creation robot could be created, implementing the process (2.5.1), guidelines and principles (both 2.5.4) as good as it can. The robot creates new components and publishes them in a store. Users try them out and give ratings. For bigger installations it could perform automated A-B tests with new users: one half of the user base gets displayed version A, the other half version B. After four weeks the system decides on usage statistics which version worked better and uses this for all users in that role as new default. But "Despite of scientists having tried to derive

models, guidelines, principles, and theories – UI design is a complex and highly creative process" [20, p. 107]. While there are first attempts to teach computers creativity, yet as of today developers and designers reach their goal faster when they use their UI knowledge directly to build components, instead of trying to transform their knowledge into a model computers understand. For example: A developer gets to know about the new `<input type="color"/>` in HTML5, reads the specification and creates a new component for colour selection using this element.

2.5.6 Front-end building blocks

In the DS2OS project it was decided that we do not want to build an own app for each OS and therefore use web apps.

As web apps are typically written in JavaScript dialects, I give an short introduction into the environment and mindset of this scripting language and developers using it in section 2.5.6.1. The final two sections are about libraries and frameworks providing toolboxes and guidelines for web app development. Section 2.5.6.2 gives an overview over the current generation web application frameworks and section 2.5.6.3 looks into map libraries.

2.5.6.1 JavaScript

The DS2OS project manager (Marc-Oliver Pahl) decided that I should implement the graphical user interface as web app. This means there is a client side JavaScript runtime, allowing advanced logic and interaction. Obviously this runtime needs JavaScript source code. Theoretically today one can automatically transform source code from nearly every[9] other language to JavaScript. But many developers consider programming directly in JavaScript as the natural way.

JavaScript was first introduced within the Netscape Navigator web browser in 1995. Other browsers adopted this idea and added similar scripting languages into their feature set. This lead to several different implementations with different features – partly incompatible. JavaScript developers nearly had the double amount of development effort, and a software had to be tested in every supported browser manually, e.g. one had to use a different method to get the contents of the browser address bar in Internet Explorer then in other browsers.

As JavaScript is a dynamic language, functions can be injected into the environment during runtime, as show in Listing 9. In this example the function Date.now() would be implemented in JavaScript if the browser lacks a native implementation. A developer can now use Date.now() in her code without having to think about the actual details where this function comes from. Today such patches to a web browser are called shims or polyfills[10].

[9] `https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS`
[10] `https://remysharp.com/2010/10/08/what-is-a-polyfill`

```
if (!Date.now) {
    Date.now = function now() {
        return new Date().getTime();
    };
}
```

Listing 9: JavaScript Shim for Date.now() [from `http://stackoverflow.com/a/31721437`]

Developers can use newest APIs today, while polyfills are taking care of providing JavaScript implementations for old browsers <R.22>. The developer only has to include a library providing these polyfills. Speaking of libraries: Managing libraries and their dependencies manually is not really common any more. As in other programming languages there are packet managers dealing with downloading and updating libraries used in a project. For example pip in Python or CPAN in Perl. In JavaScript this tools are called npm, bower or yarn <R.23>.

Especially since the new app should be flexible and extensible, programming in an object oriented fashion (OOP) is reasonable. In OOP there is typically only one class definition per file. Existing classes are reused in other classes either by reference or sub-typing. In comparison to Java or Python, JavaScript did not specify how to define classes/modules till mid-2015. As JavaScript is a dynamic language, there are multiple approaches how to define modules (see CommonJS, AMD, ES6, GCT). Therefore developers have to choose a way of implementing OOP for each project. For people who think the JavaScript community should agree on one standard, there is also light in the sky, at last: ECMAScript, a scripting-language specification initially based on JavaScript. As of 2014 JavaScript can be seen as an implementation of ECMAScript 5 published in 2009. In mid-2015 the sixth edition of ECMAScript (ES6 aka ES2015) has been released, adding native syntax for classes and modules (and other features).

But with ES6 developers have once again the problems described above: It takes time till the new standard has been implemented by the browser manufacturers, and even more time till all end users have updated their browsers. Should developers wait and still write 'old code'? Of course there is a solution: source-to-source compilers. The new ES6 syntax is converted to equivalent JavaScript/ES5 code by an IDE. These 'converters' are sometimes called **transpilers** as they transform source code to other source code.

There is also a large group of people who want to have types in JavaScript. With the transpiler technology there is also a solution for these people: The TypeScript language/dialect, which is transpiled to JavaScript.

To conclude: When developers start a new JavaScript project, they have to make several decisions: Which language version, build system, package manger, and libraries? Some libraries already come with a set of these decisions taken, which are then typically called framework.

2.5.6.2 Web app frameworks

In simple terms web app frameworks (WAF) are libraries implementing the model-view-controller pattern – or one of its successors. Some WAFs provide/dictate a whole development environment (see previous section), others force the developer to search for the matching building blocks to come to a full app.

To simplify the decision-making process, the project TodoMVC might be helpful. Quoting their website `http://todomvc.com`:

> "Developers these days are spoiled with choice when it comes to selecting an MV* framework for structuring and organizing their JavaScript web apps.
>
> Backbone, Ember, AngularJS… the list of new and stable solutions continues to grow, but just how do you decide on which to use in a sea of so many options?
>
> To help solve this problem, we created TodoMVC - a project which offers the same todo application implemented using MV* concepts in most of the popular JavaScript MV* frameworks of today."

As of February 2016 there are 64 different implementations with various frameworks. This collection of implementations for the same problem allows direct comparison based on the actual source code. Developers can form their own informed opinion.

For me following criteria are required: The framework should embrace HTML5, use native GUI where possible (= ready for Web Components), and therefore does not reimplement everything itself. To ensure long term support, there should be an active community, which also can be measured by the number of available plugins. The corresponding styles and templates should have support for tablets and desktop PCs and not just for smartphones. I prefer the all-tools-included framework approach, which for me includes a state of the art URL router and template engine, using virtual DOM diffing.

In the end, three frameworks made it to the short-list, which all published new major releases during the research period of this thesis: Ember released version 2.0 in August 2015, Facebook published React 0.14 in October, and Google released the beta version of Angular 2.0 under the name Angular.io in December. For a more detailed comparison, refer to [26]. The author of this blog article compares these three frameworks in detail and concludes:

> "Of these frameworks, Ember was the quickest to get something started. Immediately you have a web server that reloads the page on changes and best practices right out of the box. With the other two, you might spend time configuring Webpack or Gulp to get a project off the ground. You might fiddle with how you want the project laid out. Or you might spend time searching for a boilerplate project to copy. By virtue of being opinionated, Ember removes all that friction.

> Yet, for me, Ember took the longest to learn of the three. For such a small toy project, it felt like overkill. Also, it seems there are specific ways Ember wants you to do things and going outside of that is difficult. [...] To me that signals Ember would scale well to long-lived projects with many developers. In comparison, the other two frameworks eagerly played along with whatever I wanted to do. Angular 2.0 surprised me a little. It is nothing like Angular 1.X, but it was easy to build the app once I found some examples to learn from.
>
> It is easy to see why these three frameworks are so popular. They all have a lot of strengths. Because of that, **I suggest you learn and work with all three**."

So let me conclude: If you want to build a modern web app with a current generation of WAF, Angular, React and Ember are more or less equivalent. The decision depends on the associated ecosystem: Which other bricks are available for this WAF? The largest blocks for our app are the floor plan and the UI elements. The requirements regarding a UI elements library depends on the decision whether the app should look like a native app or not (see section 2.5.4). If you decide yes, this would be an advantage for Angular. The Angular based Ionic project provides bricks which solve this problem: The developer can place a switch place-holder which is then automatically replaced with an Android or iOS implementation. The other large block – libraries handling geo data – is discussed in the section 2.5.6.3.

2.5.6.3 Web map libraries

Web map libraries enable embedding of spatial data like aerial images, street maps or vector data. External street maps or aerial images are typically incorporated as bitmap tiles. The library stitches these tiles in the correct way and keeps track which tile is representing which coordinates. When the user moves or zooms into these data, the library loads only the new required bits or exchanges one tile through four new ones to increase resolution. Besides, these libraries also allow to display and set pins (alias markers) or render lines and areas from vector data e.g. from a GeoJSON file. I present three implementations: OpenLayers, Leaflet and Cesium.

OpenLayers is the oldest: OpenLayers 2 was already introduced into the DS2OS project with the first web UI prototype. Meanwhile, there is the completely revised version 3. The new API is incompatible with the old one. OpenLayers has numerous features, whereby the inexperienced developer can not get easy into. This gap was filled by **Leaflet**, created as a more lightweight alternative. For Leaflet there are many ready to use plugins, such as marker clustering, heat maps etc. It also supports marker detail popup's out of the box. Leaflet only supports one set of tile grids at a time, where OpenLayers can handle totally different tile grids in one map. This means that with Leaflet it is not possible to lay the current building plan tiles over an aerial view. Possible solutions: create a new building plan tile-set, or replace it with vector data.

Cesium is the most recent of the libraries presented in this sub-subsection and is out of the ordinary: "Cesium is a JavaScript library for creating 3D globes and 2D maps in a web browser without a plugin. It uses WebGL for hardware-accelerated graphics, and is cross-platform, cross-browser, and tuned for dynamic-data visualization." [http://cesiumjs.org]. Probably an interesting candidate for the future.

The decision for a library depends not only on the same, but also if an adapter to the WAF already exists or how easy it is to implement. For Angular there is even a general adapter called AzimuthJS[11], allowing to switch between OpenLayers and Leaflet per configuration option. But there are also two classic 1:1 adapters from Angular for OpenLayers 3 called ngeo[12] and Anol[13]. For Ember there also exist adapters for both, but in different quality: ember-leaflet[14] is a quite active project, where ol3-ember[15] is outdated. The final front-end design including my decisions is presented in section 4.3.

[11] http://mpriour.github.io/azimuthjs/
[12] https://github.com/camptocamp/ngeo
[13] https://github.com/omniscale/anol
[14] http://www.ember-leaflet.com
[15] http://boundlessgeo.com/2014/02/openlayers-3-ember/

2.6 Requirements list

This list concludes the analysis chapter by summarizing the resulting requirements. The functional requirements are split into front-end and back-end. Notations in round brackets – e.g. *(1.2.3.4)* – are references to the text section they are derived from. There they are in turn referenced with <R.x>.

Front end requirements

<R.1> UI is customizable via extensions (2.2.4)
<R.2> UI renders devices on floor plan

<R.3> Floor plan is zoomable via gestures (2.5.2.2)
<R.4> UI supports buildings with multiple floors by providing selector to switch between different floor plans (2.5.2.2)
<R.5> Floor plan allows multiple devices at the same XY coordinates (2.1.4)
<R.6> Icons become smaller/grouped in lower zoom levels (2.5.2.2)
<R.7> Shape and place of rooms are modelled (2.5.2.1)

<R.8> UI allows tablet usage in landscape and portrait mode (3.2, 2.1.4)
<R.9> Login via username/password, certificate or other identity mechanisms should be possible
<R.10> Tablets should be usable as non-personal devices: One tablet can be used by multiple users with different permissions (2.1.4)
<R.11> UI allows system administration and configuration
<R.12> Admin and Usage scenarios are split into different UI views (c.f. free@home in each sub-subsection of 2.5.2)
<R.13> Debugging: The app should enable the user to understand why something does not work and support her at investigating why that happened. (2.1.2)
<R.14> Multiple home or sites in one app or separate apps on the same device should be possible.

Backend system requirements

<R.15> Is it possible to restrict users to only be able to control certain and not all devices
<R.16> New devices are automatically discovered, without user input like devices address etc.
<R.17> Systems allows installation of extensions via central repository
<R.18> Systems knows concepts of rooms, device types, etc.

Constraints/Pseudo requirements

<R.19> UI is implemented as web app.
<R.20> Web app uses current generation web application framework (WAF): Angular, React or Ember. (2.5.6.2)
<R.21> Fast development cycle: Real world usability can only advance when suggested improvements by users can be implemented and deployed in a reasonable amount of time. (2.1.5.4)
<R.22> Developers should be able to use newest HTML5/JavaScript APIs without having to take care of older browsers. (2.5.6.1)
<R.23> External libraries and their dependencies should be managed by tools and not manually. (2.5.6.1)
<R.24> System setup and usage should work without internet connection (2.1.3).

Quality requirements

<R.25> The interface should be self explaining, so that users can use it without having to read a manual.
<R.26> How long does it take to start the app, time from touch to lamp on, time from window opened to status change.
<R.27> If something is done automatically by the system, users want to be able to understand and investigate why that happened. (2.1.5.4)
<R.28> It should be possible for future developers to improve the prototype without an extensive training period.
<R.29> Third-party developers should be able to add new UI elements in an easy way

Requirements 1–20 are used to evaluated the UIs presented in chapter 3 (Related Work) and reappear in collected form in section 3.4. Their fulfilment through this work's results is checked during chapter 6 and 7.

Chapter 3

Related work

After having analysed the problem domain and deduced requirements, we now investigate how existing products are addressing these problems. Specifically, we take a closely look at the openHAB project, ABB-free@home and Apple's HomeKit. The individual requirements are referenced with <R.x>. If a requirement is met by the product/prototype its referenced with <R.x>✓, if not it is crossed out: ~~<R.x>~~. The chapter is closed by an comparison table in section 3.4.

3.1 openHAB

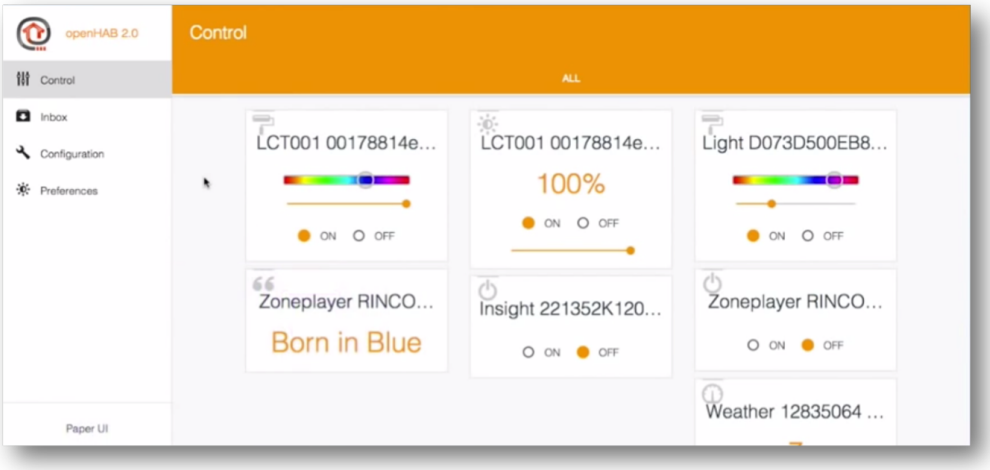


Figure 3.1: openHAB 2.0 Paper UI: control center [27, still from alpha1 preview video]

openHAB is a vendor and technology agnostic open source home automation software and community. Some parts were split into an own project called Eclipse Smart Home (ESH). As its name suggests this new project is hosted at the Eclipse Foundation. ESH is

used by vendors like Deutsche Telekom AG/QIVICON or JUNG Elektro GmbH [28, slide 20]. As of June 2016 the development of ESH is ongoing and openHAB 2 is still in a beta state. Version 1 of this system was already introduced in section 2.1.4.

There are native iOS and Android, as well as multiple web based UI implementations <R.19>✓. The UI elements are not customizable ~~<R.1>~~↯. New ones can only be added through new openHAB releases, forking an existing UI, or creating a new one. They do not use floor plans ~~<R.2>~~↯, but are supporting multiple floors <R.4> through a workaround.

In a vanilla openHAB 1, the system administration had to be done manually with a text editor. The design was flexible enough to allow a third party to develop a graphical administration web UI called HABmin. With openHAB 2 comes a new administration app <R.11> featuring auto-discovery <R.16>✓. This web app (see Figure 3.1) is implemented using the Angular library, which is a current generation WAF <R.20>✓ and allows installation of extensions from a central repository <R.17>✓. The UI design is responsive and works in landscape and portrait mode <R.8>✓.

With openHAB 2 beta1 the split between configuration (Paper UI) and general usage/-operation (Basic UI) was introduced <R.12>✓. The administrator can create users with user name and password <R.9>✓, but not restrict their rights ~~<R.15>~~↯ as different roles are currently not supported. Therefore non-personal tablets with different permission modes can not be implemented without workarounds ~~<R.10>~~↯.

In openHAB 1.x the data model and UI based on a menu tree. The menu structure is defined via a *sitemap* file, which references *widgets* which reference *items*, which reference a *binding*. Each item (attribute of a device) was typically added to a group representing the room. This room-group was then added to a group representing the floor. The floor-group was referenced in the sitemap. Alternatively, a device item could also be referenced directly in the sitemap. If you want to implement an all 'lights off switch', you have to create an additional group for all lamps and assign each lamp manually. With openHAB 2 this structure was reversed: Now a binding can provide multiple *things*, which have one ore more *channels*, which can be mapped to *items*. Things can have a physical location, but if a location is a room name or a coordinate is not defined. As far as I understood, the openHAB development team has not decided yet [1] [2] [3] [4], how exactly to tackle this problem in version 2. As of June 2016 shape and place of rooms are not modelled ~~<R.7>~~↯.

Side note: If you like to investigate the development of web UIs over time, openHAB's UI evolution is actually an interesting case study: The *Classic UI* (Figure 3.2) is a set of web pages styled to look like iOS 6 App. *GreenT* (Figure 3.3) is an actual web app implemented with Sencha Touch 2.0, and openHAB 2.0 features the *Paper UI* (Figure 3.1) and *Basic UI* both created with help of Angular (see 2.5.6.2).

[1] https://github.com/eclipse/smarthome/issues/1083
[2] https://github.com/eclipse/smarthome/pull/1019#issuecomment-183245341
[3] https://github.com/eclipse/smarthome/issues/1093
[4] https://github.com/eclipse/smarthome/issues/582

Figure 3.2: openHAB classic Web UI, based on the WebApp.Net framework [from `http://openhab.org`]

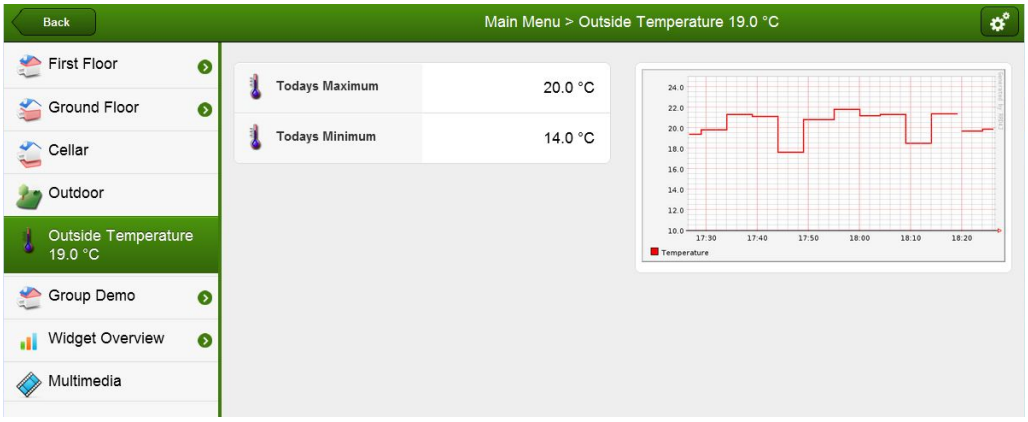


Figure 3.3: openHAB GreenT Web UI, implemented with Sencha Touch framework [from `http://openhab.org`]

3.2 free@home

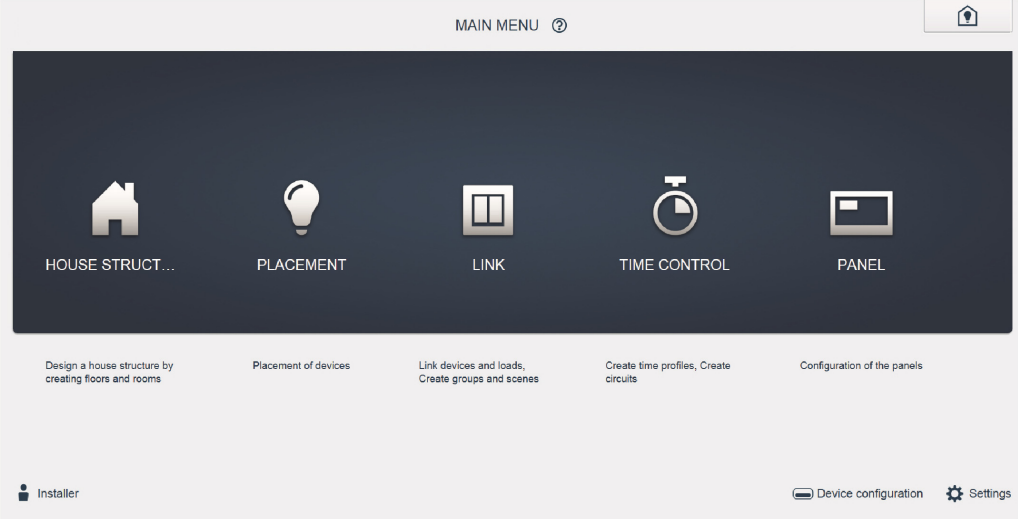


Figure 3.4: free@home main menu from [16]

The home automation system free@home is developed by Busch-Jaeger Elektro GmbH and marketed outside of Germany by its Swiss mother company ABB Ltd. Thus the German name “Busch-free@home” translates to “ABB-free@home”. See also sec. 2.5.2.

The UI is not customizable <~~R.1~~>↯ but features a zoomable floor plan <R.2>✓ <R.3>✓, supporting multiple floors <R.4>✓. The user can only approximate a room’s geometry. The coordinates have symbolic meaning and are only a vague reference to the actual building. Therefore, devices on top of each other <~~R.5~~>↯ have to be untangled by moving icons. When zooming out, icons switch to a smaller version <R.6>✓. Anyway, shape and place of rooms are modelled <R.7>✓.

The system administration is fully done within the app <R.11>✓ and the UI is strictly separated in administration and usage/operation <R.12>✓. The working area of the administration screens are all designed in a consistent way: About three-fourths of the horizontal screen space is dedicated to the floor plan, the other quarter is either a list or configuration view. The lower bar always displays things to add (compare Figure 3.5 to 3.8). The general usage interface is also floor plan based, but only displays actuators and no sensors like wall switches.

The app is implemented as web app <R.19>✓ using the Qooxdoo library <~~R.20~~>↯. It’s designed to only work in landscape mode <~~R.8~~>↯. The vendor provides a hybrid app for iOS and Android, which auto-discovers the central gateway running a web server (called “SysAP”).

As this system is designed for homes, there are only three different access levels <~~R.15~~>↯: *Operation*: Can only operate devices and can not make permanent changes to the system; *Configuration*: Can not make changes critical to the system (changing settings related

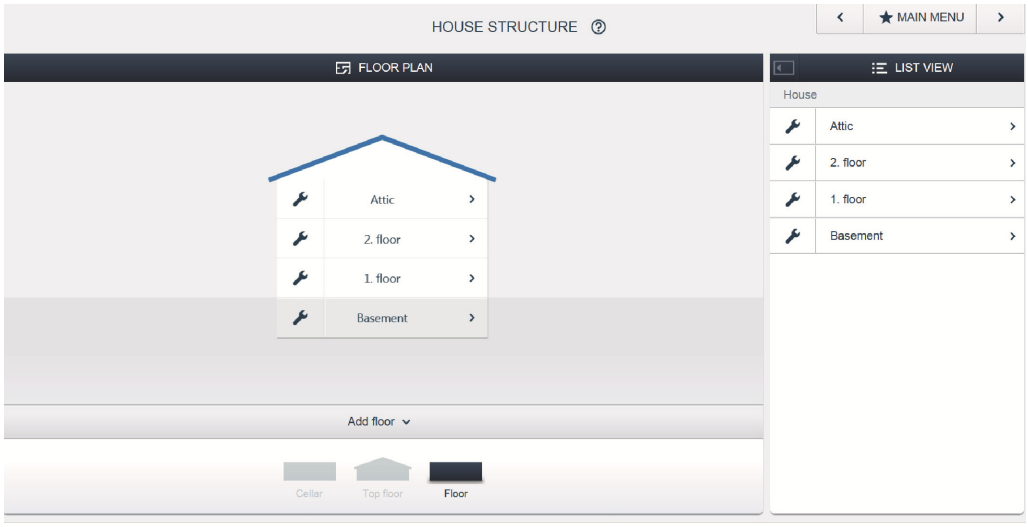


Figure 3.5: free@home: Adding floors from [16]

to the bundling of channels for dimmers, re-configuration of binary inputs); *Fitter*: Electrician, has all access rights (Master reset, creation of EF data protection). [16, p. 18]

The system is not vendor-independent. There is no store through which additional apps can be installed <~~R.17~~>↯. Apart from a Phillips Hue integration and hacks via binary inputs and outputs, there is no official way to integrate systems from other manufacturers. There is an artificial limit of 48 devices, where a 8 channel actuator with 8 additional binary inputs counts as one device, but each Hue bulb also counts as one device. The system knows the concepts of rooms and the vendor's device types <R.18>✓.

To understand the concepts of this UI in more detail, let's assume the user has just installed the app in their tablet and using it for the first time: After selecting the system language and setting an admin password <R.9>✓ the user is presented the main menu (Figure 3.4). The user first has to create floors (Figure 3.5), then individual rooms. Only simple rooms with four or six corners are possible (cf. lower bar in Figure 3.6).

After that, the user goes to the next step: Placement. The system has detected all connected devices <R.16>✓ in the background and lists the available ones in the lower bar (Figure 3.7). The user moves the icon from this lower bar up to their location in the floor plan. Let's assume the user places a light, as shown in Figure 3.7: After dropping the icon, a dialogue pops up listing all actuators having a free lighting control channel. They now have to identify the actuator by either local device operation (the actuator has a dedicated 'Ident' hardware button) or by a three character code, generated from the serial number. The dialogue screen allows to trigger each channel of the actuator to find the correct one. Placing the corresponding wall button is easier: The user places a touch sensor icon from the lower bar on the floor plan, and selects it by simply pushing the corresponding wall push-button ("rocker"). When the lower bar is empty or all used devices/channels are placed, they can continue with the final step: Linking.

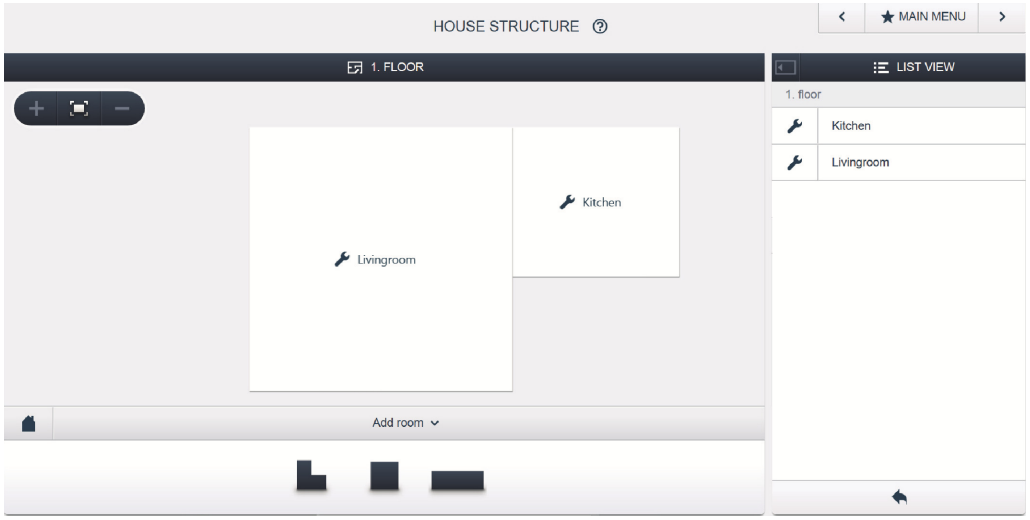


Figure 3.6: free@home: Adding rooms from [16]

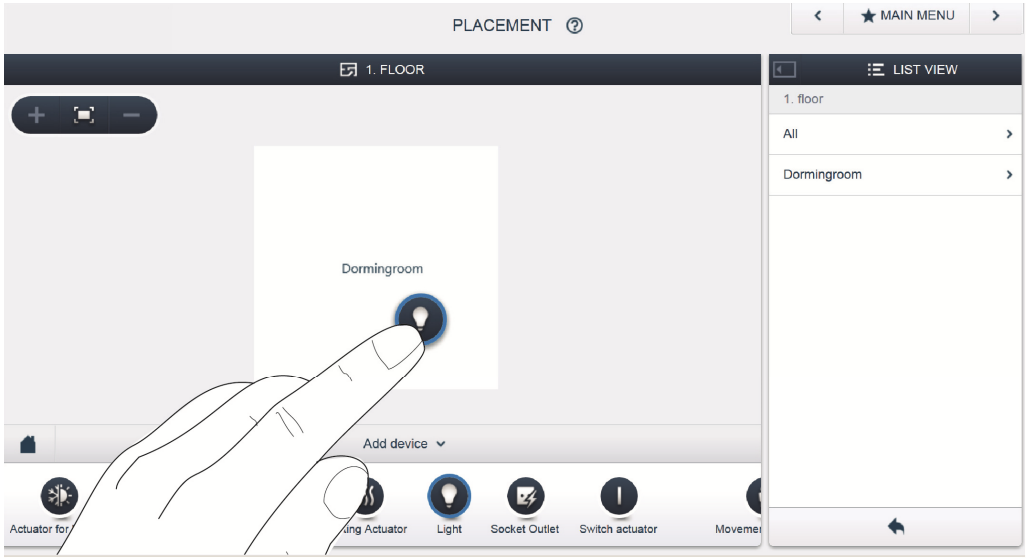


Figure 3.7: free@home: Adding devices from [16]

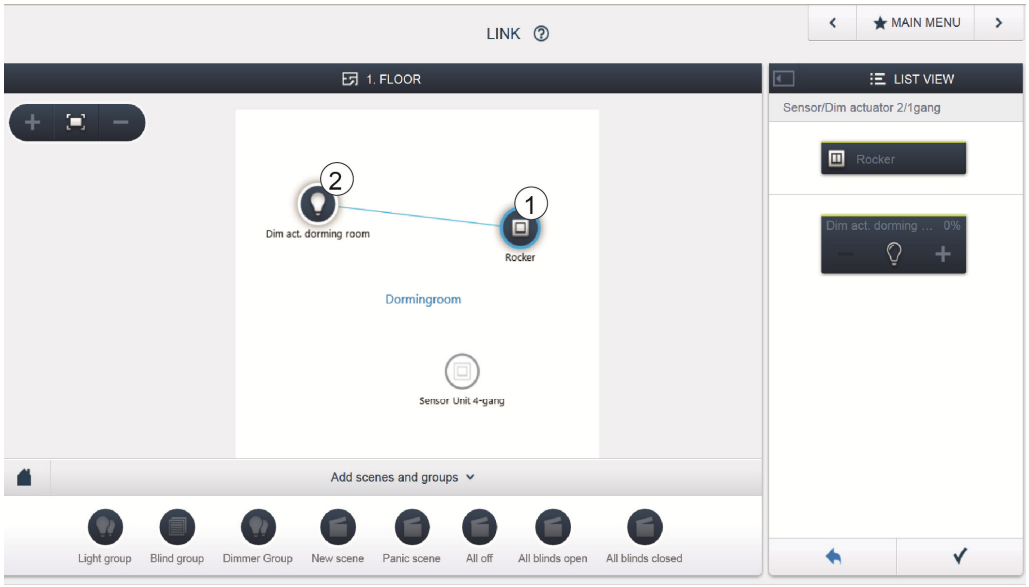


Figure 3.8: free@home: Linking devices or adding scenes/groups from [16]

In the third screen (Figure 3.8) the user connects the sensors with the actuators. In our example, the user first touches the desired sensor icon and connects the lamps by touching each one. The system draws a blue line between the connected devices. They have to confirm the new configuration with the blue tick at the bottom right. Now they can check the wall button changed its behaviour by touching it.

This screen also allows to add scenes and groups by moving them up into the floor-plan from the lower bar. The different options differ in their default settings. For example a blind group icon automatically connects itself to all blind icons inside the room it is placed in. A scene differs from a group by storage of the setting of its assigned devices. To configure a scene, they modify the state of the devices like normal operation – either through the assigned hardware buttons or in the app – during the scene is opened. These states are stored when the blue tick at the bottom right is touched and can now be recalled by activating this scene.

I am not going into details regarding the last two large buttons of the main menu (Figure 3.4): From a DS2OS view, they are only two other services: One provides orchestration services based on time or sunrise/sunset; the other allows to configure the buttons of an optional touch panel.

The two users I observed had problems with following details: Especially in the beginning, users switch constantly back and forth between the Placement and Linking step. As they are optical difficult to differentiate, different colours or backgrounds would be helpful. When linking devices, one has to confirm the new configuration with the blue tick at the bottom right: Users unwillingly discarded their changes as they forgot to click the blue tick and were confused, why their new connection did not work.

3.3 HomeKit

HomeKit is a home automation library from Apple Inc. It was announced as a pure interoperability framework in 2014. Apple only provided a natural language UI as part of Siri, data models, API and back-end. In June 2016, as part of iOS 10, Apple's own "Home App" and therefore GUI was announced. As this section was written between announcement and release in September 2016, it does not include any analysis of this app, but only the underlying technology.

HomeKit has currently no support for a flexible UI with third-party UI elements <~~R.1~~>↯. If developers want to try new concepts, they need to implement these as part of a separate app e.g. by using the HomeKit sample code. HomeKit does not use floor plans and is menu based <~~R.2~~>↯. The data model forces the user to assign each device to a room <R.18>✓, which in turn can be grouped by so-called zones e.g. 'Upstairs' <R.4>. The specific geometry and position of the individual rooms and equipment is not registered <~~R.7~~>↯.

It is possible to grant individual users access to specific devices <R.15>✓. The login is done via Apple ID (mail address and password) and optionally a second factor <R.9>✓. HomeKit depends on iCloud Keychain sync, thus usage of different logins on the same device, becomes hard to handle <~~R.10~~>↯. Probably Apple is prioritizing high device sales which might be in conflict with multi-user features in iOS. (Why to only sell one when you can sell two for twice the price.)

Administration can be done graphically within most HomeKit apps <R.11>. HomeKit apps can deal with multiple homes <R.14>✓ and support auto discovery <R.16>✓ via Bluetooth LE and Bonjour/mDNS (Wi-Fi). Apple itself does not offer a HomeKit web app <~~R.19~~>↯. If they do in the future, I expect them to use Ember <R.20> as it is used on `http://apple.com`.

Apple distributes extensions typically by reusing their existing iOS App Store <R.17>✓; compare content filters, audio plugins, or keyboard layouts. In these cases, the app in the store only serves as a "delivery package". In addition to the device types curated by Apple, custom types are possible but not controllable via Siri or foreign apps <R.18>✓.

3.4 Comparison

Table 3.1 gives an overview over this chapter using the applicable requirements from section 2.6.

			OpenHAB / ESH	free@home	HomeKit
	target space type		home	home	home
	vendor independent		yes	no	(yes)
UI requirements					
R.1	**ui-addons**	UI is customizable via extensions	no, but open source thus forks possible	no	no, but custom apps on top of HomeKit possible
R.2	**floorplan**	UI renders devices on floor plan	no	yes	no
	R.3 zoomable	Floor plan is zoomable via gestures	n/a	yes	n/a
	R.4 multiple-floors	UI supports buildings with multiple floors	via workaround	yes	via workaround
	R.5 multiple-devices-same-XY	Floor plan allows multiple devices at the same XY coordinates	n/a	no	n/a
	R.6 icon-size-zoom-dependent	Icons become smaller/grouped in lower zoom levels	n/a	yes, become smaller	n/a
	R.7 room-geometry	Shape and placed of rooms modeled	no	yes	no
R.8	**orientation**	UI allows tablet usage in landscape and portrait mode.	yes	no	n/a
R.9	**login/authentification**	Login via username/password, certificate or other identity mechanisms should be possible	Password Login	Password Login	Apple ID, iCloud Keychain
R.10	**non-personal-devices**	One tablet can be used by multiple users with different permissions.	no	no	no
R.11	**admin**	UI allows system administration and configuration	yes	yes	yes
R.12	**different-views**	Admin and Usage scenario are split into different UI views	yes	yes	n/a
R.13	**debugging**	The web app should enable the user to understand why something does not work and support her at investigating why that happened.	no	no	n/a
R.14	**multi-home**	Multiple home or sites in one app or separate apps on the same device should be possible.	n/a	n/a	yes
Backend system requirements					
R.15	**permissions**	Is it possible to restrict users to only be able to control certain devices	no	no	yes, advanced
R.16	**autoconfiguration**	New devices are automatically discovered, without user input like devices address etc.	yes	yes	yes
R.17	**store/packages**	Systems allows installation of extensions via central repository	yes Extensions repo	no	yes App Store
R.18	**semantics**	Systems knows concepts of rooms, device types, etc.	not decided yet	yes, rooms + closed set of device types	yes (for non-custom types)
Constaints/Pseudo requirements					
R.19	**web-app**	UI is implemented as web app	yes (+ native iOS, Android apps)	yes (+ hybrid apps)	no
R.20	**current-generation-waf**	Web app uses current generation WAF (Angular, React, Ember)	yes (Angular)	no (Qooxdoo)	n/a

Table 3.1: Comparison of related work with applicable requirements from section 2.6.

Chapter 4

Design

In the previous chapters I introduced the appropriate knowledge and researched how other projects addressed this problem area. In this chapter, I develop a system architecture and design to resolve the challenges resulting from the requirements listed in section 2.6.

In the first section (4.1) I split the overall system into manageable components. The second section (Back-end, 4.2) is about the first half of these components doing work in background, the other section about the actual web app (Front-end, 4.3).

The individual requirements are referenced with <R.x>. If a requirement is fulfilled its referenced with <R.x>✓, if not it is crossed out: ~~<R.x>~~↯.

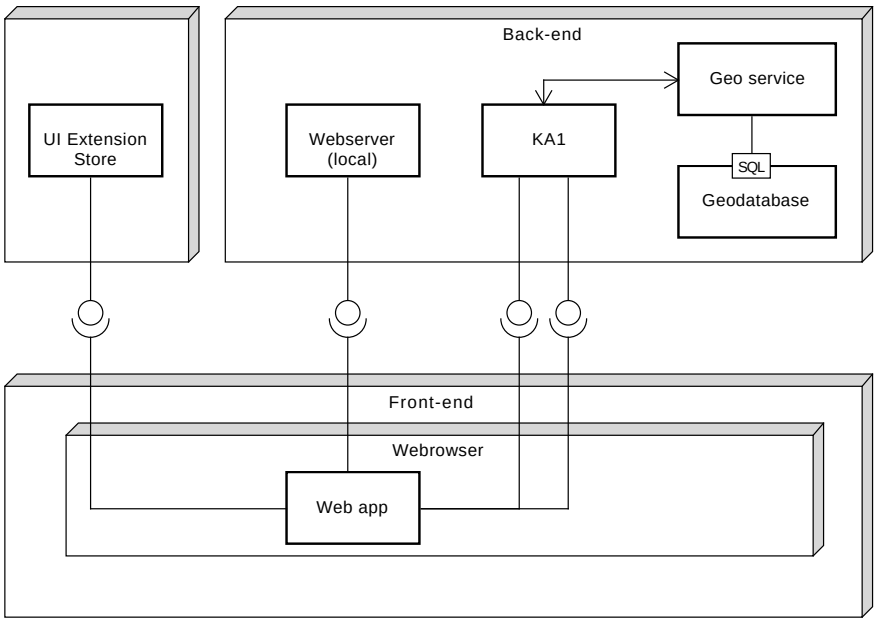


Figure 4.1: System structure diagram

4.1 System overview

The overall system is decomposed into the components shown in Figure 4.1 and in the following enumeration. The individual components are defined in more detail by the remaining sections of this chapter.

1. **Knowledge Agent** (KA)
2. **Geodatabase**: a PostgreSQL database with installed PostGIS extension.
3. **Geo service**: a VSL service bridging the gap between the geodatabase and the VSL.
4. **Web app**: Source code is hosted on a local web server and executed by a web browser on the user's tablet or PC. Communicates with the KA and Store.
5. **Extension store**: A catalogue and storage place of the third party front-end extensions. The development is not part of this work and does not exist yet. Therefore it's simulated by a (secondary) passive web server hosting files.

In practice the system set-up of this work's UI prototype is as follows: A tablet – connected to a local Wi-Fi network – communicates with a local computer running a VSL KA and VSL services. A web app on this tablet connects to the KA via a REST API and the WebSocket protocol. This protocols are set by the current release of the VSL KA. For web apps a system separation into back-end and front-end is typical. The web server serving the apps resources can also be embedded into the KA. The Extension store might be by proxied by the KA in future.

4.2 Back-end

As described in the previous section, the back-end consists of the KA (1), a geodatabase (2), the geo service (3) and an extension store (5). The geodatabase (see 2.4.2) is an off-the-shelf product, the KA (see 2.2.2) is provided by other members of the DS2OS project team. The full design of the extension store is not part of this work. The KA allows to set restrictions per node and therefore fulfils <R.15>✓. The VSL gateway concept includes auto discovery, but there is no implementation yet ~~<R.16>~~ϟ. The same applies for the S2Store and its package format ~~<R.17>~~ϟ. It has to be decided whether the UI extensions are distributed via the S2Store or a separate extension repository.

This leaves us with the last back-end system requirement: *"Systems knows concepts of rooms and device types"* <R.18>. One part of this requirement is provided by the existing VSL implementation: Through double usage of VSL types, device hierarchies can be built. The other part of requirement <R.18> are rooms. As explained in section 2.4.1 the VSL has no specification for this kind of data yet. The first part of this section is this specification. The other part is the API for a VSL service which allows to query and modify this geo information.

4.2.1 Back-end geo data model

To fulfil the goals of this work I require an advanced geo-data model. During section 2.4.1, I presented challenges for indoor building models and possible solutions.

Since the VSL is designed to work for spaces from a single flat to a office complex, I need to design a model that can take care of the special cases, without getting itself too complex. I tackle this problem using the power of 3D space in the back-end. Devices are not assigned to "office 01.05.023", but specifically to coordinate (52.4, 48.11, 1). The relation between a device (point) and a room (polygon or polyhedron) is defined by equations. The assignment of a device to a room solely results from its position within that room's geometry. This is a fundamental difference to the systems featured in chapter 3, and resolves the challenges attached to divisible rooms and portable devices.

To compute this contains-relation the floor plan is required as vector data. Floor plans in today's smart space UI are typically integrated as a bitmap image. When the space is present as vector data, one can reuse it for actual display. This eliminates costly maintenance of a bitmap floor plan and reduces data redundancy. However, for a good user experience more data besides room geometries such as doors, windows or furniture is needed.

4.2.1.1 Structure

How should one model the individual rooms, devices, furniture, floor, components, buildings in a database? What are the relevant attributes of each type? Let me list theses entities with their attributes:

- **Rooms** have a *name* and a *polygon* or *polyhedron* in 3D space
- **Devices** have a VSL *path* and a 3D *point* in form of single XYZ coordinate
- **Windows**, **doors** and **furniture** consist only of a *point*, *polygon* or *polyhedron* in 3D space. If you want to change or delete them, you require a way to address the individual elements e.g. via an *id*.
- **Floors**, building **parts** and **buildings** have a *name* and are described by a *polyhedron*. Compared to rooms, they can contain other rooms and might overlap with each other.

Let me discuss the individual attributes in more detail: VSL path and room number or name are quite similar. Relative to the site, they both address an entity and are unique. Points, polygons and polyhedrons are geometries and thereby represented via the geom type.

I further add a primary key, as unique identifiers are always recommended for objects in databases. In particular in distributed systems, random UUIDs should be used instead of consecutive ones.

I want the geodatabase to be as flexible as the VSL, therefore I use one single database table to represent all entities listed above, compare Figure 4.2. This leads to reduced complexity for the first API implementation, as I do not have to create own methods for each entity. In order to be able to separate the entities again in a different layer or reiteration, I add a string attribute named *type*.

This leads to the final attribute set for the combined entity "**Location**": uuid, name, type, geom

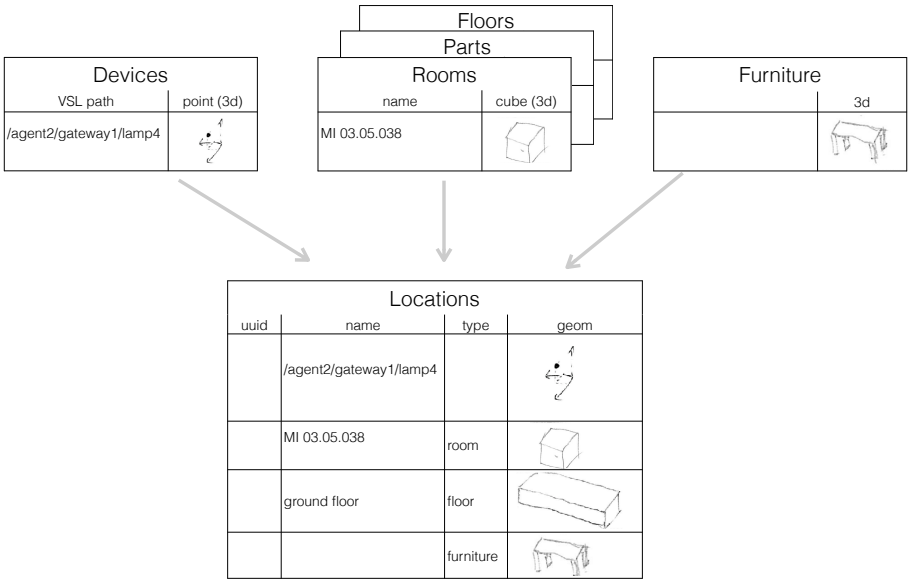


Figure 4.2: Combining all spatial entities into one

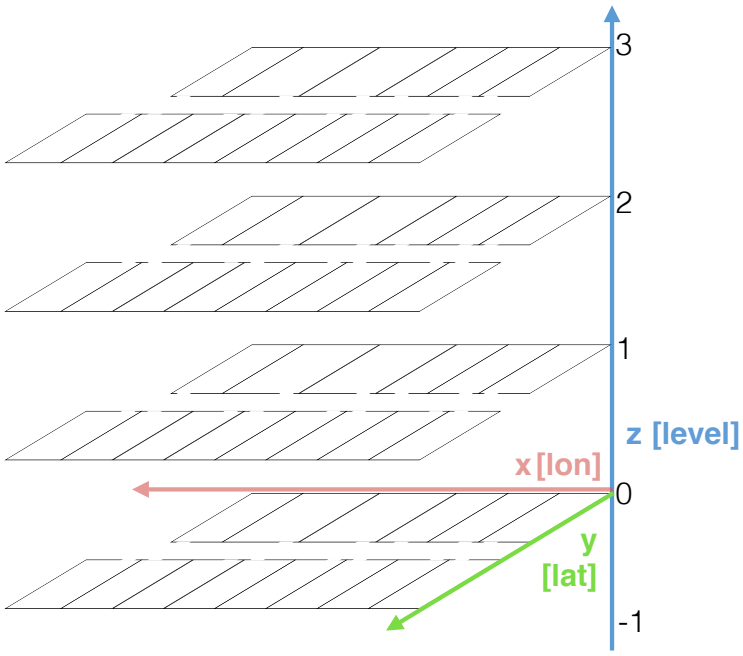


Figure 4.3: Four floor example building with chosen coordinate system

4.2.1.2 Coordinates and geometries

In the simplest case the geom attribute contains a point with one 3D coordinate, which is adequate for a lamp or an oven. In the case of a simple room it contains the base area as a 3D planar polygon parallel to the XY-plane. In other words: the Z component of each coordinate is the same. A standard rectangular room is therefore defined by at least four different 3D coordinates, e.g. `{"type":"Polygon", "coordinates":[[ [30,10,1], [40,40,1], [20,40,1], [10,20,1], [30,10,1] ]]}`. Notice that the start and end coordinate is the same. A floor, a building, or multi-storey room is defined by a polyhedron, e.g. a cube with eight different 3D coordinates, which results in 30 entries list as each surface of the cube is enumerated separately, see section 2.4.3.

In practice the end user does never see the actual figures representing a location, only icons rendered on-top of a floor plan. Thereby during selection of the coordinate system, I do not have to take the end user into account. WGS84 long lat has established itself as a quasi-standard in web maps. Therefore this project uses latitude (lat) as X, and longitude (lon) as Y coordinate (see Figure 4.3). The third dimension of the XYZ coordinates is based on the floor number. All coordinates with Z value between $[0; 1[$ are part of the ground floor, $[1; 2[$ the floor above (British English: first floor, American English: second floor) and a coordinate Z value between $[-1; 0[$ is in the first basement. An alternative is the height above mean sea level (MSL). But the actual MSL of the ground and the floor heights are normally unknown to users, whereby floor numbers relative easily to gather.

PostGIS (2.4.2) supports custom coordinate systems aka spatial reference systems[1] and is able to transform coordinates to different systems. Therefore even an own CRS per smart space would be possible without much overhead. But other parts of the design might break e.g. usage of GeoJSON which only allows WGS84 long lat as specified by [10, Ch. 4] – whereby earlier versions of the specification allowed other coordinate systems.

4.2.2 Geo service API

In the previous section I specified the geo data model, this section specifies a VSL service API which allows to query and modify this model. This *geo service* connects the geodatabase with the VSL KAs (see Figure 4.1) by transforming VSL requests like *get /search/positionOf/** (get all locations in the database) into a corresponding SQL query `SELECT * FROM locations`. To implement this functionality the VSL allows to create virtual nodes (see section 2.2.1). For the example above: *positionOf* is a virtual node and all read and write accesses are redirected to the appropriate callback method inside a Java class of the geo service package.

The geo service has two main tasks: implement CRUD operations for the geo entities (see previous section) and execute advanced queries e.g. search.

CRUD stands for create, read, update and delete [29, p. 381] which are the four basic functions of persistent storage. These four operations can be found under different names in SQL: insert, select, update, and delete. The VSL has only the data access operations get and set, so the question is how to map four CRUD operations to the these two, compare Table 4.1.

Operation	**VSL**	**SQL**	**HTTP methods**
Create	**?**	INSERT	**PUT** / POST
Read (Retrieve)	get	SELECT	**GET**
Update (Modify)	set	UPDATE	POST / **PUT** / PATCH
Delete (Destroy)	**?**	DELETE	DELETE

Table 4.1: Mapping VSL methods to SQL

Read is identical to a VSL get and update is identical to VSL set. To discuss the different implementation possibilities for the remaining operations, create and delete, I have to expand a little: The VSL concept only allows to dynamically create and delete VSL nodes if they are part of a list. Elements of a list node can be added with *get /KA/gateway/device/list/add/<name>* and removed with *get /KA/gateway/device/list/del/<name>*. In my point of view, using a VSL get operation to changes state – as it's currently done – is a design flaw. A delete or create operation changes state, a regular read operation (VSL get) does not. Therefore the delete operation should be performed by a VSL set or a new VSL delete operation, having to be introduced. As the individual API entities

[1] `http://postgis.net/docs/using_postgis_dbmanagement.html#spatial_ref_sys`

are implemented as virtual nodes, I can decide how to handle this issue. For create I can reuse VSL set, as I do not need to differentiate if a object is created or updated. For delete I can also reuse set – but this time with `del` as special keyword. An alternative is an own virtual node, e.g. named *deleteLocation*.

Let us turn to the specific API entities: In the previous section I merged all entity types into the Location entity. The client requests geometries of devices from the geo service, so I name the virtual node providing access to them "geometryOf". Geometries can be represented as GeoJSON or WKT (see 2.4.3). A *get /search/geometryOf/<foo>* returns the full geometry of the VSL path specified by parameter *foo* as GeoJSON. To reduce the implementation effort I decided to only return GeoJSON and not WKT. A *set /search/geometryOf/<foo> <value>* accepts GeoJSON, WKT and a BOX3D. WKT is more command line user friendly than GeoJSON, and a BOX3D is a lot shorter to write than a corresponding Polyhedron in WKT. All three formats can be handled with the same virtual node. The Polygon specified by GeoJSON snippet `{"type":"Polygon", "coordinates":[[ [30,10,1], [40,40,1], [20,40,1], [10,20,1], [30,10,1] ]]}` is `POLYGON Z((30 10 1, 40 40 1, 20 40 1, 10 20 1, 30 10 1))` in WKT.

The first implementation iteration with only *geometryOf* shows that this universal approach might be too complicated. Thereby I define a second virtual node *positionOf* which ensures all return values are a single 3D coordinate, regardless what kind of geometry is in the geom column. It is always the XY centroid with the lowest Z value of the stored geometry. In case of a point is the selfsame, in case of a c-shaped corridor in might be a point outside. Output format are three numbers separated by blank, e.g. `5.23 7.42 1`.

The resulting API for the CRUD operations is described in Table 4.2, where *foobar* can be a name, VSL path or UUID.

	geometry	**position**
Create Update	set /search/geometryOf/foobar `{"type":"Polygon", "coordinates":[[ [30,10,1], [40,40,1], [20,40,1], [10,20,1], [30,10,1] ]]}`	set /search/positonOf/foobar `5 2 1`
Read	get /search/geometryOf/foobar	get /search/positonOf/foobar
Delete	set /search/geometryOf/foobar `del`	set /search/geometryOf/foobar `del`

Table 4.2: Geo service API, part 1: CRUD

Having specified the basic CRUD operations of the geo service API, I now continue with the more advanced queries. The remainder of this section deals with all get requests which are not related to only one single object, for example "*return all devices in the living room*" or "*return devices near my current position*". With the setting described above, there are three different response types:

- **location**: only the name/path
- **position**: name/path and one coordinate aka centroid
- **geometry**: name/path with full geometry

As it is kind of complicated to design an API while thinking about all potential applications in an abstract way, let's try to represent them by three examples: a tablet app with floorplan (Web UI 2) in different development stages, a smartphone app, and other services. An overview of the resulting API is also given in Table 4.3.

4.2.2.1 API consumer example 1: Web UI 2

When the smart space has only one floor (**milestone 1**), Web UI 2 only needs to display the devices on an existing bitmap floor plan via *get /search/positionOf/**. When adding new devices the user should not have to enter VSL paths manually, therefore I require *get /search/devicesWithoutLocation* to fill an auto-completion or drop-down field.

In order to work with larger amounts of data (**milestone 2**: larger spaces, multiple floors, etc.) the Web UI 2 should be able to query only parts of the data, e.g. filtered per floor. Since floors are regular objects in the geodatabase, this can be achieved via a *get /search/positionsIn* query. For all positions from the first floor, on would perform a *get /search/positionsIn/1OG* request.

The floor selector UI element might require a list of all floors of a building: *get /search/locationsOfType/floor* or an extra virtual node *get /search/floors/**.

Web UI 2 **milestone 3**: The Floor plan is available as vector data from the geodatabase and is no longer served via bitmap tiles – filtered by floor. The corresponding request is *get /search/geometriesIn/1OG*, response is a GeoJSON document, which Web UI 2 can pass to the web maps library, in this case Leaflet.

4.2.2.2 API consumer example 2: Smartphone UI

The phone knows its whereabouts and would like to query which devices are in its immediate environment.

Via a *get /search/locationsNear/<lat>/<lon>* request, the phone gets a list of all location names near the specified coordinates. If the phone knows the current floor, it can get a filtered response via the optional level parameter, e.g. *get /search/locationsNear/<lat>/<lon>/<level>*. Future experiments have to define if the response should be limited by area or number of response entries.

Table 4.3: Complete overview of geo service API

Operation	VSL	SQL	geometry	position	without any geometry
Create	set	INSERT	set /geoservice/geometryOf/*<foobar>* *{"type":"Polygon","coordinates":[[[30,10],[40,40],[20,40],[10,20],[30,10]]]}*	set /geoservice/positonOf/*<foobar>* x y z	
Update (Modify)	set	UPDATE			
Read (Retrieve)	get	SELECT	get /geoservice/geometryOf/*<foobar>*	get /geoservice/positonOf/<foobar>	
Delete (Destroy)	set	DELETE	set /geoservice/geometryOf/*<foobar>* del	set /geoservice/postionOf/<foobar> del	
	VSL	**SQL**	**geometry**	**position**	**without any geometry**
everything	get	SELECT		get /geoservice/positonOf/*	
	get	SELECT			get /geoservice/devicesWithoutLocation
inside	get	SELECT	get /geoservice/geometriesIn/*<foobar>*	get /geoservice/positonsIn/*<foobar>*	get /geoservice/locationsIn/*<location>*
locationType	get	SELECT			get /geoservice/locationsOfType/*<type>*
locationType + inside	get	SELECT			get /geoservice/locationsOfTypeIn/*<type>*//*<location>*
coordinates	get	SELECT			get /geoservice/locationsNear/*<lat>*/*<lon>* get /geoservice/locationsNear/*<lat>*/*<lon>*/*<level>*
deviceType + inside	get	SELECT			get /geoservice/typeIn/*<type>*//*<location>*
reverse	get	SELECT			get /geoservice/locationsReverse/*<foobar>*

Milestone 1 | Milestone 2 | Milestone 3 | *<foobar>* is a VSL path, name or UUID

4.2.2.3 API consumer example 3: Other services

The last example is about other services which want to access geo information to do reasoning, e.g. interpret voice commands with context.

The virtual node named *locationsIn* allows hierarchical access. If your service wants to list all devices in the living room, it requests *get /search/locationsIn/livingRoom*. The response is a list of names of all entities within the location specified by parameter – in this example *livingRoom*.

Let's assume a voice recognition service interpreting "Turn off the lights in the living room". The key terms are lights and living room. Living room refers a room stored in the geodatabase, lights refers to the type */gahu/lamp*. The virtual node *devicesOfTypeIn* aka *typeIn* is designed to answer such questions: *get /search/devicesOfTypeIn//gahu/lamp//livingRoom*. The response is an intersection of *get /typesearch//gahu/lamp* with *get /search/locationsIn/livingRoom*.

The last virtual node created for usage by other services is *locationsReverse*. The request *get /search/locationsReverse//KA1/gateway1/lamp5* returns all locations in which the entity specified via parameter (in this case "/KA1/gateway1/lamp5") is part of, sorted ascending by area size, i.e. typically room, floor, building, site.

Some implementation details of this API are described in section 5.2.

4.3 Front-end

Having dealt with the back-end design in the previous section, this one is about the actual user interface. As described in the start of this chapter, the front-end consists of a single component named "web app" (see Figure 4.1) to be built from scratch. Its UI elements should be customizable via extensions <R.1> and use a floor plan <R.2>.

4.3.1 Front-end libraries

In theory the UI could be implemented as native, web, or hybrid app – see section 2.5.3. The task description for this thesis requires me to create a web app <R.19>. In contrast to a pure native implementation, a web app works on all relevant platforms without the need to create multiple implementations in different programming languages. This allows fast and continuous development cycles <R.21>✓. New developers can improve the prototype without an extensive training period as knowledge of HTML and JavaScript is widespread. Some frameworks can generate native apps and web apps from the same source code. If necessary in future, the web app can be bundled as hybrid app without much effort.

In the Analysis chapter, I introduced the Front-end libraries Angular, Ember and React (2.5.6.2). In mid of 2015, I decided to use Ember in this project, due to the active community and an expected sustainable further development. For example the APIs will not break so easily from release to release, and a relatively large collection of add-ons. With the Ember CLI utility a common project structure is endorsed, leading to better understandable source code projects. Therefore new developers, especially ones having experiences with Ember, can start contributing faster.

The web app's source code is a modern JavaScript dialect (ECMAScript 2015 aka ES6), HTML5 and SCSS <R.22>✓. Instead of ECMAScript 2015 one can also use Typescript (cf. React, Angular 2 – see subsubsection 2.5.6.2), however this is not the default for Ember apps, yet. SCSS (Sassy CSS) is a syntax variant of Sass 3, and is a superset of CSS3's syntax.

Regarding the web map libraries, I introduced OpenLayers 3, Leaflet, and Caesium in section 2.5.6.3. I select Leaflet for two reasons: New developers are more likely to cope with it <R.21>✓, there are ready to use plug-ins such as marker clustering, heat maps etc. It also supports marker detail popups out of the box. The other reason is ember-leaflet, an already existing add-on adapting between Ember and Leaflet objects. Therefore I do not have to write my own glue code. This choice leads to some restrictions: Leaflet only supports one tile grid at a time, where OpenLayers can handle different tile grids in one map. This means that the new prototype only displays the building floor plan as bitmap tiles and is no longer able to lay this floor plan on top of other maps or aerial photos. In the long term, however, the building plan should be rendered directly as vector data and no longer as bitmap tile sets. Therefore the tile layer would be free for other maps again and the problem would have resolved itself. In addition, a global map with detailed floor plan overlay is only reasonable when a user has access to several smart spaces like building services companies.

As Human interface guidelines (2.5.4) I choose the Material Design specification [22] from Google Inc. In mid 2015 there were three different approaches available on how to implement these guidelines with Ember. I choose an add-on named ember-paper, as it looked most promising and the authors of the other approaches started collaborating on ember-paper.

4.3.2 Front-end software architecture

The next three sections go into more details regarding the actual software architecture, the UI extensions and the graphical part of the user interface development process.

Through the choice of Ember and Ember Data, a large part of the web app's software architecture is already set. Ember implements a successor of the MVC pattern – so the classes can be divided into models, views and controllers. Even though the controllers are not called like that. The Ember Data classes are prefixed with DS.

Figure 4.4 gives an overview of the architecture. The initial version of Web UI 2 has two pages: A map and a list of the devices. Ember calls different 'pages' Routes. Based on

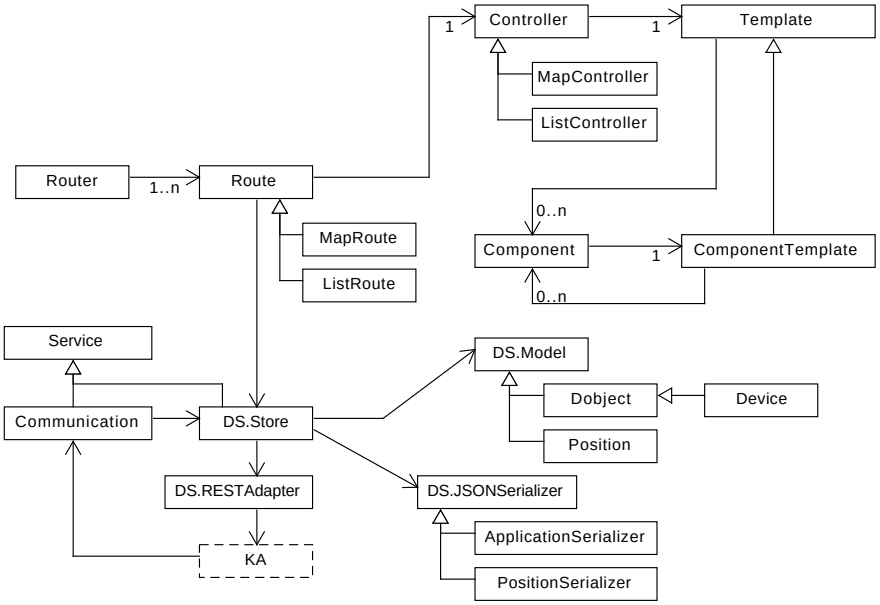


Figure 4.4: Software architecture of Web UI 2, heavy influenced by Ember

the URL, the Router class decides which Route is currently active. The corresponding Route class requests the Models from the DS.Store and passes them to the Controller. The DS.Store asks the KA via the Adapter for the data, if its not already in the DS.Store's cache. The Controller has a corresponding Template which defines the View. The Template can include one or multiple Components which also have a view definition in form of a ComponentTemplate, which again can include Components.

Ember Data has a fixed internal data model and comes with two API clients implementing communication with a server side persistence layer: RESTAdapter and JSONAPIAdapter. The Adapter connects the DS.Store with a back-end API by adapting the data operations into HTTP requests. It has the responsibility to build URLs and map operations to the corresponding HTTP methods. I opted for the RESTAdapter as it uses a HTTP PUT for update as required by the VSL API (2.2.3), whereas the newer JSONAPIAdapter uses HTTP PATCH. The conversion between back-end and front-end data format is done via Serializer classes: The ApplicationSerializer implements the common VSL concepts like handling child nodes, the PositionSerialiser further specialises by transforming the result of a position query to front-end models.

The asynchronous back channel is implemented in the Communication class. It connects to the KA via WebSocket and is notified when one of the subscribed VSL nodes was changed. When it receives a message from the KA, it looks if there is an Dobject for that node in the DS.Store. If necessary it tells the DS.Store to update the model class, which automatically results in view updates through Ember's binding concept. Communication and Store are both children of the Ember Service class and thereby singletons, every Ember class can reach them.

Model classes

For the this iteration of Web UI 2, the responses from the VSL API can be split in two categories: positions and VSL nodes. I use three model classes to store the data on client site: Dobject, Device and Position – compare Figure 4.5.

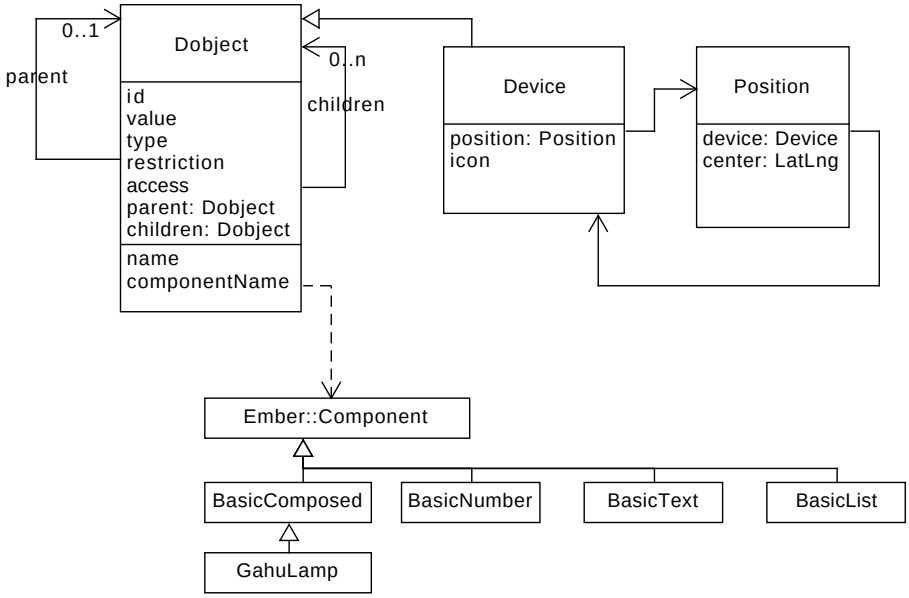


Figure 4.5: Web UI 2 model with type based rendering component classes

A Position has a reference to a Device and the eponymous position representing the centroid of the device as 3D coordinate. For the latter the LatLng class of the Leaflet library is used, which – beside one would guess from its name, can store altitude as third value.

The class Device is just a special case of a VSL node: It has a back reference to Position and an attribute called icon which returns a Leaflet Marker instance. The device's attributes are modelled via the relation "children", that the class inherits from its parent class, Dobject.

The front-end model for VSL nodes is the Dobject class. The name Dobject is a short form of "DS2OS base object". It has two recursive references: one 1:n relation to its children and a n:1 relation to its parent. This allows developers to traverse Dobjects in both directions in an easy way. The other attributes are id, value, type, access, and restriction. The id attribute is enforced by Ember Data and is the full VSL path. Value is the actual content of the node, besides children. The type attribute is a list of VSL types which is used to interpret the value in the UI. These types are defined in the VSL Model Repository (CMR, see subsection 2.2.1). Restrictions and access read-only are enforced by the UI components and the back-end - not here in the model layer. The method componentName returns the Ember Component class name which is best suited for rendering based on types. See subsection 5.3.4f for more information.

4.3.3 Front-end UI extensions

Subsubsection 2.2.4.1 described following UI extensibility scenario:

> The user has radio controllable LED bulbs in his living room. The software control element is build with three sliders, adjusting the red, green and blue value of the light (compare left part of Figure 2.10). When he wants to dim the light, all three sliders have to be adjusted, and he always needs some time to find the right colour. As he is dissatisfied with this situation, he opens the "Extension Store". There he finds an extension that has only one slider for the colour, a second slider for the intensity and a toggle switch to turn the light off (compare right part of Figure 2.10). He hits the install button and the initial control element is replaced with the new one from the extension. Now he can switch off or dim the lights without destroying the colour settings.

For this development iteration, a Web UI 2 Extension is a package consisting of one or multiple Ember Components. A Ember Component consists of a 'template' (HTMLBars) and a 'controller' (JavaScript) containing the marker icon specification and other logic/interactivity code (4.3.2).

Each of these Components maps a VSL node type like */basic/text* or */basic/number* to a corresponding HTML representation – e.g. for the previous examples `<input type="text"/>`, `<input type="number"/>` or – if minimum and maximum values are defined – `<input type="range"...>` etc.

To allow browsers to process the templates as effectively as possible, they have to be preprocessed/translated into an intermediate format [30]. Thus, installation of an extension from source currently requires either full ember development environment to be installed, e.g. via `npm install -g ember-cli` or a customized build process using the `ember-cli-htmlbars`[2] module. [31]

In future iterations following issues have to be addressed:

- When an UI extensions is installed, should it be installed for all users/certificates using this Web UI 2 deployment, or should users be able to choose which extensions they want to use and which not?
- When deciding to allow the latter, an new VSL services storing this user configurations is required
- A back-end service needs to notify running Web UI 2 instances, when new UI extensions are installed. Otherwise, users needs to trigger the web app's reload manually.

For a (central) UI Extension Store, decisions about the following issues have to be made:

- Package format: The most obvious is to base on ember-cli (in-repo) addons, npm or bower packages. Or we could also decide to create a own format.

[2] `https://github.com/ember-cli/ember-cli-htmlbars`

- Upload in source or binary: Should we force developers to upload the source code to the UI Store? In contrast to a pure 'binary' exchange platform, the extensions could be recompiled centrally if a future ember release changes the API or the template intermediate format. Maybe even force them to provide git repositories on github.com as it is typical for npm/bower packages.
- Distribute extensions in pre-compiled form to users? Or force users (or a service in the user's smart space) to compile the extensions themselves?

4.3.4 Graphical UI-Design

This section is about the development process of the graphical aspects of Web UI 2, the prototype resulting from this work. Literature recommends to involve graphic designers [2, p. 57], which are unfortunately not available in the DS2OS project group. Therefore, for this iteration, I have to make decisions without input from the graphic design field.

During chapter 2, Analysis, I elaborated scenarios from user interviews, which I used to create a use-case model. A use-case model consists of actors and use-cases which I revisit in this section together with the requirements.

The requirements (2.6) relevant for the GUI are:

<R.2> UI renders devices on floor plan
<R.3> Floor plan is zoomable via gestures
<R.4> UI supports buildings with multiple floors by providing selector to switch between different floor plans
<R.5> Floor plan allows multiple devices at the same XY coordinates
<R.6> Icons become smaller/grouped in lower zoom levels
<R.8> UI allows tablet usage in landscape and portrait mode
<R.11> UI allows system administration and configuration
<R.12> Admin and Usage scenario are split into different UI views
<R.13> Debugging: The app should enable the user to understand why something does not work and support her at investigating why that happened
<R.25> The interface should be self explaining, so that users can use it without having to read a manual
<R.27> If something is done automatically by the system, users want to be able to understand and investigate why that happened

The relevant actors for Web UI 2 are the Occupant and the Admin. Actors are a role an individual can take, e.g. an Admin can simultaneously be an Occupant. (Section 2.3) The use case model (Figure 2.11) provides the following top level use cases involving an Occupant:

<U.1> operate/control devices
<U.2> manage device location: add, update, delete device positions
<U.3> configure service
<U.4> install service

Only the first two use cases are part of this release of Web UI 2 (compare Figure 2.11).

The top level use cases of actor Admin are explicitly not part of Web UI 2:

<U.5> manage floor plan
<U.6> manage user, permissions

Nevertheless, <U.5> is part of this work, but without a GUI.

Let me combine this into a possible resolution:

In theory, everything can be on the floor plan. But when users want to install a new service into the space, they should not have to search for the computers running the DS2OS in the floor plan. So in any case I require some kind of menu to switch between different views or use cases.

The users can use this menu to switch between the floor plan view, list view (both <U.1>), the floor plan view in edit mode <U.2> <R.12>, service management <R.11> (configuration <U.3>, service installation <U.4>), and UI preferences (install and uninstall extensions <U.4>)

Due to the scope of work, the floor plan view is in primary focus: I am required to build the floor plan <R.2> with a zoom control <R.3> and floor/level selector <R.4>. On top of the building plan are icons representing the devices. When users click on an icon, they can inspect and change values <U.1> e.g. switching a light on. To fully implement <U.2> there must be a way to add new devices to the floor plan and move or remove existing device positions. I choose to implement this by adding a "Floating Action Button" in the right bottom corner, as known from Material Design (4.3.1).

The markers of the individual devices might even display its current state based on colour or different icons. E.g. Apple used a grey light bulb for an inactive lamp and a coloured for an active one in their HomeKit presentation in 2014, see Figure 2.26.

As the device representations should adjust itself depending on zoom level, I require a concept how device icons are handled, which come too close to each other, for example the icons get smaller or are merged <R.6>. The latter is relevant for multiple devices at the same XY coordinates <R.5>.

For the representation of the floor plan itself, I presented different existing approaches in Analysis subsection 2.5.2 – my conclusion was that I require an abstracted version with outlines of rooms, where gaps represent doors or maybe even windows. Optionally also add fixed furniture such as tables (2.5.2.4). In my point of view, a North-oriented floor plan is suitable for Admins, where Occupants cope better with a rectangular representation.

If users have access to multiple spaces <R.14> the likelihood of confusion should be reduced. This can be achieved by a own wallpaper (cf. Apple iOS 10 Home App, section 3.3) or colour schema per space. Support for tablet usage in landscape and portrait mode <R.8> might require different concepts based on the screen width.

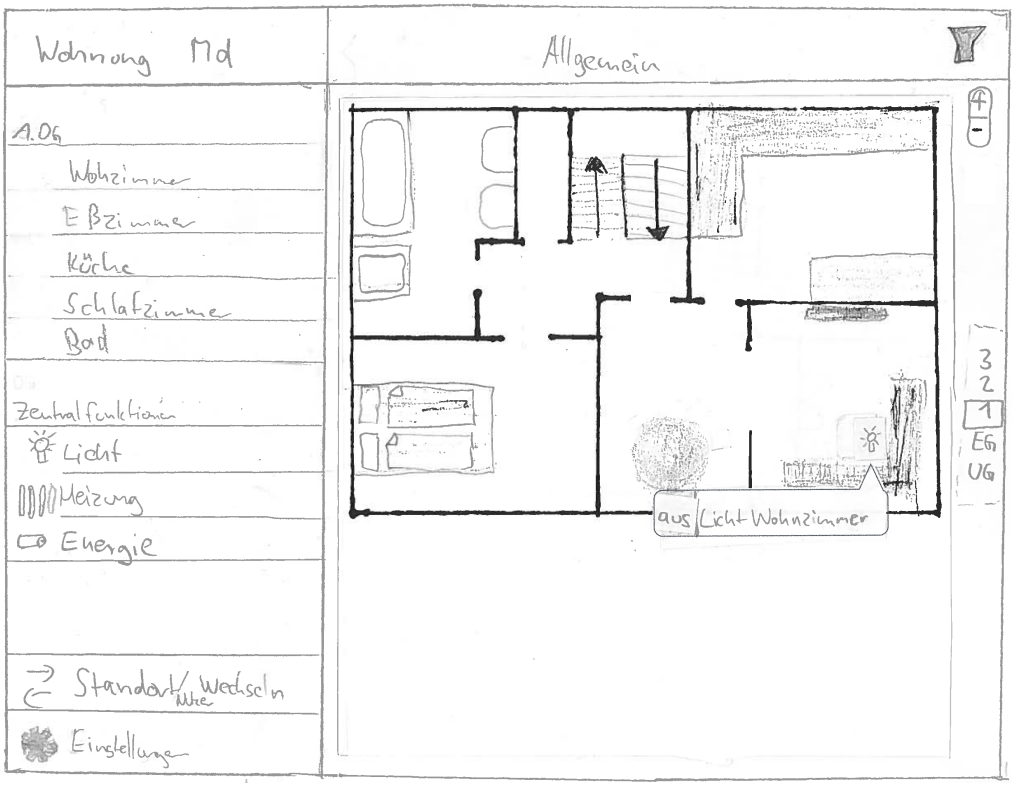


Figure 4.6: Paper prototype showing one floor of a multi-family home

4.3.4.1 Paper prototype

During the design phase, I implemented an intermediate version of the UI described above, as paper prototype. In usability engineering, paper prototypes are used to design, evaluate and improve aspects of a user interface even before a running system is available [2, sec. 3.4].

The prototype shown in Figure 4.6 consists of several parts: The left third is a vertical menu, the other two thirds are occupied by the floor plan. It is thought to mock an iPad in landscape orientation and is influenced by Apple iOS's settings and maps apps.

The building representation is as described above: walls are without diameter, doors are indirectly represented by gaps in the walls, and not sub-circles describing the opening direction. Also featured are large or fixed furniture such as seating, the TV in the living room, tables, bed, bathtub, shower, sink and stairs. To simulate zoom, the floor plan is available in different sizes – which were created via photocopier.

There is only one device, a lamp in the living room. The figures depicts the situation as the user has just touched on the maker and thereby opened a popup. The popup contains a button for switching off the lamp and its name. The lamp icon is available in two versions: One switched on (with more rays); and one switched off (without rays).

UI elements around the floor plan: Zoom buttons at the top right. Underneath: vertical

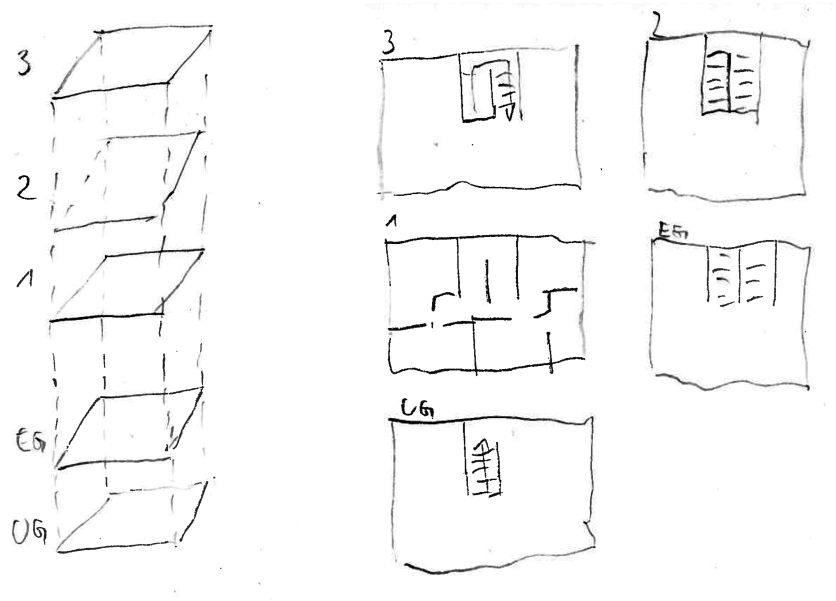
list of floors acting as floor selector – the first floor is selected.

What you still see in this design iteration is the side menu - in the final design, this was remove due to feedback. The side menu lists the individual rooms per floor. Selecting a room in the menu would centre and enlarge the selfsame room in the floor plan view.

The side menu also contains master functions for light, heating and energy. The idea was to provide a complete overview: For lights a dark overall plan (cf. Figure 4.7), wherein the switched-on lamps illuminate the rooms, cf. lightmap[3]; for heating a heatmap actually displaying temperature; and energy with consumption history graphs or the current energy flow. Figure 4.7 shows two experiments regarding parallel display of all floors: Sub-figure A is a perspective view, where renders floors side by side. This feature was not followed up in this work.

In the upper right corner is an option to filter by device groups (lighting, HVAC, entertainment, security etc.), which should improve getting an overview when there are more than 50 devices in the building plan view.

The final UI is shown in section 5.3, specifically in Figure 5.1ff.



(a) 2.5D general overview (b) side by side

Figure 4.7: Different variants for parallel floor plan display

[3] http://lightmap.uni-hd.de/

Chapter 5

Implementation

In the previous chapter I presented a design for a floor plan based smart space user interface and its associated back-end components.

This chapter presents relevant details from the implementation phase: Section 5.1 describes the database setup process including the schema; section 5.2 gives some details on the implementation of the geo service itself e.g. the mapping between API calls and corresponding SQL queries – highlighting interesting ones; and finally section 5.3 describes the implementation process of the web app, including screenshots and an interaction diagram.

5.1 Back-end: Geodatabase

The geodatabase is a PostgreSQL database with the PostGIS extension. I choose this setup as I had some experiences with it from previous projects and many algorithms are thereby available already in the database layer. This choice might arise some problem if the whole server setup might be transferred to low budget home server like an Raspberry Pi, but this was no requirement for this thesis. If such situation will arise in future and problems occur, one might switch to SQLite with SpatiaLite. This change might require changes on SQL queries.

As explained in subsection 4.2.1, the geodatabase layout consists of one table with the columns uuid, name, type and geometry. This design allows flexible models without the need to change database layout regularly. The relations between the individual entries emerges indirectly from the geometry field. The following SQL statements produce an instance of this database:

```
CREATE EXTENSION "postgis";
CREATE EXTENSION "uuid-ossp";
CREATE TABLE locations (
  "uuid" uuid NOT NULL DEFAULT uuid_generate_v4() PRIMARY KEY,
  "name" text UNIQUE,
  "type" text,
  "geom" geometry(GeometryZ, 4326) NOT NULL
);
CREATE INDEX locations_geom_idx ON locations USING GIST (geom);
```

Thus the **geometry** field can be used with advanced PostGIS functions, an indication on the CRS is required. As discussed in section 4.2.1, I use WGS84 long lat in this project, which the database knows as SRID 4326. The database requires the CRS to be set for all geometries.

To add full 3D functionality to PostGIS (c.f. 2.4.3, in particular Listing 8), a additional PostgreSQL extension has to be loaded:

```
CREATE EXTENSION "postgis_sfcgal";
```

Within the Ubuntu distribution, this extension was included starting with release 16.04. Unfortunately, as of 2016 the PC in the AHN lab (aka *bling* – see 6.2.3 and 2.2.0) runs an older Ubuntu release. The extension is also not included within the PostgreSQL distribution[1] I am using on my development computer. Compiling the extension oneself is not simple task. For these reasons, I decide to only use PostGIS's own functions for this iteration.

[1] http://postgresapp.com

5.2 Back-end: Geo service

The geo service connects the geodatabase with the DS2OS VSL. Basically it transforms VSL API requests like *get /search/positionOf/** (get all locations in the database) into a corresponding SQL query `SELECT * FROM locations`. For further details have a look at the corresponding section 4.2.2 of the Design chapter. Table 5.1 gives an overview about the implemented methods. Future work on this component is outlined in section 7.3.

Some notes on special cases:

get /search/**positionOf**/<name>
: **get**: return the centroid of the entity specified by parameter <name>.

```
SELECT uuid, name, type,
       concat(ST_X(ST_Centroid(geom)), ' ',
              ST_Y(ST_Centroid(geom)), ' ',
              ST_ZMin(geom))
FROM locations WHERE name = ?
```

possible extension: when value is only a lat/lon coordinate: do not overwrite more detailed geometry in database, instead merge: For 3d points: get z and only overwrite x and y. For more complex geometries:

- get centroid of old geom,
- calculate a move vector between this centroid and the new location
- move existing geom by this vector

get /search/**geometriesIn**/<name>
: Returns geometries of all objects which are spatially inside of the object specified by parameter <name> as GeoJSON.
To simplify room data management the design allows planar areas for simple rooms (see section 4.2.1). Therefore a special handling has to be implemented for this case: Instead of a classic intersection in 3D space we have to filter for all elements between $Z \leq z < Z + 1$, where Z is the level of object <name> and z the level of the individual objects to be filtered. The result set is than intersected in 2D and we get the final result. The actual runtime order of the filter steps in the SQL query is not important as the query optimizer of the database takes care about the optimal order. To get the level of the geometry I use `ST_ZMin()`.

```
SELECT d.uuid, d.name, d.type, ST_AsGeoJSON(d.geom)
FROM locations d, locations area
WHERE ST_ZMin(area.geom) <= ST_ZMin(d.geom)
  AND ( ST_ZMin(d.geom) < ST_ZMin(area.geom)+1
     OR ST_ZMin(d.geom) < ST_ZMax(area.geom)  )
  AND ST_Contains(ST_Force_2D(area.geom), ST_Force_2D(d.geom))
  AND area.name = ?
```

Method	VSL method	Description	SQL query key
used by Web UI 2.0			
positionOf/*	get	returns **centroid** for every object in the geodatabase	getAllCentroids getAllDeviceCentroids
positionOf/*<name>*	get	returns **centroid** for object specifiyed by parameter <name>, same format as above	getCentroidByName
	set	**insert or updates geometry** of object specified by <name> in(to) the geodatabase – similar to set geometryOf, but in simpler format. possible extension: when value is only a lat/lon coordinate: don't overwrite more detailed geometry in database, instead merge.	insert, updateGeom
devicesWithoutLocation	get	returns VSL **paths** of all nodes directly below a Gateway (**devices**), which **do not have a geometry** assigned yet	getDeviceLocations
implemented + integration test			
geometryOf/<name>	get	returns **geometry** of the object specified by <name> as GeoJSON	getGeomByName
	set	**insert or updates geometry** of object specified by <name> in(to) the geodatabase. Recognises if the input is GeoJSON, WKT or BOX3D	insert, insertWKT, insertBOX3D updateGeom, updateGeomWKT updateGeomBOX3D
geometryOf/*<name>* del postitionOf/*<name>* del	set	**deletes** object specified by <name> from geodatabase.	delete
implemented, currently unused and without integration test			
geometriesIn/*<name>*	get	returns **geometries** of all objects which are **spatially inside of** object specified by parameter *<name>* as **GeoJSON**.	getGeomsInName
positionsIn/*<name>*	get	returns **centroid** of all objects which are **spatially inside of** object specified by parameter *<name>*. Same implementation as above, but returns centroids and not geometries.	getCentroidsInName
locationsIn/*<name>*	get	returns **names/paths** of all objects which are **spatially inside of** object specified by parameter *<name>*. Same implementation as above, but without any geometry.	getLocationsInName
locationsReverse/*<name>*	get	returns list of **locations** (names of area/room/building) in which object *<name>* is **part of**. sorted by geometry size. (First area, room, then building, site etc)	getLocationsReverse
typeIn/*<type>*//*<location>* deviceOfTypeIn/*<type>*//*<location>*	get	returns VSL **paths** of nodes with type specified by <type> which are inside of the geometry of *<location>*. **Intersection of typeSearch** with data from geodatabase.	getDeviceLocationsIn Name
locationsOfType/*<type>*	get	returns list of **locations** (names of area/room/building) where the content of **type column** equals parameter *<type>*	getLocationsOfType
locationsOfTypeIn/*<type>*//*<location>*	get	returns list of **locations** (names of area/room/building) where the content of **type column** equals parameter *<type>* and which are **inside of** the geometry of *<location>* – e.g. for floor list of a building.	getLocationsOfTypeIn Name
locationsNear/*<lat>*/*<lon>* locationsNear/*<lat>*/*<lon>*/*<level>*	get	returns list of **locations nearest** to the specified *<lat>*/*<lon>* **coordinate**, optionally filtered by *<level>*.	getLocationsNear
partially implemented but dropped from specification			
~~geometryOf/*~~ ~~geometryTextOf/*~~	~~get~~	~~returns geometry for every object in the geodatabase as GeoJSON~~	~~getAllGeoms~~ ~~getAllGeomsWKT~~
~~geometryTextOf/*<name>*~~	~~get~~	~~returns geometry of the object specified by <name> as WKT~~	
~~geometriesTextIn/*<name>*~~	~~get~~	~~returns geometries of all objects which are spatially in object specified by parameter <name> as **WKT**.~~	
~~geometryReverse/*<name>*~~	~~get~~	~~returns list of geometries (area/room/building) in which an object <name> is part of.~~	~~getGeomsReverse~~

Table 5.1: Geo service implementation status overview

5.3 Front-end: Web app

The web app is implemented with Ember 2.6, the substantial addons are *ember-data*, *ember-paper 0.2*, *ember-leaflet* and *ember-websockets*. The *ember-paper* addon provides an implementation of Google's Material Design Specification (2.5.4).

5.3.1 Floor plan view

Figures 5.1 and 5.3 are screenshots of the map view: Depicted are five devices in the TUM Autonomic Home Networking (AHN) lab as described in section 2.2: four lamps and one blind. Each device is represented by a marker.

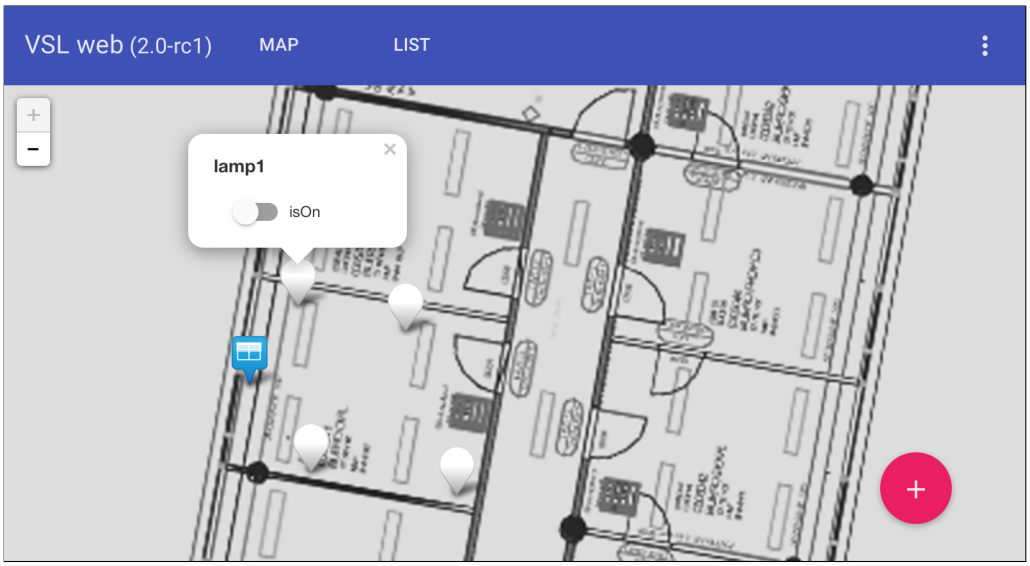


Figure 5.1: Web UI 2 RC1 map view with open lamp popup

When the user touches a marker, a Leaflet bubble popup listing the device's attributes is opened. In contrast to free@home (3.2), the app can not directly trigger an action, as the VSL data model allows multiple attributes per device.

Figure 5.1 shows the popup of a lamp: There is only a single switch as this lamp type has only one attribute of type boolean, named *isOn*.

Figure 5.3 shows the popup of a blind: This type consists of two attributes, both of type percentage: a *closed* value of 50 means that one half of the window is shaded, 100 the window is completely shaded. The second value indicates the angle of the blind panels. The percentage type is defined as number (aka integer) in the range from 0 to 100, which the UI represents by a slider.

To add a location to new devices, there is a floating action button (FAB) with plus icon at the bottom right.

The markers have a shape, colour and an optional icon. The lamps are represented with a white rounded marker, blinds with a blue rectangular marker with a blind icon inside. Icons for different blinds opening states are provided by *knx-uf-iconset*, but not used yet. From my point of view, markers without a tip would be better for this use case, but unfortunately the Leaflet *extra-markers* plugin only provides shapes with pointed bottom. However, the user study (6.3) showed that the markers are very well recognized because of the pointed bottom.

5.3.2 List / grid view

For smartphones and other small screens a list view might be more useful that a map view – especially in portrait orientation. In a later iteration the list view was replaced with a tile grid, to use the screen area more efficiently. Figure 5.2 shows a prototype reusing the same UI components as the map view's popup. Future iterations of this grid view could include tiles spanning over 2 columns or rows.

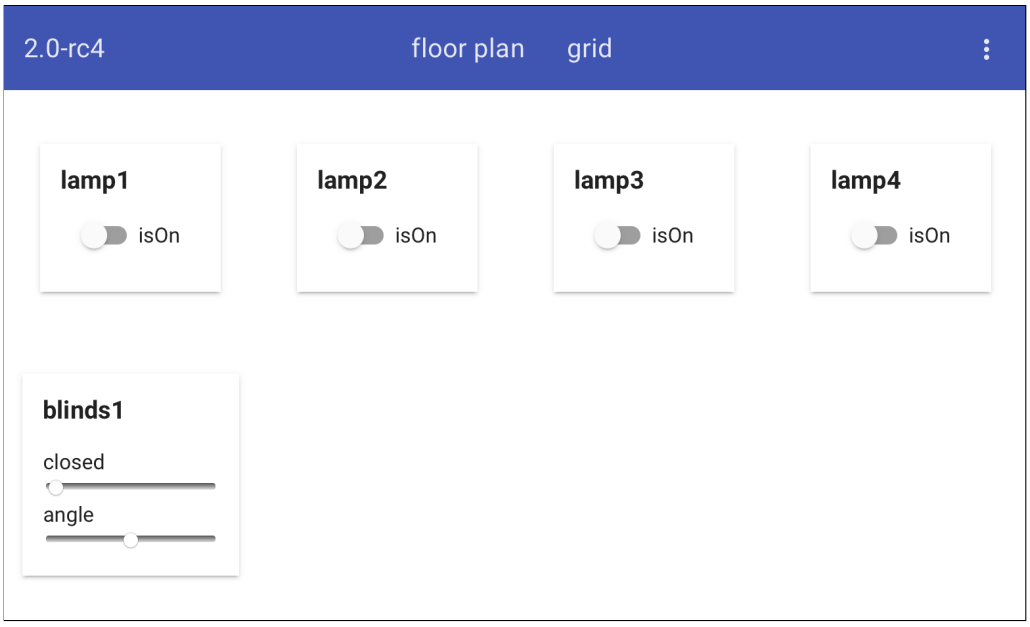


Figure 5.2: Web UI 2 RC4 grid view prototype

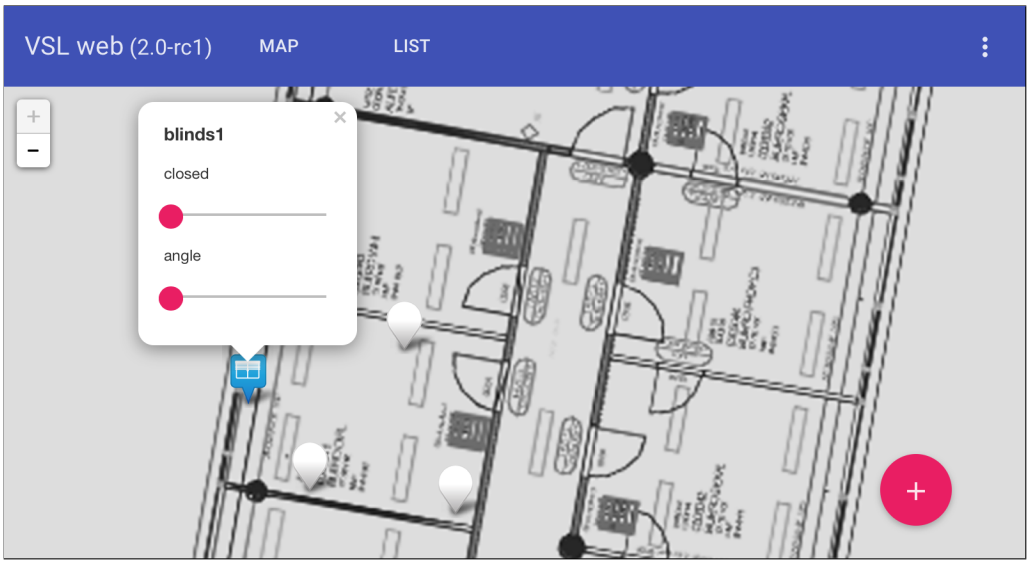


Figure 5.3: Web UI 2 RC1 map view with open blind popup

5.3.3 Web app – KA interaction

The software architecture has been described in the previous chapter (4.3.2), which I largely implemented one-by-one.

The sequence diagram in Figure 5.4 shows the interaction between front-end and back-end, starting at the time the user opens the app in her web browser.

The app starts with floor plan view which requires the device positions. To do this, the Web UI performs a HTTP GET request */search/locationsOf/** to the KA, the KA asks the geo service for these information. The geo service responds with a list of tuples consisting of a VSL path and coordinates. This response is forwarded back to the client (Web UI 2) by the KA.

Thus we know all devices/their address in the VSL as well as their position on the floor plan. The floor plan view shows first markers, but can not yet display what type the respective device has. For this purpose further GET requests with the respective VSL path have to be sent to the KA. The KA responds with a JSON document containing the meta information as well as the names of the children, which are the available attributes.

If the user now touches on one of these markers, a popup opens, which implicitly triggers further GET requests to the KA: For each child the same query is repeated as above. We get type, value as well as children and can decide which UI element we need to render in which state.

By changing the state of an UI element, e.g. a switch, the user triggers a PUT request to the KA. In the example shown in Figure 5.4, `{value:"1"}` is sent to */KA/gateway1/lampX/isOn/desired*. This triggers a callback function in all services that have subscribed to this *desired* node. In our case, this is the gateway service of the lamp. As

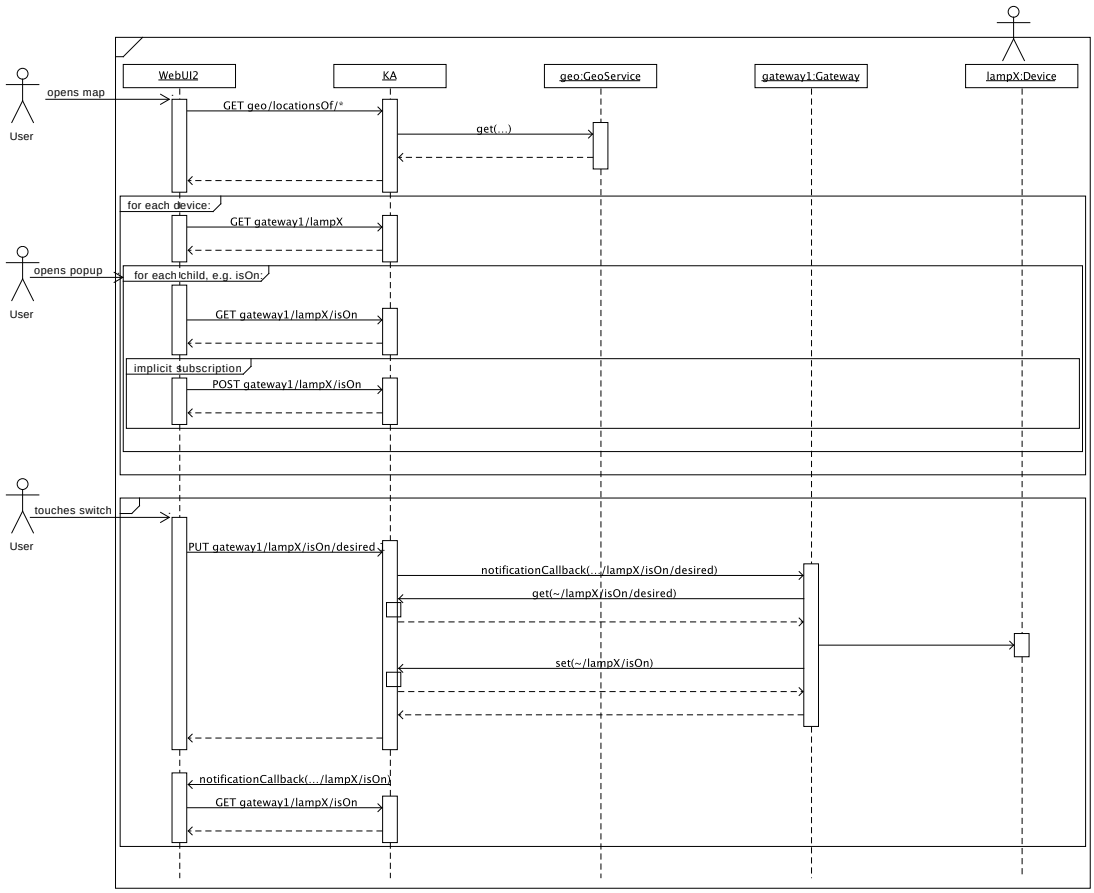


Figure 5.4: Sequence chart of web app ↔ KA → Device interaction

the callback only contains the address of the changed node, the gateway has to query the concrete value from the KA per *get* request and sends the corresponding command to the lamp. In this case the lamp is not a smart device, so we do not know if this was successful. However, we think positively and confirm that the command was sent by writing the new value to */KA/gateway1/lampX/isOn*. As all instances of the Web UI subscribed this node, they are notified of the change via WebSocket. Again, this message contains only the information that something was changed, but not the new value. Therefore, the Web UI's send HTTP GET requests to the KA, get the new value and the switches in the other Web UI instances flip to the new state.

In future iterations the Web UI might display this feedback even in that instance in that the user initially triggered the change, e.g. by holding the movable part in a intermediate state, till the confirmation arrives (c.f. iOS switch, which is oval till change was successfully performed).

5.3.4 Front-end extensions

Section 2.2.4 describes the extensibility of the Web UI desired by the project manager, in the form of a scenario. The UI element selection algorithm has been described there, too. In the current development iteration, a Web UI 2 extension consists of one or more Ember Components. A Component always consists of a 'template' (HTMLBars) and a 'controller' (JavaScript) containing the marker icon specification and other logic/interactivity code (4.3.2).

Each of these Components maps a VSL node type like */basic/text* or */basic/number* to its corresponding HTML representation – e.g. for the previous examples `<input type="text"/>`, `<input type="number"/>` or – if minimum and maximum values are defined – `<input type="range".../>` etc. Using `{{foo-bar p=bla}}` in a template, the Component FooBar with parameter `p` set to the value of `bla` will be included.

The algorithm from 2.2.4 was implemented using a 'component helper', cf. `{{component p.device.componentName c=p.device}}` in line 8 of Listing 10. The component helper embeds the template specified by the first parameter. In this case 'componentName' is a method of the Dobject class returning the name of the responsible component as string, see Figure 5.5.

Let's revisit the AHN lab as described in (sub)sections 5.3.1, 2.2, and Figures 2.7, 5.1; and substantiate the scenario from 2.2.4.1: In Listing 10a, line 4 we iterate over all positions and create a marker for each one (line 5). We attach a popup to this marker (line 6–9) and embed the corresponding component (line 8). As described above and shown in Figure 5.5, 'componentName' is a method of the Dobject. Let us assume that the current device is `lamp1`, which is of type */gahu/lamp,* which is traced back to */basic/composed* (compare Listing 2, line 2). Therefore we embed the component shown in Listing 10b which simply implements the rule for composed – to iterate over all children. `lamp1` has only one child node: `isOn`, which is of type */derived/boolean*. So in line 14, we embed Listing 10c which contains the switch as UI element. The other case in the AHN lab are the blinds: `blinds1` is of type */gahu/blind*, which also traces back to */basic/composed*, so the UI again iterates over all children (line 13). This time these are `closed` and `angle`, both of type */derived/percent*. As the UI does not know this type, it falls back to the next type */basic/number* (compare Listing 3, line 5/10) and we are at Listing 10d. The rule for number has a case distinction (line 18): As the restrictions *minimumValue* and *maximumValue* exist, the corresponding UI element is a slider (line 19).

To allow browsers to process the templates effectively as possible, they are preprocessed/translated into an intermediate format. Thus, even installation of an extension currently requires the full ember development environment to be installed, e.g. via `npm install -g ember-cli`.

In the future, the DS2OS project might provide ready to use, 'compiled' extensions for end users. A possible architecture is a central UI Extension Store, in which developers upload the source code of their new components. In contrast to a pure binary exchange platform, the extensions can be recompiled centrally if a future ember release changes the API or the intermediate format. (4.3.3)

a) templates/map.hbs (Fig. 5.1, 5.3)

```
{{#leaflet-map center=center zoom=zoom as |layers|}}
  {{layers.tile url="/static/fmi/{z}/{y}_{x}.png" minZoom="3" maxZoom="7" }}

  {{#each positions as |p|}}
    {{#layers.marker lat=p.center.lat lng=p.center.lng icon=p.device.icon as
    ↪  |marker|}}
      {{#marker.popup}}
        <h3 title="{{p.device.type}}">{{p.device.name}}</h3>
        {{component p.device.componentName c=p.device}}
      {{/marker.popup}}
    {{/layers.marker}}
  {{/each}}
{{/leaflet-map}}
```

b) templates/components/basic-composed.hbs (Fig. 5.1, 5.2, 5.3)

```
{{#each c.children as |child|}}
  {{component child.componentName c=child}}
{{/each}}
```

c) templates/components/derived-boolean.hbs (Fig. 5.1, 5.2)

```
{{#paper-switch checked=c.value }} {{c.name}} {{/paper-switch}}
```

d) templates/components/basic-number.hbs (Fig. 5.3, 5.2)

```
{{c.name}} <br/>
{{#if c.restrictions}}
  {{input type="range" min=c.restrictions.minimumValue
  ↪  max=c.restrictions.maximumValue value=(mut c.value) disabled=c.isReadonly }}
{{else}}
  {{input type="number" value=(mut c.value) disabled=c.isReadonly}}
{{/if}}
```

Listing 10: Various excerpts from HTML-bars template files

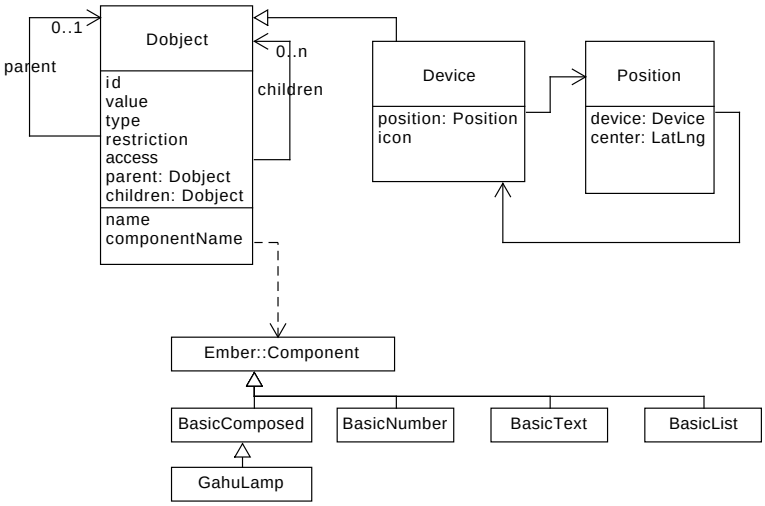


Figure 5.5: Web UI 2 model with type based rendering component classes – copy of Figure 4.5

Chapter 6

Evaluation

This chapter evaluates if the design and implementation of the web app and the geo service meets the goals of this thesis. I combine evaluation methods from software engineering (SE) and usability engineering.

The first section is classic SE and revisits the requirements concluding the analysis chapter and checks analytical if they are fulfilled or not. Depending on requirement, different procedures are necessary: Some functional requirements and constraints can simply be checked of, other requirements – especially ones related to quality – require tests with actual users which is postponed to section 3.

In section 6.2 miscellaneous tests and measurements are performed: The first test checks if the web app works with the three major desktop browsers – which is only successful after several attempts, due to non-uniform implementation. Test 2 uncovers a bottleneck in the back-end. Test 3 bridges desktop to mobile browsers: I examine the effects of the network connection by comparing cable and Wi-Fi. Test 4 is like test 1, only this time with mobile browser variants on two tablets: It turns out that we are set to a particular browser per operating system because of the client certificates. Finally, Test 5 compares the performance of the app between the tablet operating systems iOS and Android.

Section 6.3 uses user oriented procedures by inviting people to test the app into the lab, to analyse how they manage usage.

6.1 Requirements evaluation

During the Analysis chapter, I compiled requirements (summarized in section 2.6) which I revisit in this section: If a requirement is fulfilled it is referenced with <R.x>✓. If a requirement is not met, it is crossed out: ~~<R.x>~~↯.

Let's start with the functional requirements for the **front-end**: The UI is customizable via extensions <R.1>✓, and renders devices on a floor plan <R.2>✓. Trough Leaflet, the floor plan is zoomable via gestures <R.3>✓, but the current implementation does not support buildings with multiple floors ~~<R.4>~~↯. Multiple devices at the same XY

coordinates and grouped icons in lower zoom levels might be possible with Leaflet's clustering plugin, which has not been tested yet ~~<R.5>~~↯. Leaflet has no build-in support for zoom depended icons ~~<R.6>~~↯. The shape and place of rooms is modelled only in the back-end, but not yet at the front-end ~~<R.7>~~↯, as the floor plan is embedded as bitmap tile set and not as vector data. On a tablet, the web app is usable in both landscape and portrait orientation <R.8>✓.

User authentication is done via client certificates <R.9>✓. The practical usability of client certificates in today's web browsers on mobile and desktop PCs has to be tested as part of a user study with web browsers on iOS, Android and on desktop PCs (Google Chrome, Firefox, Safari). <R.10>, part 2: *"One tablet can be used by multiple users with different permissions"* currently only works if the OS or browser have multi-user support, as the corresponding KA feature to temporary upgrade permissions of a specific client certificate is not implemented yet <R.10>. The current UI implementation is the foundation for a modern web UI, concrete use cases like system administration and configuration have to be build on top ~~<R.11>~~↯~~<R.12>~~↯. In order to enable the user to understand what the system does in the background <R.13>, a full SHE implementation with an API for KA and service logs, etc. is necessary. Such a SHE is currently not available, therefore the web app can not give much insight ~~<R.13>~~↯. I assume there will be at least one deployment of the web app per DS2OS site. Remote access is gained via VPN. Therefore access to multiple sites from the same device is be possible <R.14>✓. I recommend to set a different primary colour per site.

Functional requirements of the **back-end**: The VSL allows to restrict users, so they can only control some devices <R.15>✓. VSL gateway services with auto discovery of new devices have not yet been implemented ~~<R.16>~~↯. The infrastructure for a central extension repository and service inter-exchange between DS2OS sites does not exist yet ~~<R.17>~~↯. But the VSL is aware of device types and with the geo service now also of rooms <R.18>✓.

Let's check the **constraints**: The UI is implemented as web app <R.19>✓ and uses Ember, which is a current generation web application framework <R.20>✓. Implementation as web app allows fast development cycles <R.21>✓. Ember allows developers to use newest HTML5/JavaScript APIs without having to take care of older browsers <R.22>✓, as its build chain transpiles the source code to a JavaScript dialect older browsers can also cope with. Ember uses npm and bower to manage libraries and their dependencies <R.23>✓.

The **quality requirements** can not be checked of as easily as the other requirements. Some require measurements <R.26>, other studies/tests with users performing the Occupant <R.25> or Developer <R.28> <R.29> role. But <R.27>*"If something is done automatically by the system, users want to be able to understand and investigate why that happened."* can be dismissed already: The necessary back-end services, e.g. a full SHE implementation, are not yet available. Therefore ~~<R.27>~~↯ is not realisable within this work.

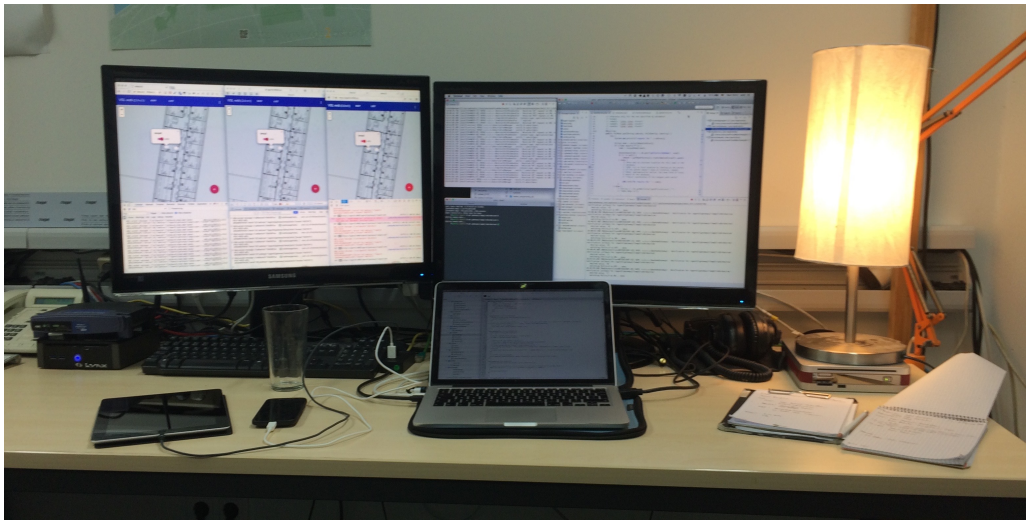


Figure 6.1: Test setup at AHN lab

6.2 Tests and measurements

This section is about tests I can perform myself without calling in users. Specifically, the following questions are to be investigated: Does the app work in all relevant browsers, both on the desktop and on tablets? Are there bottlenecks in the overall system? How does connection via wire in contrast to Wi-Fi affect the performance? Is the app's performance different when run on iOS then when run on Android?

In order to carry out these tests, some preliminary work is necessary:

6.2.0.1 What to measure?

First: What to measure exactly? The initial loading time of the app? The reaction time from touch of the switch, over lamp is actually on, to the KA callback arriving in the app? Let me illustrate this start-up process: First, the app itself is loaded (HTML, CSS, JavaScript), then the bitmap tiles and the marker images, then the metadata of the devices (the markers change to their device-specific shape and colour). When clicking on a marker, more specific data about the device (attributes, etc.) will be loaded and the popup rendered (see 5.3.3). There are many different events one can trigger at, some even repeat several times. The nature of the used framework leads to concurrency which makes measurement more difficult. These concurrent activities might be disabled to increase reproducibility as done in the benchmark mentioned below ("we're forcing frameworks to do more work synchronously than needed [...] to ensure run time can be measured."[1]), but I do not think it is a constructive solution for this case.

[1] `http://browserbench.org/Speedometer/` → about

6.2.0.2 How to measure?

Another question is: How to measure performance of a web app on the target platforms (desktop PCs and tablets)? I came up of three ways: Through *web browser debugging tools* and via additional JavaScript code – either *within the existing app itself*, or by embedding the app via iframe as seen in JavaScript *benchmarks*.

Desktop browsers include '**developer tools**', e.g. Google Chrome's Timeline view shown in Figure 6.2. It displays not only the pure processing time, but splits it up into Loading, Scripting, Rendering, Painting, Other, and Idle. This allows developers to analyse where bottlenecks are, which requests are made to the back-end, how long the execution of JavaScript codes takes, and which resources take a lot of time to load or render. Apple Safari and Mozilla Firefox have similar features. Some desktop browsers can connect to their mobile variant via a USB cable and thus debug a web app running on tablet or phone (Safari on iOS, Chrome on Android). How far values from Safari are comparable with those from Chrome is unclear.

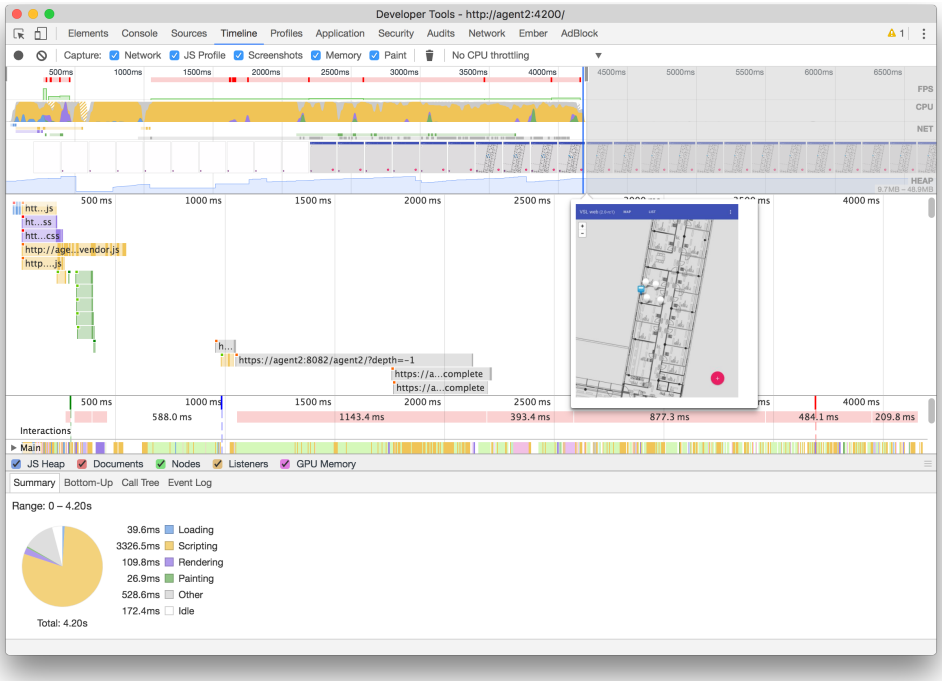


Figure 6.2: Web UI 2 RC1 in Google Chrome Developer Tools timeline with 105 devices

To measure via additional JavaScript code **within the existing app itself** exist ready to use Ember plugins like `ember-perf`[2]. This plugin allows to trace transition and render events, while the verbosity is controlled via the central configuration file. However, I trigger the same problems as above: I can not identify the parts I want to measure. During the initial call of the app, the same components are rendered several times – I get a long list with times of which I do not know, which are relevant.

[2] `https://github.com/mike-north/ember-perf`

JavaScript **benchmarks** are typically used to compare and improve JavaScript engines. However, the early benchmarks are not really realistic in today's reality: Benedikt Meurer (tech lead of Google's JavaScript execution optimization team) recommends[3] to test real websites and identifies Speedometer from Apple's Webkit Team[4] as a benchmark that makes it right:

> "There's another set of benchmarks, which try to measure overall browser performance, including JavaScript and DOM performance, with the most recent addition being the Speedometer benchmark. The benchmark tries to capture real world performance more realistically by running a simple TodoMVC application implemented with different popular web frameworks."

Perhaps the source code of this benchmark can be used for our use case – the code to be adapted is at `https://github.com/WebKit/webkit/blob/master/PerformanceTests/Speedometer/resources/tests.js`

6.2.0.3 How to collect data?

When I do the measurements myself via browser developer tools, I can simply use pen and paper. When the measurement is embedded into the app, one can set up services to collect and evaluate this telemetry data. For example, you could use the existing KA infrastructure and create a new VSL service writing to a log file. Alternatives are web analytics tools like Piwik, which not only collects the data, but also provides reporting features[5].

For the sake of completeness: By integrating the web app build and test process into continuous integration systems like Jenkins, a continuous evaluation could be implemented. Additional tests with a remote-controlled Chrome session or with Selenium[6] might be added. This would allow to understand which code change improved or worsened the actual performance.

Further influences on the tests are: Was the app compiled in production or development mode? How many entries are in the geodatabase? Has the browser cached parts of the app?

6.2.0.4 Conclusion

For the tests in this document, I decided to only measure the time from (re)start till the app has loaded everything aka 'calmed down', via the browser debugging tools in Chrome and Safari. The tests where performed with the public stable releases of these browsers, in February 2017.

[3] `http://benediktmeurer.de/2016/12/16/the-truth-about-traditional-javascript-benchmarks/`
[4] `https://webkit.org/blog/3395/speedometer-benchmark-for-web-app-responsiveness/`
[5] Piwik is probably not be best example, see `https://github.com/piwik/piwik/issues/7131`
[6] http://www.seleniumhq.org

6.2.1 Test 1: Cross browser compliance on PC

This test should show if the web app works with all three major browsers: Google Chrome, Mozilla Firefox and Apple Safari. Table 6.1 summarizes the results.

Preparation: All back-end components (KA, geo and gateway services, geodatabase) are installed on my notebook (Intel Core i5, 2x2,8 GHz, with SSD). Each browser requires the CA and individual client certificate. Some browsers use the operation system's certificate infrastructure and therefore share the certificate store – nevertheless the current KA implementation requires every browser/client to have a different certificate.

Setup 1: The KA runs at `https://localhost:8082`, the web app is served directly from the development environment via `ember serve` at `http://localhost:4200`. Tasks:

1. open app in browser
2. switch lamp2 on via web UI
3. switch lamp2 off via KA console

Hypothesis: When the lamp is switched on in the web UI the corresponding lamp in the lab lights up. When it's switched off via console, the light extinguishes and at the same time the switch in web UI slides back to off.

Result 1: The WebSockes back-channel only works in Chrome. Firefox refuses to connect due to a Cross-Origin Resource Sharing (CORS) issue: From the perspective of specification's author (Mozilla employee) the browser must not send credentials (in this case data derived from the client certificate) within the preflight OPTIONS request[7], whereas the KA implementation only accepts authenticated requests (as of mid of January 2017). The responsible Google employee confirmed that this probably a bug in Chrome. So it is pure coincidence that this setup works at all. Safari fails with an unspecified TLS error opening the WebSocket connection.

I repeat the test with slightly changed setup:

Setup 2: The KA runs at `https://localhost:8082` and now also serves resources files of the web app. By this way, browsers no longer categorize the communication between KA and web UI as Cross Origin.

Result 2: In Chrome everything works as before, in Firefox the app works too, but Safari still fails as above. A bug report[8] suggest that Safari's WebSocket implementation has no support for client certificates at all. After a discussion with the corresponding DS2OS project team member, we added an temporary workaround to the KA.

Setup 3: As in the previous test, everything is served from `https://localhost:8082` – but with following code change: When the WebSocket client does not present an client certificate, fall-back to the identity Safari uses.

Result 3: Now everything works as expected. When multiple clients are a Safari, the WebSocket return channel only works with exactly one of these. All further tests use

[7] `https://bugzilla.mozilla.org/show_bug.cgi?id=1019603#c9`
[8] `https://bugs.webkit.org/show_bug.cgi?id=158345`

a KA including this workaround. In the future, the KA should support authentication via token – at least for WebSockets. Such a token could be requested at the start of the session via REST, where client certificates work without problems.

test iteration	1	2	3
Chrome	✓	✓	✓
Firefox	×[9]	✓	✓
Safari	×	×	✓[10]

Table 6.1: Cross browser compliance of web app on notebook

6.2.2 Test 2: Finding bottlenecks

This test should reveal any bottleneck in the overall setup of KA, geo service, and web app – excluding any interference by external network connections.

Preperation: With a python script I can generate random devices with XYZ coordinates in the target area. They are named /agent2/gateway5/test3/device-XXXX. This script accesses the database directly with SQL and is configured via command line parameters: For example

- `generate-random-devices.py -N 100` – adds 100 devices,
- `generate-random-devices.py --remove` – removes all entries generated by the script from the database.

To avoid confusing errors in the browser's JavaScript console, a dummy gateway 'gateway5' is implemented. It consists of a virtual node */agent2/gateway5/test3*, which sends a valid response to any *get* request.

Setup on a single computer: the web app, one KA with geo service, PostgreSQL database and the dummy gateway.

Initially, the database has locations of five devices.

1. Open the web app with Google Chrome and open the Developer Tools Timeline.
2. Start recording and (re-)load the page, wait 10 seconds and stop the recording. The timeline automatically focuses on the active time frame, as shown in Figure 6.2. Check if the full page load was recorded, otherwise increase waiting time for future iterations. Record the 'total time' listed below the pie chart in the bottom left corner.
3. To record multiple samples, repeat step 2 five times.
4. Add 100 random device location to the database via `generate-random-devices.py`.
5. continue with step 2

[9] CORS preflight issue
[10] via workaround in KA

Result: As expected and shown in Figure 6.3, the loading time increases linearly with the number of locations being queried. The test starts failing with 505 devices, due to VSL internal implementation issue regarding virtual nodes: The HTTP GET request `https://agent2:8082/agent2/geoservice/positionOf/*` returns error 500, the KA logs reads *"Text message size [77389] exceeds maximum size [65536]"*. This is a limitation of the communication channel between KA and geo service.

This means that this setup can return up to about 405 positions, as in the currently implemented milestone 1 (see 4.2.2.1) all devices are loaded at once. Of course, for a productive deployment one would not load all data, but only these relevant to the current view – e.g. only the current floor, or only the visible devices. For smaller zoom levels one might also implement server side device clustering and only transmit a single location for the whole cluster.

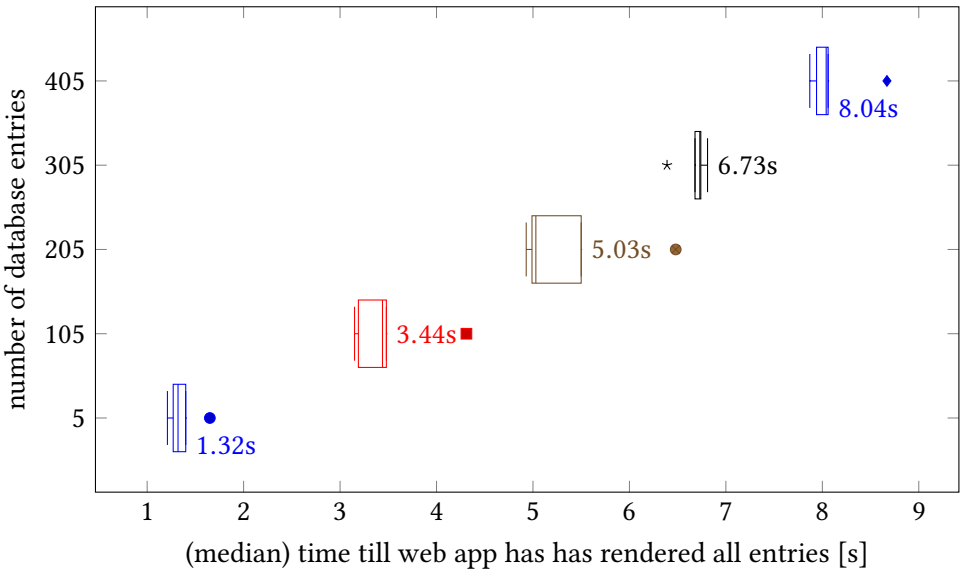


Figure 6.3: Results of test 2 on 2,8 GHz Intel Core i5 PC with 5 samples per number of entries which was performed with Google Chrome

6.2.3 Test 3: AHN lab: wire vs wireless

Running web app and KA on the same computer is probably not typical. A setup with two computers is more realistic: One runs the KA, the other (e.g. a tablet) runs a web browser with the app. To simulate this real world scenario, I deploy everything (KA, geo and gateway services, geodatabase) to a Intel Core i5-2520M (2x2.5GHz, with HDD) machine in the AHN lab. This machine has the hostname *bling* and is connected to a integrated Ethernet switch of a Wi-Fi router. See left part of Figure 6.1, below the left screen. The web app is served by the web server embedded into the KA.

Setup: I perform three test runs, all with Apple Safari on following clients:

1. notebook (Intel Core i5, 2x2,8 GHz, with SSD) via wired Ethernet,
2. the same notebook via Wi-Fi, and
3. a fourth-generation iPad (Apple A6X, 2x1.4 GHz, Late 2012) via Wi-Fi.

The time measurements are all done on the notebook via the Safari's web inspector timeline. This works also in the last test run, as the iPad is connected via USB. For each test run, the cache usage is disabled by reloading via Cmd+Shift+R. The tests with desktop Safari are performed in private mode, so browser addons do not interfere.

Hypothesis: The notebook is slightly slower via Wi-Fi then via wire. The iPad is slower than notebook Wi-Fi as it has older processor.

Result: The hypothesis was confirmed, results are shown in Figure 6.5. The concrete time value can not be read off as easy from Safari as with Chrome. My typical workaround was to take the start time of the last event, see annotation (red arrow) in Figure 6.4.

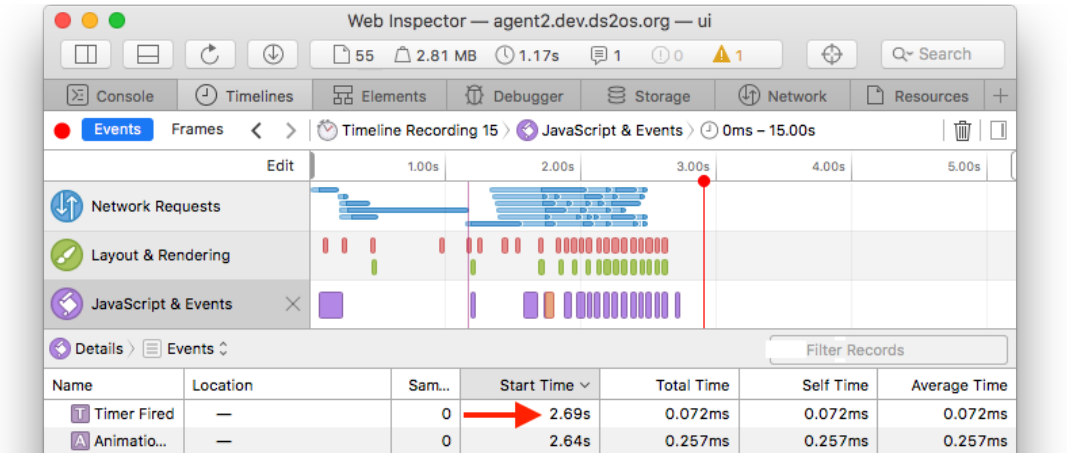


Figure 6.4: Safari Web Inspector Timelines on notebook Wi-Fi, with sample value 2.69s

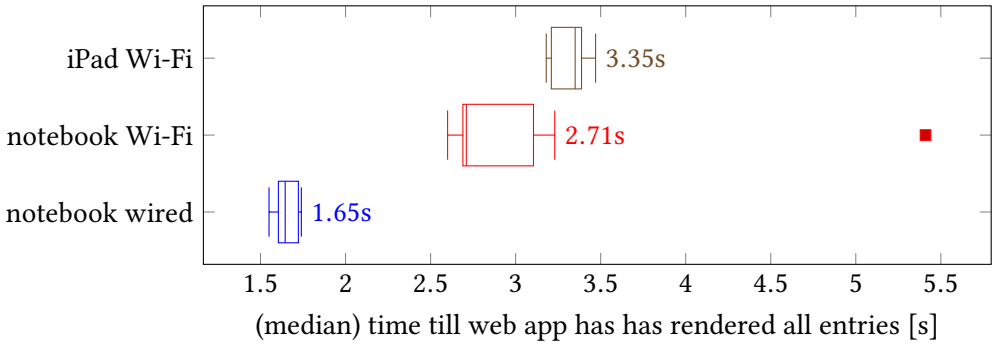


Figure 6.5: Results of test 3 performed with Apple Safari 10, per subject 6-8 samples

6.2.4 Test 4: Cross browser compliance on tablets

This test is similar to test 1, but this time on iOS and Android tablets instead of a computer with a desktop OS.

Preparation and setup: Preparation as above, i.e. install client certificates on the device, etc. Test browsers are Google Chrome, Mozilla Firefox and Apple Safari (last one is iOS only). I reuse the back-end setup from test 3.

Result: On iOS the app only works in Safari, on Android only in Chrome: Apparently the iOS versions of Chrome and Firefox have no UI to add client certificates within the browser and do not access these in iOS's key-chain for unknown reasons. Thus even the app's first REST request to the back-end fails and to no floor plan is displayed. Firefox has a the same problem on Android.

	iPad 4 (iOS)	**Galaxy Tab 4** (Android)
Safari	✓	n/a
Chrome	×	✓
Firefox	×	×

Table 6.2: Cross browser compliance of web app on tablets operation systems

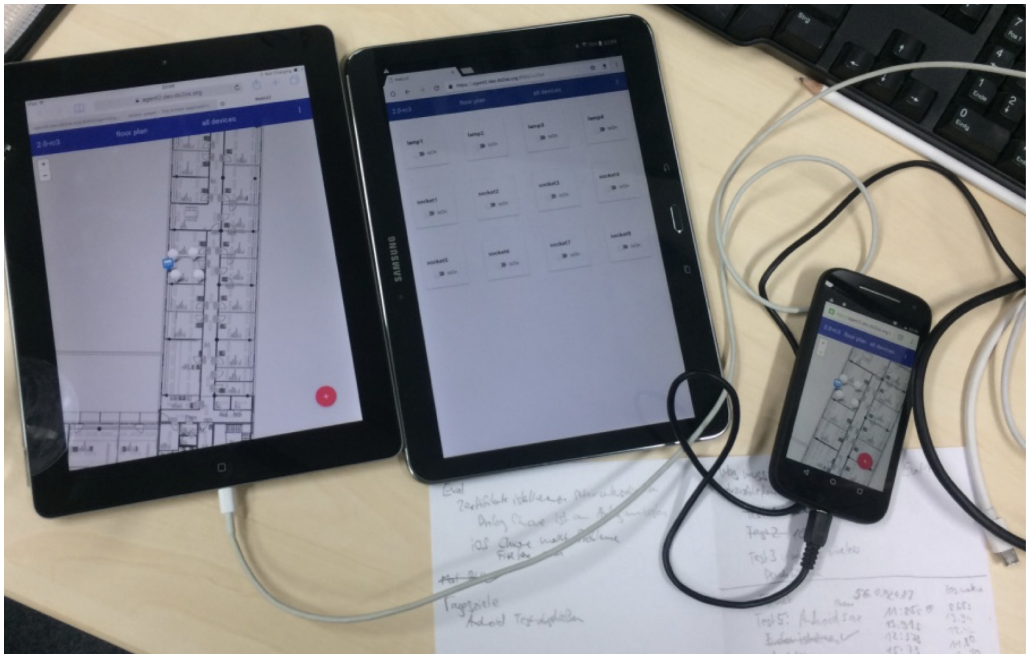


Figure 6.6: Mobile devices test setup in AHN lab

6.2.5 Test 5: Android performance

Setup: I perform three test runs, all with the same release of Google Chrome via Wi-Fi:

1. notebook (Intel Core i5, 2x2,8 GHz, with SSD)
2. tablet: Samsung Galaxy Tab 4 10.1 (ARM Cortex-A7, 4x1,2 GHz)
3. phone: Motorola Moto E – 2nd gen. 4G model (ARM Cortex-A53, 4x1.2 GHz)

Hypothesis: On Android the web app is slower by a factor of 3 to 5, as there are reports[11] of performance issues with Ember in Chrome/V8 on Android. At the time of the test, the app is using Ember 2.4. With Ember 2.10 major improvements have been introduced[12].

Result: With exactly the same release of Chrome, the web app is 2.89 times slower on Android Phone or 4.47 times slower on Tablet; both compared to Chrome on the notebook, cf. Figure 6.7. Compared to iOS Safari on an iPad 4, Chrome on the Android phone needs 3.42 times or on the Android tablet 3.75 times as long to start the app, cf. Figure 6.8.

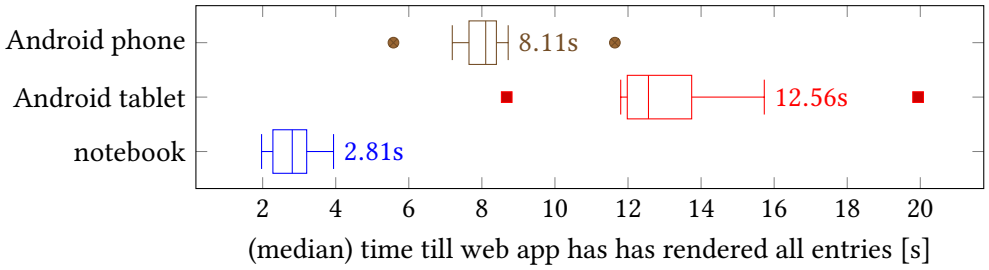


Figure 6.7: Results of test 5 performed with Google Chrome 53

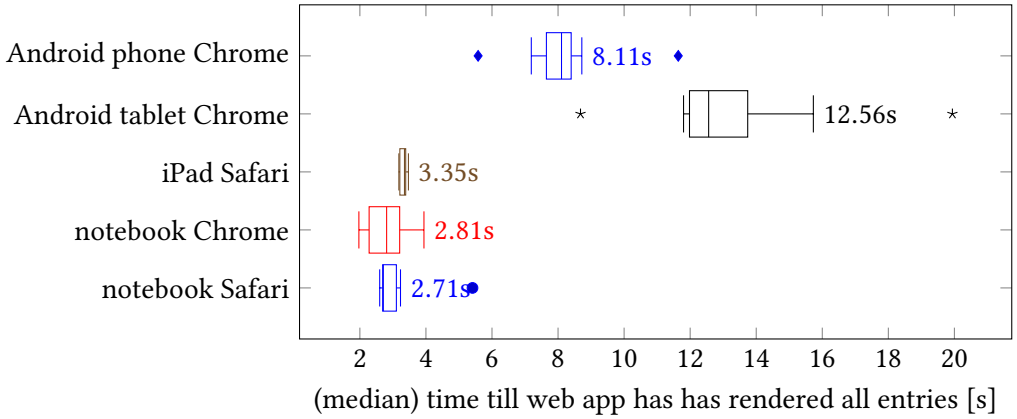


Figure 6.8: Results of test 3 and 5 combined, all connected via Wi-Fi

[11] https://bugs.chromium.org/p/v8/issues/detail?id=2935
[12] http://discuss.emberjs.com/t/why-is-ember-3x-5x-slower-on-android/6577/60

6.2.6 Test 6: UI walk-through on mobile devices

This test is a complete walk-through through the whole app with each browser: Does everything work as expected? Are there any feature differences? We know from test 4 that for iOS the app only works in Safari and for Android the app only works in Chrome.

Test sequence:

1. open app
2. switch lamp on
3. close blind
4. add a new device to floor plan
5. switch to grid view, switch a device there
6. switch a device remotely and see how the switches move by themselves

Hypothesis: In theory we can assume that the app works in both browsers, as we tested the desktop variants during test 1.

Results: With Chrome (Android) everything works as expected, in Mobile Safari (iOS) touches on UI elements yield no reaction. For switches, I added a workaround[13], sliders are still broken under iOS. In some cases the event propagation is broken, e.g. sliding the switch leads to displacement of the map and thereby the popup. Upgrading ember-paper from 0.2 to 1.0 should fix some of this issues, but proper upgrading was postponed to future work.

6.3 User study

As introduced in section 2.5.1, during UI development expert reviews (previous section) and user tests (this section) should be alternated.

Based on the available resources, I decided to perform the user tests as 'usability walk-through' as defined by Richter/Flückiger in [2]: "The user is accompanied by the test facilitator, who is moderating the test sequence. The facilitator has the possibility to intervene directly and walk through certain processes with the user. However, the facilitator must know very well how to guide the user without affecting him too much." According to Nielsen, five users are typically enough to find over 75 percent of usability problems [32]. Richter/Flückiger says one should use 4-6 people in iterative prototyping and not among 10 test persons in quality control prior to the introduction of the system [2].

I performed the user tests with six users aged 20 to around 30 in the AHN lab in April 2017. All testers were male and have a computer science background. Five use Google Android, one Apple iOS. The test setup consists of the hardware in the lab itself (lamps, blind, misc sockets; see 2.2) and the web app prototype on a Linux desktop PC in Chrome, on an iPad 4 in Safari, and an Android phone in Chrome. These are the same devices

[13] `https://github.com/saerdnaer/ember-paper/commit/b79e4eb97131e5bd697abfc14831e8ce665b0f23`

with the same browser versions as in the previous section. The audio of all test sessions is recorded for later analysis.

The tests were performed with Web UI 2 release candidate 4 (RC4) which has following known bugs: The switch does not stop UI event propagation properly, whereby sliding the switch leads to displacement of the map and thereby the popup. The slider does not react to iOS touch events. The current releases of ember-paper and ember-leaflet fix this problems – however, the type-dependent marker icons no longer work. Upgrading these libraries was postponed to future work.

6.3.1 Results

All six users found the app intuitive to use. It was totally clear to touch the markers and thereby open the popup, as they knew this concept from Google Maps. The design and place of the plus button was perceived as common object. They were initially not sure whether it was a native app or a web app. They found the performance of the app acceptable, except for one exception: During floor plan usage on the tablet one to two users complained about the not so smoothly zooming – probably due to the 5 year old and meanwhile discontinued device (Apple released the iPad 4 end of 2012). The same users were satisfied with the performance on the phone (Motorola Moto E 2nd generation, released in February 2015).

The users only touched and did not slide the switch inside the **popup**, thus the associated bug was not noticed. To use a slider for blinds was seen as a good idea, one user even found it useful for fine tuning. When adding icons to the sliders like the Material Design specification [22] suggests ("*Sliders may have icons on both ends of the bar that reflect the value intensity.*"[14]) we should use the existing blind marker icons for the closed value and brightness icons for the angle value.

Currently, new **markers** are created at a fixed position. It was suggested that we should always use the centre of the screen. One user suggested to animate the appearance of new markers with the plus button as origin. It was criticized that deletion of markers has not yet been implemented. Some proposed displaying a deletion area at the edge screen when a marker is touched (c.f. Android home screen), others a delete button in a circular context menu after a long press (cf. OpenStreetMap web editor 'iD'[15]). The user which suggested the animation above, pitched to hide the plus button and place the deletion area in that corner, thus a marker live cycle is established.

The **grid view** was well received, especially with the users who knew the predecessor (list). However, the users wanted more structure, e.g. functions for sorting; filtering; and grouping the individual tiles according to status, device type, room, or floor. One user also remarked that due to existence of the floor plan, too much spatial structuring would not make sense.

[14] `https://material.io/guidelines/components/sliders.html`

[15] `https://blog.openstreetmap.org/2013/08/23/id-in-browser-editor-now-default-on-openstreetmap/`

Some testers were positively surprised that the individual instances of the app do instant **syncing**, e.g. when a switch is flipped in one instance, it automatically flips in the others.

Surprisingly, the floor plan is also quite usable on **smaller screens** of mobile phones. I had predicted that the users would prefer the grid view on this device class.

6.3.1.1 Suggestions for improvement

All testers wished that they can change the device's **label**. The current implementation displays the last part of the VSL path as device name. The proper solution might be a 'label service' (similar to the geo service) which stores a VSL (sub-)path together with a descriptive name, maybe even with localization features. One tester even suggested personalized labels per user.

It was unclear that the texts '*floor plan*' and '*grid*' in the **top bar** are active elements, which change the view. They touched the 'floor plan' and nothing happened; they touched 'gird' and the view was changed. It was proposed to highlight the current view (aka route). A later lookup in Material Design specification revealed, that there already exist two solutions: Either a vertical menu or tabs[16]. Tabs are currently not (completely) implemented in ember-paper, cf. `https://github.com/miguelcobain/ember-paper/pull/578`. Additionally, the specification states "*Because swipe gestures are used for navigating between tabs, don't pair tabs with content that also supports swiping*", but the floor plan exactly captures these gestures to move the plan. So maybe we should choose a vertical menu after all.

Save and recall of **scenes** and other group actions were missed. When a scene is triggered, all affected devices/attributes should be displayed in a separate view. The users wanted to see how the activation affects the devices, which could be fulfilled via animation of UI elements. One user also suggested an undo function for accidentally triggered scenes.

6.3.1.2 Bugs

When users controlled the blinds via the slider, the actual behaviour of the blinds was confusing: The blinds moved down to 10%, but slider was at 50%. Explanation: The web app transmits the values immediately, even as the user is still touching the slider. Any further change is also immediately sent to the blinds gateway, but this rejects the new commands with '*Device is busy*' until the initial command is executed. It has to be clarified if this issue should be fixed in the web app, in the Java gateway service, or directly in the Arduino firmware.

One user found the snap back and white flashing of the floor plan, when zooming too far in, on the tablet too disruptive. The white flashing might result from no map tile being displayed for a short time – regular tiles have a grey or transparent background.

[16] `https://material.io/guidelines/components/tabs.html`

6.3.1.3 Survey questions

Since I had the users already there, I asked them on their opinion regarding how to implement future features:

Currently, the marker icons are purely dependent on the type of the associated device. When explicitly asked, all testers found that the state should also be included: For the current white lamp marker nearly everyone suggested to use yellow. For RGB lamps, one user recommended the current colour.

Although the markers can currently be moved directly, none of the testers accidentally moved one. They could still imagine this to happen in practice. Implementation of requirement <R.12> '*Admin and Usage scenarios are split into different UI views*' would resolve this problem – about half of the testers found this solution a good idea. Others proposed introduction of an undo feature, or snapping to the old position.

In 4.3.4 'Graphical UI-Design' I stated "*a North-oriented floor plan is suitable for Admins, where Occupants cope better with a rectangular representation.*" One tester said: "Floor plans, shown on tablets mounted to walls, should always be orientated in relation of this mount point; and only on mobile devices it should be North-oriented." Another tester complained "I do not care where North is: Walls should be parallel to the display edge."

In one test, it was discussed to enable the self-locating via GPS/Wi-Fi, which is already part of the used map library (Leaflet). The users position should be shown automatically when opening the app, and disabled when the floor plan is moved by gesture. While the locating is enabled, the floor plan should follow the users path automatically.

As the slider on the iPad (iOS) did not work (see above), I had prepared a separate test[17] with the current releases of the libraries (c.f. Figure 6.9). All testers preferred the slider from ember-paper to the 'HTML5 native slider' (`<input type="range"/>`). The paper slider was referred as 'more modern' and 'suites better to the switch'.

[17] `https://ember-twiddle.com/890df418d241195b24497c377cff4332`

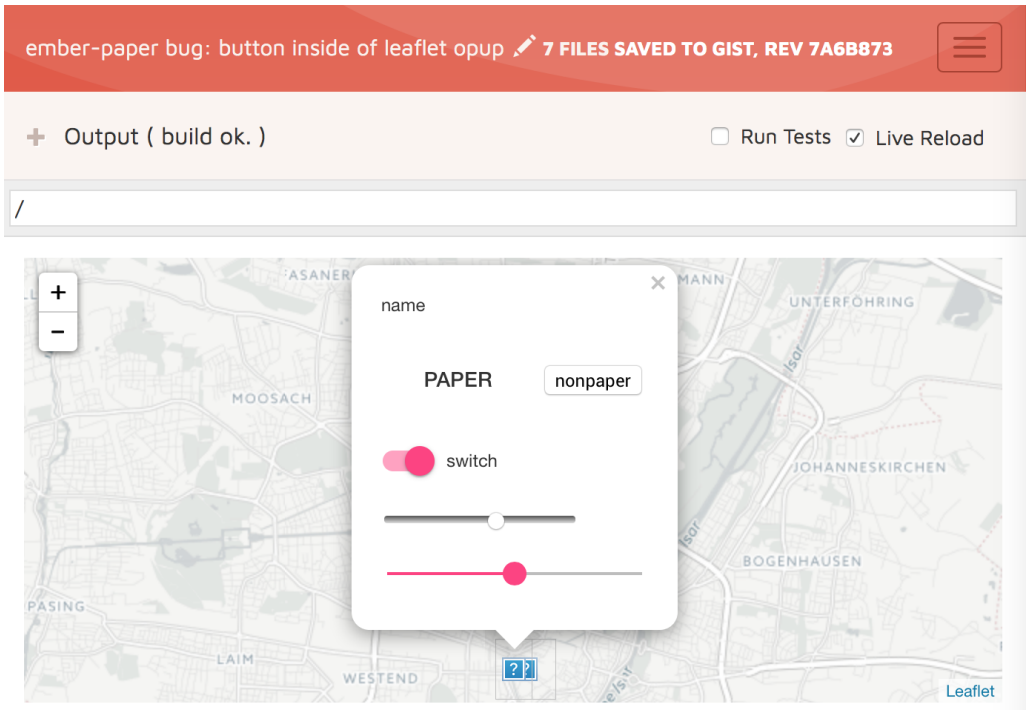


Figure 6.9: Separate test with current releases of the UI libraries

Chapter 7

Conclusion

This thesis laid the foundation for adaptive, state of the art graphical user interfaces within the DS2OS project. Besides analysis, the main results of this work are a VSL geo service and a floor plan based web UI prototype, whose design is as flexible as the VSL.

7.1 Assessment

This section checks if all issues of the problem statement are solved. Let's reiterate on the problems with today's smart space UIs initially introduced during chapter 1:

Solutions implemented for one space are not simple reusable in an other space.

- *API is not abstract enough: services use fixed devices addresses and not generic terms like 'living-room lamps'.*

  With the geo service, such queries are now possible via the VSL.
  As described in subsubsection 4.2.2.3, this query corresponds to
  get /search/deviceOfTypeIn//gahu/lamp//living-room

- *No or only insufficient exchange platforms: no app store, but forums or blogs with instructions or snippets to copy&paste, rather than easily installable software packages (apps).*

  Within the DS2OS project, there exist concepts, but currently no real implementations of SHE and Store – aside from early prototypes.

Semantic relationships are not modelled as such, e.g. they are often only represented by menu structures. Thus computers do not know, which device is in which room.

> This problem is addressed by the geo service: 'In which room is this device?' translates to *get /search/locationsReverse/<device-path>*, and 'Which devices are in this room?' to *get /search/locationsIn/<name>*.

Some UI apps are not platforms in-depended, e.g. ActiveX 'web UI' requiring a Windows desktop PC; or the devices UI app only exist for iOS but not for Android etc.

> As the UI created within this work is a HTML5 web app and thereby usable on all platforms, for which a current web browser is available.

Users themselves can not adapt the UI, but need to call in (external) experts for changes.

> When comparing the current state of the Web UI 2 with the real world as described in 2.1.5.4, it is already much easier to customize our UI. One only needs 'web developer knowledge' and no special hardware. Once the UI extensions store (which was not part of this work) is implemented, UI customizing can be done by the common user.

Solutions are more expensive than they need to be: Instead of mounting Android tablets to the wall, five times as expensive touch screens with a severely restricted feature set are installed.

> Web app is usable on cheap Android tablets (as soon as the app was loaded, cf. 6.2.5).

Vendors create silos for their devices:

- *UIs are distributed to different apps or mounted in different locations in a room.*
- *System overlapping group actions (scenes) are not possible due to non-existent, proprietary or incompatible APIs.*

> The VSL, its gateways, and other services already solve the silo problem. However, no one has implemented a real group/scene service yet.

New users have to find out which devices can be controlled and how these devices are labelled in the UI, e.g. "Where is lamp 3?".

> This problem is addressed by using a floor plan. Nevertheless the user study (6.3) showed renaming devices (introduction of a 'label service') is still desired.

7.2 Future work

This section lays out remaining issues and tasks to be addressed in future.

VSL architecture / Back-end

Currently, the Web UI is only guaranteed to work correctly, as long as only one client is a Safari browser. Each additional Safari would fall back on the same certificate as the first one, which makes the return channel unreliable (c.f. 6.2.1). To solve this issue, the design has already been modified [33, sec. 4.3.6f], but not yet been implemented. In addition, the KA currently has a response limit of 65kB for virtual nodes (6.2.2). Again,

there is already a draft how to fix this [33, sec. 4.3.7.4], but to my knowledge it is not implemented as of June 2017.

The VSL should be extended with arrays for return values, where an array is a collection of multiple values without a key. Currently, typeSearch and geo service send one string including dedicated separators, which the client has to split manually. (see *get /typeSearch/<type>*, *get /search/devicesWithoutLocation*, or *get /search/locationsReverse/<name>*.)

Analogous to the HTTP protocol, *delete* should also be added as VSL method besides *get* and *set* (4.2.2).

Due to external dependencies, I could not tackle Debugging <R.13> and Traceability <R.27>: There is no reasonable SHE implementation providing the necessary APIs, such as programmatic access to logs, etc. As of May 2017, it is not yet possible to transfer the services used at the AHN lab setup from the Screen session (2.2.2) to OSGI bundles. Regardless of this, one could analyse implementing a SHE with systemd or NixOS instead.

To provide ready to use, 'compiled' extensions for end users it has to be decided whether the UI extensions are distributed via the S2Store or a separate one (4.2). A possible architecture is a central UI Extension Store, in which developers upload the source code of their new components, see subsections 4.3.3 and 5.3.4.

UI

Low hanging fruits are experiments with existing Leaflet plugins: The Clustering plugin should resolve <R.5> and partial <R.6>. The heatmap plugin could be used to display temperature (see 4.3.4.1).

Finish upgrade from Ember 2.6 to current release, see corresponding git branch at `https://gitlab.dev.ds2os.org/ds2os-devs/web-ui2/commits/feature/ember-upgrade`.

Upgrade ember-paper from 0.2 to 1.0, which introduces new features and components like dialog, but requires modification of templates. Replace HTML5 slider with PaperSlider. Add vertical menu (6.3) to replace floor plan and grid links in top bar.

Replace bitmaps map tiles with vector data <R.7>: Preparatory work has already been done – the building part of our research group/chair is available in the appropriate format. There exists a python script fetching indoor data from the OpenStreetMap project[1] and importing it into the geodatabase. There is a GeoJSON plugin for Leaflet (the web maps library used in Web UI 2), which should be able to consume the response of *get /search/geometriesIn/<floor>*.

Then the floor selector <R.4> can then be implemented according to the specification in 4.3.4.1 and 2.5.2.4. Additional GUI challenge: How to switch levels, while holding device icon, when moving devices across floors; e.g. TV from ground floor to upstairs.

[1] Side note: Other parts of the MI building are also available from OpenStreetMap.

Then split the floor plan into Usage and Admin view <R.12> – while Usage view does not display all sensors (e.g. push buttons).

Filter by device type, e.g. lighting, HVAC, entertainment, security etc. – c.f. free@home (3.2).

Switch to Typescript? Angular and React both use Typescript as default scripting language – in long term Ember will switch to Typescript too, c.f. glimmer.js [30]. Perhaps, inheritance from Dobject to Device (4.3.2) might improve, but that is currently not really an issue.

VSL services

Refactor current openHAB gateway from static approach to dynamic model to use VSL */basic/list* type, and retrospection features of the openHAB API. Enables realization of auto-discovery and -configuration <R.16>, add 'inbox' view to Web UI 2 as shown by Paper UI (3.1) [27].

Create a VSL scene/group service allowing save and recall of presets for a collection of devices (6.3.1.1).

Create an VSL extension service, which stores metadata like which user installed which UI extensions and notifies other UI instances when a user installed an UI extension so they can get in sync without manual reload (4.3.3).

Geo service: Make *positionOf* subscribable, so other Web UI 2 instances are able to get notified when device locations are added or changed.

Allow linking devices in the floor plan view, e.g. a lamp with a wall switch (3.2). Example 1: When connecting two devices of the same system – e.g. switch and lamp, the switch should be automatically configured to the lamp's address. Thus, the button touch reaches the lamp directly, and does not has to be routed through the VSL. For example a KNX switch should control a KNX lamp directly. But, when the user connects a KNX switch with a non-KNX radio controlled outlet, the VSL comes transparently into play. As of today, such configuration is seen as gateway/system specific and therefore not accessible via the VSL. Additional challenge: Connecting devices across floors.

Evaluation of different database implementations and schemas: How does PostgreSQL with PostGIS perform on a Raspberry Pi (or similar) in contrast to SQLite with SpatiaLite, for the data of a family home? Also evaluate 3D vs. full-3D vs. 2D geometry. (7.3)

The content of the KA HSQL database could be moved into the same database as the geo data (2.4.2).

7.3 Discussion

In my opinion the fundamental goals were achieved.

Front-end: In subsubsection 2.5.6.2, I quoted an article recommending to try out Ember, Angular and React. In this work I only choose Ember, so should we implement the basic features in other two frameworks, to make a good decision? Can one make a good decision only after having implemented the basic features in all three frameworks, once? I do not know – With Angular, Material Design would probably have been easier to implement and the UI event propagation bugs might not have occurred. At the time of the framework selection, Angular 2 was not yet announced, and it was unclear how the Angular ecosystem would continue. Meanwhile in March 2017, Angular 4 has been released. React is probably still interesting, especially because of React Native's features to create a native app from the same sources as the web app. Nevertheless, Ember's components concept was very helpful to implement the UI extensions and the upgrade between Ember releases were feasible. At this point, I close with a quote from [34]:

> "So how do you know what tool to use to build a modern web application? I would recommend that you look at the demographics of your organization to figure out which framework will suit best."

Back-end: A goal was to handle relations directly in the 3D space. I expected to take 3D geometries and everything works. It turned out that PostGIS – and maybe even geodatabases in general – have no support for 3D solids and only store 3D surfaces. It is almost impossible, to represent 3D surfaces in for humans easy readable text. Probably, one should rework the geodatabase schema in a future iteration: Possible variants are: Two geometry columns – geom_2d and geom_3d. Storage/caching of the centroid in a separate column. Another variant is to store the room's base area as 3D polygon and use a separate column for the height to allow extrusion. Or only store 2D geometries and use separate columns for floor level and height. In PostgreSQL, the latter could be realised via the numrange[2] type.

The initially chosen approach with ST_Contains (5.1) has its limits and makes the queries unnecessarily complicated as ST_Contains has no real support for PolhedralSufaces. The problems can be bypassed[3], but only in a very unpleasant way. A rewrite of these queries using PostGIS full-3D functions (e.g. ST_Within3D) seems reasonable. The server at the AHN lab (*bling*) was meanwhile upgraded and the `postgis_sfcgal` PostgreSQL extension is ready to use (5.1).

For future UI theses, I would recommend to set the focus either on homes or on commercial buildings. Both at the same time is too complex for single person at one point in time. Maybe also involve students from design and ergonomics.

[2] `https://www.postgresql.org/docs/current/static/rangetypes.html`
[3] `https://lists.osgeo.org/pipermail/postgis-users/2017-June/042191.html`

Bibliography

[1] M.-O. Pahl, "Distributed smart space orchestration," Dissertation, Technische Universität München, München, 2014. [Online]. Available: http://www.pahl.de/download/publications/dissertation_pahl_2014.pdf

[2] M. Richter and M. Flückiger, *Usability Engineering kompakt: benutzbare Software gezielt entwickeln.* Spektrum Akademischer Verlag Heidelberg, 2013. [Online]. Available: http://doi.org/10.1007/978-3-642-34832-7

[3] J. Bortz and N. Döring, *Forschungsmethoden und Evaluation: für Human- und Sozialwissenschaftler.* Springer, 2002.

[4] M. O. Pahl and G. Carle, "Crowdsourced context-modeling as key to future smart spaces," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, May 2014, pp. 1–8. [Online]. Available: http://doi.org/10.1109/noms.2014.6838362

[5] C. Ebert, *Systematisches Requirements Engineering: Anforderungen ermitteln, spezifizieren, analysieren und verwalten.* dpunkt. verlag, 2012.

[6] Apple Inc., "HomeKit Developer Guide." [Online]. Available: https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/HomeKitDeveloperGuide/FindingandAddingAccessories/FindingandAddingAccessories.html

[7] S. Pole, T. Knerr, P. Barth, and A. Hubel, "Simple indoor tagging," 2014. [Online]. Available: http://wiki.openstreetmap.org/wiki/Simple_Indoor_Tagging

[8] A. Martinez, "How to choose a geological relational database management system," August 2015. [Online]. Available: http://opengeostat.com/how-to-choose-geological-database-management-system/

[9] J. Herring, "OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture," Open Geospatial Consortium, Tech. Rep., 2011. [Online]. Available: http://portal.opengeospatial.org/files/?artifact_id=25355

[10] H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, and T. Schaub, "The GeoJSON Format," Internet Requests for Comments, RFC Editor, Tech. Rep., August 2016. [Online]. Available: http://doi.org/10.17487/RFC7946

[11] K.-P. Engelbrecht, P. Ehrenbrink, S. Hillmann, and S. Möller, "Messung und Bewertung der Usability in Smart Home-Umgebungen," in *VDE-Kongress 2014*. VDE VERLAG GmbH, 2014.

[12] S. Moeller, K.-P. Engelbrecht, S. Hillmann, and P. Ehrenbrink, "New ITG Guideline for the Usability Evaluation of Smart Home Environments," in *Speech Communication; 11. ITG Symposium; Proceedings of*, Sept 2014, pp. 1–4.

[13] Bundesministerium für Verkehr und digitale Infrastruktur, "Building Information Modeling (BIM) wird bis 2020 stufenweise eingeführt," Pressemitteilung, 2015. [Online]. Available: http://www.bmvi.de/SharedDocs/DE/Pressemitteilungen/2015/152-dobrindt-stufenplan-bim.html

[14] A. Borrmann, M. Hochmuth, M. König, T. Liebich, and D. Singer, "Germany's governmental BIM initiative–Assessing the performance of the BIM pilot projects," 2016. [Online]. Available: http://www.cms.bgu.tum.de/publications/2016_Borrmann_BIMPilotProjects.pdf

[15] J. Kozel, "How to Visualize Indoor Data in 2D Map? Is This the Way to Go?" in *FOSS4G 2016*. Masaryk University, 2016. [Online]. Available: http://doi.org/10.5446/20394

[16] Busch-Jäger Elektro GmbH, "ABB-free@home® System Manual," April 2015. [Online]. Available: http://new.abb.com/docs/librariesprovider84/freeathome/system-manual_free@home_en_abb_05_03.pdf?sfvrsn=2

[17] (2014, June). [Online]. Available: https://youtube.com/watch?v=P3SxkcKGZpU#t=27s

[18] M. Weiser, "The computer for the 21st century," *Scientific american*, vol. 265, no. 3, pp. 94–104, 1991.

[19] G. Gruman, "The iPad's victory in defining the tablet: What it means," July 2011. [Online]. Available: http://www.infoworld.com/article/2622583/tablets/the-ipad-s-victory-in-defining-the-tablet--what-it-means.html

[20] B. Shneiderman, *Designing the user interface*, 5th ed. Upper Saddle River, NJ: Addison-Wesley, 2010.

[21] Apple Inc., "iOS Human Interface Guidelines." [Online]. Available: https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/

[22] Google Inc., "Material design specification." [Online]. Available: https://www.google.com/design/spec/

[23] C. O'Sullivan, "A Tale of Two Platforms: Designing for Both Android and iOS," April 2015. [Online]. Available: http://webdesign.tutsplus.com/articles/a-tale-of-two-platforms-designing-for-both-android-and-ios--cms-23616

[24] J. Nielsen, *Usability engineering*. Elsevier, 1993.

[25] D. M. Jones, "The 7±2 urban legend," in *MISRA C Conference*, 2002.

[26] Z. Kuhn, "Choosing a Front End Framework: Angular vs. Ember vs. React," October 2015. [Online]. Available: http://smashingboxes.com/blog/choosing-a-front-end-framework-angular-ember-react

[27] K. Kreuzer, "openHAB 2.0 Paper UI Preview," November 2014. [Online]. Available: https://youtube.com/watch?v=NolVoL8ewO0

[28] K. Kreuzer, "Home Automation Reloaded," online, May 2016. [Online]. Available: https://youtube.com/watch?v=hPX4wAxsbxA

[29] J. Martin and S. Institute, *Managing the data-base environment.* Prentice-Hall Englewood Cliffs (NJ), 1983.

[30] M. Otte-Witte, "Feel the Glimmer," June 2017. [Online]. Available: https://youtube.com/watch?v=vIRZDCyfOJc

[31] T. Ornelas, "Using ember with webpack," February 2016. [Online]. Available: https://medium.com/@tulios/using-ember-with-webpack-e03290b61dec

[32] J. Nielsen, "Why you only need to test with 5 users," 2000. [Online]. Available: http://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/

[33] F. Kuperjans, "Native Service Interfaces for the Virtual State Layer," Master's thesis, Technische Universität München, April 2017.

[34] T. Mankovski, "Choosing a frontend framework in 2017," June 2017. [Online]. Available: https://medium.com/this-dot-labs/building-modern-web-applications-in-2017-791d2ef2e341