Cloud architecture for Smart Homes - an implementation for the HOMER platform

Anders Jensen

Kongens Lyngby 2016

Abstract

This master thesis presents a cloud architecture for smart homes running the HOMER platform by AIT. The architecture achieves remote access to data stored on local servers in smart homes, by foregoing the usage of port forwarding, VPN's or other remote access techniques and instead storing all smart-home related data on a central database server. By doing this we can allow users do modify the smart home configuration running on the local server (also called Home gateway or Hub), using a developed app or webpage. The app and the webpage communicate with the server through a client API, which is implemented on the server as a RESTful service oriented architecture.
To facilitate communication with the HOMER platform running in the smart home, an OSGI bundle has been developed that synchronizes the central server with the local server. The proposed and implemented architecture seems promising, since it foregoes traditional remote access techniques such as port forwarding or VPN's and instead actually moves all the data to a cloud server. This makes the system easier to maintain and develop further on.

Acknowledgements

I would like to thank my advisor Lukas Roedl from AIT, for all the support he has given me and for always being available to discuss ideas with. He has been a great help throughout the entire project and has helped me refine the project goals as the project developed. He's proofreading has also been greatly appreciated. Secondly i would like to thank my advisor at DTU Christan Damsgaard Jensen, for being available to discuss project research topics with and for generally providing guidance. Lastly a thanks also goes out to Emanuel Sandner at AIT for technical discussions and sparring.

Contents

Chapter 1

Introduction

Home automation has become increasingly popular over the last decade and is becoming more frequently integrated in modern homes. The main reason for introducing computer based control systems in homes, is to increase the comfort level by automating tasks and offering inhabitants extended control over their homes, for example through their smart phones. Other benefits could be reducing consumption of resources such as electricity, heating and water.
However, home automation is still not a very common technology to have in homes, because it is too difficult for non-tech users to set up and because the system needs to be maintained, which often involves working with very non-user friendly platforms. The underlying problem is often the myriad of incompatible communication protocols and standards that exists on the market and previous research [GYYL09] [BMST10] [HMK+05] has focused a lot on developing application layer software that allows other software developers to abstract from the underlying protocols.
Even though some companies have this smart piece of software running in the house, allowing devices using different protocols to speak with each other, then they still have no means of remotely maintaining that software. Users are left to maintain their own system (and often also set it up themselves), unless they will pay for having a company expert come out to the house and configure it for them.

For this project i have worked with Austrian Institute of Technology (AIT), which has such an application layer software platform as described above. The platform is called HOMER (Home event recognition) and is freely available at (http://homer.aaloa.org/).

1.1 Problem definition

At the moment the AIT is facing the issue that they have a piece of well developed software (in this case HOMER), but with no means of remotely installing or configuring it in peoples homes. When setting up new smart homes, a technician always needs to go to people's houses and spend a lot of time in their house, while installing the devices at the correct locations, setting up a local server, pairing devices with the local server etc. Additionally, the software running on the local server needs to be configured exactly in the right way before the technician leaves the house, since he cannot modify the software once he has left the house. This means that users must re-configure the house themselves, if something afterwards doesn't work as it should, or if they think of a better way the sensors could be configured etc. This is not a very flexible setup and it means that technicians will have to spend a lot of time in people's homes, which is expensive for the customers as well as for the company and creates a poor user experience. In order to solve these issues a solution is required that speeds up this process.

1.2 Contributions

This thesis will contribute to public research by developing and implementing a prototype architecture that provides remote access to smart homes, by moving all smart home related data from local server in the houses, onto the cloud. This solution makes it unnecessary to obtain remote access to the smart home by using e.g. port forwarding or VPN's, since it will store and manage all the date on an online server. Software used to run the smart homes should still be stored and run from a local server in the house to ensure offline functionality, but it should also be stored on the cloud. The local server and the cloud will need to synchronize with each other.

In this thesis the following has been developed and implemented:

- A central server (the cloud server), with a database storing all device-related information from all houses on it. The server offers a client API for clients to consume.

- A software bundle to be deployed in houses running HOMER, facilitating synchronization with the central server.
- An app allowing users to log into a house and get an overview of the devices present in the house.
- A web page for managing all the device stored on the central server.

1.3 Thesis overview

The remainder of this paper is organized as follows.

- Chapter 2 gives the reader an introduction to the smart home field, by discussing some of the previous work that has been done in the field, and presenting the smart home market. Afterwards, a thourough background on the software architecture of a smart house is provided, ending the chapter by describing the HOMER platform.
- Chapter 3 elaborates on the problem definition in more detail and concludes that the problem is the lack of remote access. Afterwards, it discusses the solutions with which one can achieve remote access and concludes that moving the architecture online is the most flexible and extendible solution. The chapter ends with a specification of the requirements for the system consisting of: A central server, a HOMER synchronization bundle, an app and a web page.
- Chapter 4 presents the design of each of the 4 proposed system components. It describes in detail how the various clients communicate with the server using a RESTful client API and how the server accesses the central database using a DAO interface. The synchronization procedure between the HOMER bundle and the central server is also described in detail, as well as the structure of the android app and its UI components.
- Chapter 5 describes the implementation details of the system, showing the reader the general coding patterns used for implementing the 4 system components. Extra attention is also given to certain pitfalls or tricky implementations.
- Chapter 6 evaluates the 4 developed components, in relation to the stated requirements for the system given at the end of chapter 3. A few tests on the system are presented as well, including a stress test of the central server and space consumption of the central database.

- Chapter 7 finally sums up the conclusions of this thesis.

Chapter 2

State of the art

2.1 Research in the Home Automation field

Since the early 2000's, home automation has become an increasingly popular field of study, although one of the oldest home automation protocols X10, was invented in 1975. By the early 90's many companies had gotten in to the market, and a myriad of incompatible standards had started to appear [Wac02]. [Wac02] and [VF02] from 2002, were some of the early works that addressed the problem of protocol heterogeneity. As a solutuon to this problem, [VF02] proposes an OSGI Service Platform that includes a common device abstraction layer, mapping devices from all sub networks to the same common entities. It also suggests providing a common programming interface, so that developers can develop applications for the platform without knowing anything about the underlying device protocols or how the individual devices connected to the system are configured. It concludes that all companies developing home automation products, should each provide a piece of OSGI-compatible firmware capable of managing their products. To the knowledge of this author, this was one of the first real proposals of a software architecture to solve the problem of incompatible protocols at the application layer.

In 2005 the Gator Tech Smart House was created [HMK+05], which was a university made project that very nicely described the complete architecture of a Smart House, including all the sub layers of the system. It even implemented the described system in a big model house. A few years later the myriad of protocols and standards on the market, still posed a big problem and a lot of the research within home automation was being dedicated to this topic [MTMT06] [BMST10] [GYYL09][PRL+08] [KBY+12] [WKLA13]. The general solution amongst these papers is to have n gateways (one for each subnetwork), and connect them to a service layer that maps all of the different protocols to a common language. Thereafter they have some software in the application layer that implements a client API on top of the common mapping, allowing developers and/or users to work with devices connected to the sytem in a uniform manner regardless of device protocols. This last step differs quite a bit, with some of the articles not specifying how this last part is done, some of them implementing a new "programming" language to be used, while others suggest the idea of modelling devices as RESTful resources. [KBY+12] [WKLA13] from 2012 and 2013, build upon the mentioned architecture adding a cloud component to the system capable of automatically installing device driver components to the system. They suggest having all gateway-specific firmware installed on a cloud platform and then downloading drivers for new subnetworks/protocols when the system detects a new sub network (this still requires that the cloud is updated when new protocols are invented).

Most of the articles above have in common, that all software is placed on some server running locally in the house (often combining this server with the gateways into one big "hub"). This also makes sense since most devices on the market back then (and still today) communicate over low-rate protocols (802.15.4), requiring dedicated local gateways to pick up their radio signals. However, over the last few years IP-based protocols have become increasingly popular in the home automation market, despite using more power for sending their signals. [DGV09] argues that embedding mini webservers into every device communicating with TCP/IP, is not as bad power-wise as the industry tends to believe. By implementing some memory optimization and using off-line precalculations it shows that the expensive TCP/IP communication can be done it a more efficient way. [GTW10] [KP12] suggest embedding mini web servers into all non-IP speaking devices and then having all devices communicate directly with a web server, which means that no local server would be used at all. This will be discussed further in section 3.2

2.2 Home automation market

There is currently a standardization "war" going on in the Home Automation market, where each company is striving to make their protocol the de facto protocol for home automation [Wro14] [Kas14]. It is similar to the VHS vs. Betamax standard war, back in the 70's and 80's [Wik15b]. Therefore the companies form alliances, to gain as many supporters as possible. Such alliances include the ZigBee Alliance, the Z-wave Alliance, the Thread Group, the AllSeenAlliance, the UPnP Forum and the Digital Living Network Alliance (DLNA) and many more, each of them claiming they have the best solution. The result is that vendors developing and selling smart home devices to consumers, will have to chose a protocol (or a suite of protocols), that their products will support. But in the end, it is really the consumers who suffer the most, since if a consumer goes out and buys a ZigBee device and a ZigBee gateway to communicate with it, he will be limited to only ZigBee products. If he then later sees a clever device communicating over the new Thread protocol, then he will also need another gateway that speaks Thread, and it will get even more complicated if he want those two devices to interact with each other (having the Zigbee-speaking motion sensor, trigger the Thread speaking device). This makes home automation cumbersome, expensive and something only tech-users do. Regular consumers don't know about protocols and don't care about protocols [Wro14].

The optimal solution to this problem, would be that the whole world chose one protocol as the standard to adhere to. This is of course an unrealistic scenario even for large companies such as Google or Apple. However, both of these companies joined the home automation market in 2014 - Google having acquired the NEST company developing the Thread protocol and Apple having started development on their own proprietary Apple HometKit system [Mer14] [Quo14]. In the meantime until the number of protocols on the market have converged to a only a few ones, the next-best solution for the average smart-home vendor, is to ensure that your products support as many protocols as possible. What is equally important, is that you have a really user friendly implementation that also appeals to non-tech users.

Some of the newer companies on the market have thought

2.3 Technological background for smart homes

This section will present the reader to the relevant protocol layers of a smart-home architecture, relating them to the OSI model and where standards, technologies and brands fit into. Very often companies mostly focus on selling and up-talking their products rather than specifying exactly what they are. To clarify this, figure 2.1 shows where in the OSI model a technology, standard or brand belongs. The grouping of the technologies, standards and brands in layers does not mean that each element of a layer works with every element of another layer. For example UPnP only works with TCP/IP while Apple Homekit only works with Bluetooth and TCP/IP.

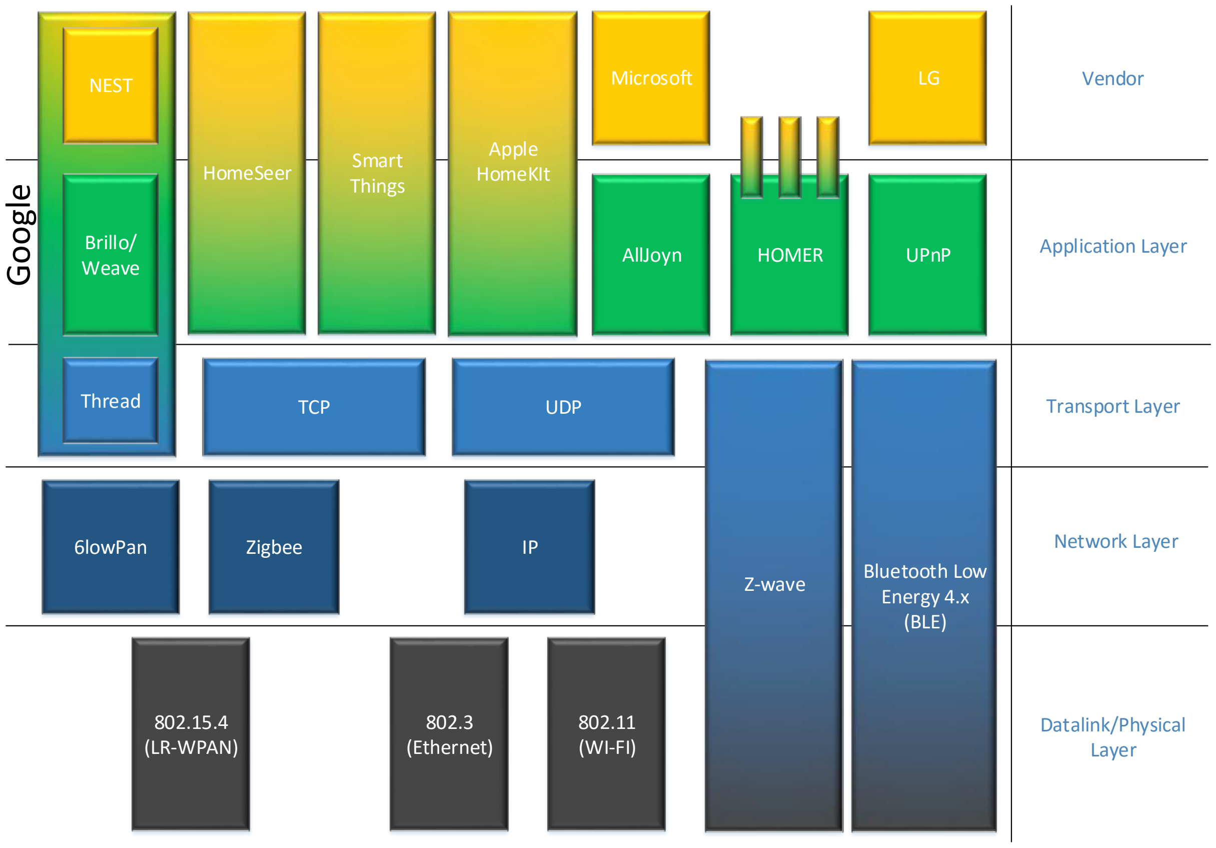


Figure 2.1: SmartHome market and the OSI layers.

Figure 2.1 clarifies the relationship between the various technologies and standards on the market. It is important to understand the difference between the layers when discussing them, since it makes no sense to for example say that Apple HomeKit is "better" than ZigBee or that NEST works better than Z-Wave. Apple HomeKit is a development framework and toolkit created by apple and intended to be used over Bluetooth or IP [Quo14], while ZigBee is network layer communication technology using the 802.15.4 phy layer and

TCP as its transport layer [Ash13]. Theoretically Apple HomeKit could even run over ZigBee if Apple had chosen to go that direction. Comparing Apple HomeKit with SmartThings and HomeSeer is 100 % valid, since they are all 3 companies that have built their own application layer, working on top of a few select transport/network protocols (apple; Bluetooth & IP, SmartThings; IP, ZigBee & Z-wave; HomeSeer; Z-wave, X10, Insteon & IP [Sma16] [Hom16]). All 3 companies have created their own pseudo "programming" language for customers and other developers to use when configuring their houses. You can even create apps with these configuration languages that other people can use (provided they use products from the same company).

2.3.1 Physical/Datalink layer

As we know from the OSI model, the Physical/Data Link layer is the bottom most layer of the OSI model describing the physical means of transportation used in the network together with the very basic data exchange protocols used. This layer defines the very basics of a network, i.e are we using Ethernet cables, power lines or radio waves (and which frequency) for the communication in the network. Examples of standards defining the Physical and Data Link layers are the 802.x family of standards, for example 802.15.4(LR-WPAN), 802.3 (Ethernet) and 802.11 (WI-FI). Other standards at this level include the proprietary Z-wave stack and Bluetooth 4.0 (also called BLE - Blueooth Low Energy).

In relation to the field of smart homes, the main requirement for the physical/-datalink, is that the devices in it should be able to operate with a very low power consumption, since most smart home devices run on batteries and changing batteries is an unwanted burden for the inhabitants. That is exactly what the 802.15.4 [CGH$^+$02], BLE and Z-wave standards are developed for. These are standards where the maximum transmission unit (MTU) of the packages sent have been reduced, in order to achieve low power consumption and low speed at the cost of bandwidth. On the opposite side of the spectrum we find well-established standards such as 802.3 (Ethernet) and 802.11 (WI-FI) that are already widely used with the Internet, and which offer large bandwidth, but with no special consideration towards power consumption. Therefore devices operating over Ethernet or Wi-Fi most often operate in the context of ample power supply and highly capable devices.

2.3.2 Network and Transport layers

The network and transport layers describe how data is packaged in the network and how it is transported. The requirements for a smart home are again the same: low power consumption. For this purpose the 802.15.4-family, Z-wave and bluetooth protocols have been developed, where in order to achieve low power consumption, the data rate and the maximum packet sizes have been drastically reduced i.e 128 byte max packet size and 250 kbps throughput for the 802.15.4-family, 22 byte for BLE and 270 kbps and 64 byte and 40 kbps for Z-wave. It should be noted that the stated throughputs for 802.15.4 and BLE are the raw maximum data throughput, meaning that on the application level this throughput will be significantly lower. This is in contrast to the well-established IPv4 and IPv6 which allow for much larger throughputs, but again with no consideration towards power consumption. 6LoWPAN is a promising protocol specially developed for the IoT (Internet of Things) and also for smart homes. It allows IPv6 packets to be sent and received over 802.15.4 networks, and it implements header compression techniques to reduce the packet size.

2.3.3 Application and Vendor layer

The application layer in a smart home consists of several parts, amongst them a mapping part, configuration management and developer services. The mapping integrates one or more protocols and maps them to a unified data structure allowing further development to abstract from underlying protocols. As an example, this means that a motion sensor using the ZigBee protocol and another motion sensor using an IP-based protocol will simply be mapped to the same entity called "motion sensor". In this way all "kinds" of devices from various protocols should be mapped to common entities. Preferably the mapping is done using some standard such as ISO/IEEE 11073 [Wik15a].

Configuration management regards the creation and management of house plans (often also called a configuration or a house configuration). This means having some means for the user to create and manage a house plan with. Although it is a very un-user friendly solution, most solutions today involve creating a new if-else-programming language/mapping language that users are supposed to use themselves to create and manage the configuration of their house with. Along with this a UI usually follows visualizing the house and sometimes semi-replacing the programming language by implementing rudimentary drag-n-drop mapping.

The open-source application layer projects such as AllJoyn, Weave and HOMER include developer services in them. They are extra infrastructure services that other developers can use to extend the system with such as easy database access, frameworks for supporting new protocols or frameworks for extending the configuration management tools and much more. Proprietary systems have developer services too but they are kept within the companies developing them (or subcribing to them in the case of Z-wave and Apple HomeKit MFI).

The difference between the vendor "layer" and the application layer might seem vague, but the point is that some companies chose to implement existing application libraries and sell their products using them (Microsoft and LG), while many other companies chose to both develop their own libraries and afterwards also implement them in apps and devices etc (SmartThings, HomeSeer and Apple). Projects such as AllJoyn, HOMER, Weave and UPnP seek to establish open-source application layer libraries for vendors to use and implement.

2.4 High level architecture of a smart house

This section will give the reader an overview of the high level architecture of most smart houses. Figure 2.2 shows this architecture.

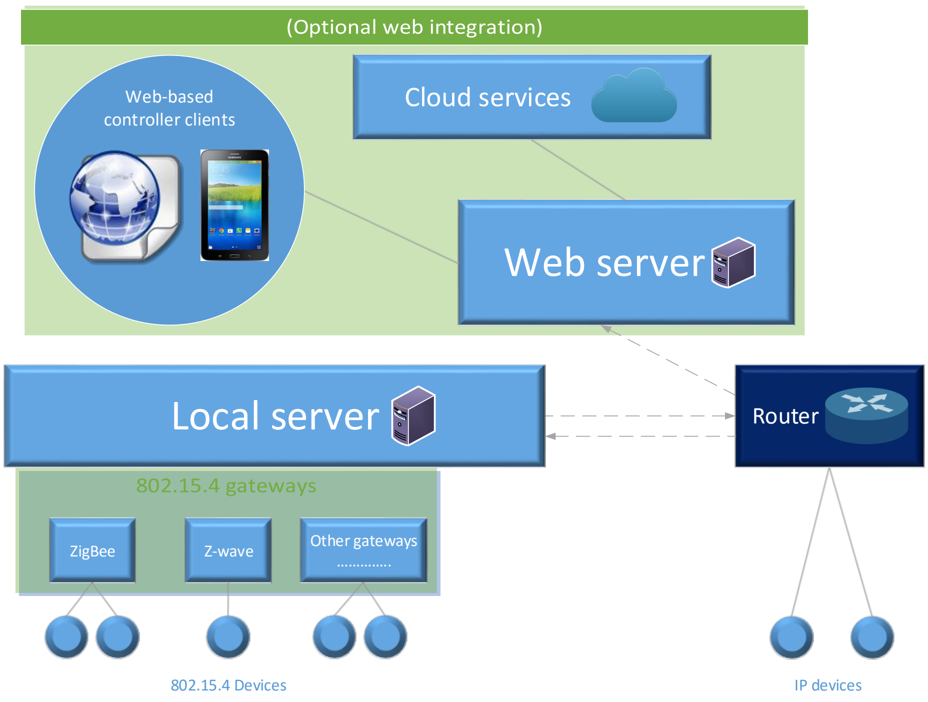


Figure 2.2: General smarthome architecture

A common thing for any smart house, or for that matter also when talking about the Internet of Things (IOT), are sensors and actuators which we under one name call devices. A sensor is a term for any device that measures some element e.g a motion sensor, a light sensor, heat sensor (thermometer), water-level sensor, smoke sensor, contact sensor etc. An actuator is a term for any device that does something depending on what state it is in e.g a lamp can be on or off, a heating element might have 5 levels of heat it can supply or a power outlet might be turned on or off. Devices can use a variety of protocols to communicate their data with, but common for almost all of them (the exception are IP devices) is that they need some gateway to pick up their radio signals. Whether this gateway is a ZigBee, Z-wave, Bluetooth Low Energy (BLE) or another gateway, they all play the same role of picking up the signals sent from their corresponding devices. Most gateways nowadays are small and often built-in to the local servers, but they often also come as USB dongles that you simply plug in to a computer and it instantly acts as a gateway. As mentioned IP devices are the exception because they don't need a specific gateway, rather they simply use a router to pick up their signals. So actually IP devices in some way do need a gateway to pickup their signals, but they are able to use the already existing infrastructure i.e the router installed in most modern houses nowadays.

The local server (also often called Hub or Home Gateway or Logical Controller etc) usually consists of a number of gateways used to communicate with sub networks, as well as a software module running some software (the house plan or the configuration), to control the house. It is responsible for doing the thinking in the system i.e making logical decisions based on the information from sensors, state of actuators and the stored configuration on the server. What the local server does, depends on what software is running on it, which is different from company to company. A simple use case could be that it turns on lamps **A** and **B** when motion sensor **C** detects movement, and off again when no movement has been detected for a while. A more advanced scenario would be if we only wanted to turn on those two lamps if motion sensor sensor **D** has not detected any motion for the last 2 minutes, and only if actuator **E** is not in a certain state. This can make up entire scenarios triggered by certain conditions, and even scenarios triggering other scenarios based on other conditions. Companies such as SmartThings and HomeSeer have built their own custom programming languages, that customers can program their houses with (only for the products supported by those companies), while projects like AllJoyn and HOMER and Google Weave offer open-source platforms for this purpose.

5 years ago most homes worked with just the above described components i.e a local server running the house configuration,a gateway for each protocol and the devices connected to the gateways. However, with the increasing usage of smart phones, a web integration has become important since users want to control and monitor their house from the smart phones. Furthermore the local server can also use information from the smart phone (for example its location), essentially making the smart phone an extra "sensor" providing input information for the local server. This has almost become the standard nowadays and most company solutions include features to control and monitor your house with your smart phone and with other web platforms.

House plan/Configuration - a house plan or a configuration is a piece of software consisting of two parts. One of the parts is the software device entities and all their associated data. Each physical device in the house, has a matching device entity in the house plan, with all the associated data attributes for that device e.g `id`, `type`, `protocol`, `room`. The other part of a house plan is the interaction between all those devices; for example specifying that when sensor **A** is triggered then actuator **B** should go to state `On`.

2.5 What is HOMER

HOMER (Home event recognition system) is an application layer system as shown in figure 2.1. It is developed for the OSGI specifications and runs in Apache Karaf which is an OSGI implementation. Apache Karaf is an OSGI container where bundles can interact with each other, by declaring which packages they export and import. A bundle is simply a jar file that contains extra manifest files needed by the OSGI container. A bundle also contains a Blueprint file that the OSGI container uses to manage the lifecycles of bundles and to wire bundles together e.g. if bundle **A** exposes a service and bundle **B** uses this service then this is specified in the Blueprint file and the OSGI framework then uses the Blueprint file to wire these two bundles. Bundles can be loaded and unloaded at run time without stopping the underlying OSGI container.

HOMER is an open source system supporting a variety of established protocols, by uniformly mapping devices according to the 11073 ISO standards. All devices that are already ISO 11073 compliant can be directly connected to a system running HOMER as shown in figure 2.4, while other devices need to be mapped first. Each supported protocol has its own developed bundle that if loaded will map devices speaking that protocol to HOMER using the ISO 11073 mapping. In this way the technician specifies which protocols need to be supported and hence configures HOMER to load the according mapping bundles. Figure 2.3 shows each protocol having its own bundle for the mapping.

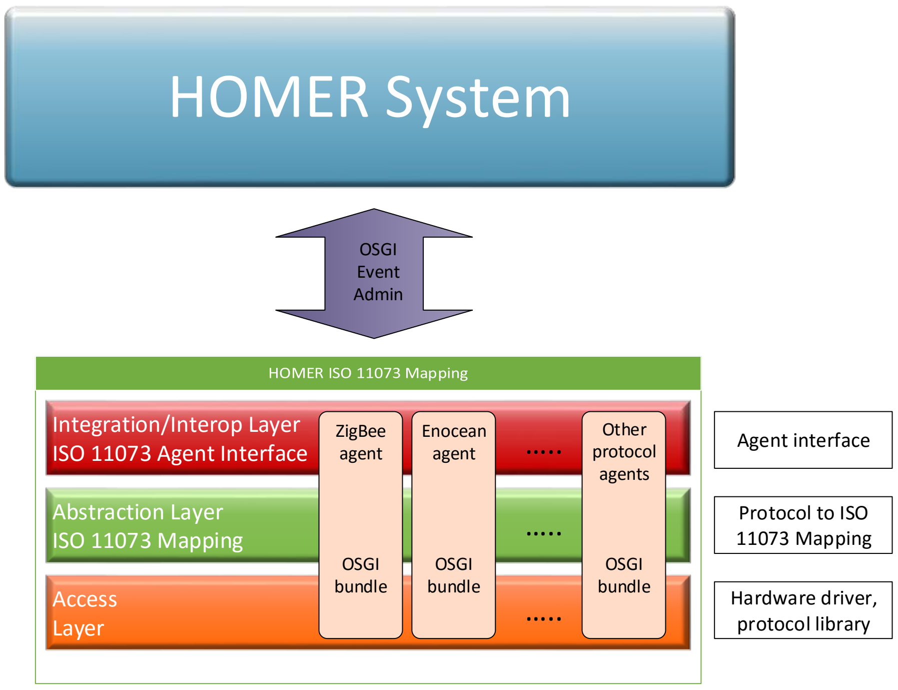


Figure 2.3: How HOMER maps the various protocols to the rest of the HOMER system

HOMER is an open source platform and therefore includes a bunch of services meant to help other developers extend it, such as easy database access, temporary data storage, scheduled bundle execution, templates for setting up new bundle services and much more. It is made to be extended. It includes a rudimentary GUI for users to create and manage their house plans with.

HOMER consists of a collection of many bundles, that all run simultaneously in Apache Karaf. Each instance of HOMER running in a home will at least have the HOMER Core bundles running in Apache Karaf, plus a bundle for each protocol used on devices in the house. However depending on what special needs are needed by the bundles created by other developers, additional bundles can be loaded that offer other services, and over the years since HOMER started many extra bundle services have become part of the HOMER core bundles.

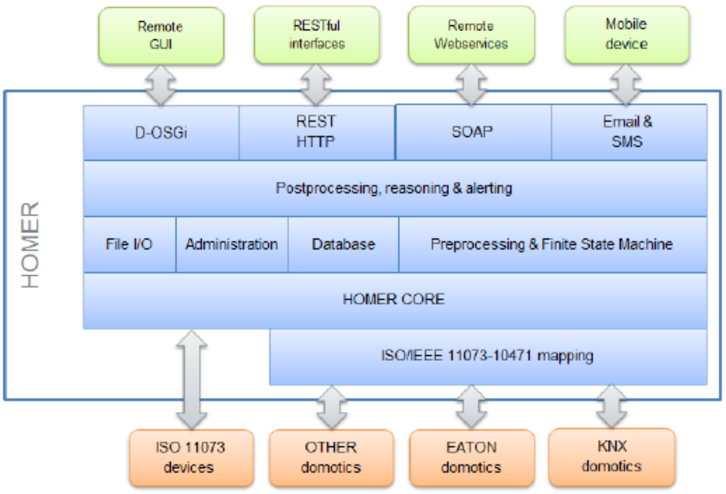


Figure 2.4: HOMER Architecture (from AIT's HOMER manual)

Figure 2.4 shows some of the features that HOMER offers, including a database and an implemented DAO for it, File I/O logical reasoning (pre-implemented scenarios) etc. AIT is a non-profit research institute funded by the Austrian government and therefore all work produced here is publicly available. Companies can chose to use the HOMER platform as it is and extend it to their needs or build their own system inspired by the architecture of HOMER. It was made by AIT to show companies how an application layer for home automation can look like. HOMER is currently running in a number of houses as test projects to gather data about the performance of it.

Chapter 3

Analysis

3.1 Detailed problem definition

The problem for AIT is that, it's currently too expensive and cumbersome for them, to set up new smart houses. The local server running in each house cannot be accessed remotely, making it necessary for a technician to be on-site to setup or reconfigure the software running in it. So the basic problem is that we have a piece of well developed software (in this case HOMER, but it could have been other systems as well), but with no way of remotely installing or configuring it in peoples homes. The work flow for a company-technician when setting up a new house currently involves the following steps:

1. While sitting at company offices, the technician creates a house plan for all the devices in the house and their interactions. This is maybe done having incomplete knowledge of the precise house layout, but is still necessary in order to try to minimize the amount of time spent in peoples homes.
2. Drive to the house where the installation is to take place.
3. Physically place all devices where they belong.
4. Set up the local server consisting of the gateway(s) to the various devices in the house, as well as the software component running the house plan.

5. Compile and deploy the configured house plan on the local server (initial configuration can be pre-deployed on the local server).

6. If/when things don't work as intended, then modify the house plan while still being in the inhabitants house and recompile and redeploy it to the local server.

The biggest problem is that AIT currently has no software that allows the technician to remotely deploy a house plan a local server. It is possible to create a house plan before going to the house, but you have to be on-site to deploy the software to the local server. This can become a problem if something does not work as intended when he is setting up the house. A sensor might need to be moved because it cannot transmit properly from it's location, or while setting up the house, the technician and the inhabitants realize that something could have been done in a smarter way etc. This is not very flexible or dynamical solution and it means that the technician will have to spend quite some time in peoples homes, since he needs to ensure that the configuration works 100 % as intended before leaving the house.

3.2 Advantages and disadvantages of a local server

HOMER is developed to be running on a local server in each house, which implies that an actual server is installed in each house. Before we accept this solution we should ask the questions: Is a local server needed ? How little on-site soft/hardware could we get away with? To answer these questions we need to consider which devices are installed in the house. A few devices are native TCP/IP-speaking devices, meaning they have their own embedded mini web server in them. These devices could communicate directly with a central web server meaning that no extra hard- or software at all would be needed in the house. However, most devices currently on the market do not include an embedded mini web server, but instead communicate over 802.15.4 networks. These networks do not use TCP/IP, but rather need a gateway to pick up the radio signals sent from them. If an ultra lightweight solution was desired for devices running over these networks, you could chose to have the gateway forward the signals it receive directly to a central server. The gateway would need to be equipped with a mini web server to do this, but this is affordable. In this way we would have all TCP/IP-speaking devices in the house communicating directly with a central server, while all 802.15.4-speaking devices would communicate through the gateway directly to the central server. This architecture allows for very easy maintenance of the system since no software is deployed locally.

Inhabitants and/or technicians can simply log on to a central web server and modify the software running in that specific house.

The downside to this solution lies in how expensive TCP/IP communication is compared to low rate protocols (802.15.4), since with this architecture all communication needs to go over the internet. So every time you turn on the light, or a motion sensor triggers or basically anything happens in your house, the data needs to be sent over the internet. This will create a lot of traffic. Devices with embedded mini servers on them would either need to be wired or have *very* good/large batteries in them. However, as we saw earlier some articles argue that the embedding of mini web servers communicating over TCP/IP is not as power consuming as one might think [DGV09]. And with the recent development of low power Wi-Fi protocols, such as ZigBee IP [Zig15], as well as 6LoWPAN allowing efficient IPv6 communication over 802.15.4 networks, embedded mini web servers in every device might not be a bad idea for the future.

But since these new low power protocols have not yet become widely adopted and because very few devices currently on the market come equipped with embedded web servers in them, AIT's solution (and the solution used by most vendors) with a local server in each house makes sense. The main advantage to a local server is that it saves a lot of power. Devices communicating over 802.15.4 and other low rate protocols, can send their signals to the gateway, which simply forwards it to the local server - no TCP/IP communication needed. TCP/IP embedded devices can communicate directly with the local server, which still takes up considerably more power than low rate protocols, but is better than communicating with a central server. The downside to this solution, is that we cannot modify the house plan in each house as easily, since it is now located on the local server and not on the central server. Therefore some means of communicating with the local server is needed.

3.3 Achieving remote access to the local server

There are several means to achieve remote access to a local server behind a firewall. Port forwarding is a technique that allows computers outside the local area network, to pass data to a specific computer or service within the network. This is done by creating rules on your local router, configuring it to listen to inbound traffic on certain ports, and then forwarding all incoming traffic on these ports to internal services on the local network. Port forwarding is very insecure because no encryption is in place, but also because hackers have port scanning available as a tool to them, where they can find the open port on the firewall and use the channel in a malicious way. Until 2014 port forwarding was used in all homes using the nr. 1 smart-home vendor on the market [Car15]: HomeSeer (and is still used in many houses today). In 2014 they released their optional Remote Access Service [Hom13], offering customers to sync their house plan to a cloud and then access the cloud with their mobile devices. 4Control, Another large Smart Home, is also strongly against port forwarding and warned in 2013 their users against using this technique [4Co13]. Other large vendors such as SmartThings and Apple HomeKit have also opted not to use port forwarding.

A way more secure alternative to port forwarding is using Virtual Private Networks wherein the client downloads a VPN client program that securely sets up a connection with a remote server behind a firewall. It provides several layers of security, by both encrypting datagrams but also sending them through a so called "tunnel". A popular packet encryption method for VPN's is the IPSec protocol, which is often used in conjunction with a tunnelling protocol e.g L2TP. VPN's offer a lot of security but can impose quite some overhead slowing traffic down. It is however still a good viable option for remote access.

3.3.1 Moving the architecture online

Instead of using either of the remote access techniques, you could simply chose to move the entire architecture online. You could store all devices and device-related information in an online database and avoid using any kind of remote access to a local server. We would then store house plans for **all** houses in the system on a central web server and to facilitate this we would need to build an information exchange architecture, allowing communication between local servers running HOMER and the central server. A synchronization mechanism between the local and the web server would also be needed, to ensure that the users are always viewing a correct version of the houseplans on their mobile phones or browsers.

For software developers, the benefits of an online architecture would be huge, since you no longer need to deal with information begin stored on many distributed servers, but instead in one big database. Having devices stored in an online database, enables you to put a tag (NFC, QR, Bar code) on each device linking that device to an entry in the online database. Private users can then simply scan the device, and the underlying information needed to configure that device will simply be fetched from the database where after the system automatically configures the device, all without the user having to know anything about protocols, device types or anything. For software developers this architecture helps immensely because it is a flexible and extendible solution, making it much easier for software developers to extend the system, much easier to maintain the system and much easier to plan for future unknown use cases. Backwards compatibility also becomes easier; for example if you want to change the way a device is stored in the database, or the information linked to a certain type of devices, or implement new services to be used by other developers. Another benefit is that we don't need to access any data behind any firewall. Client controllers (mobile APP and browsers etc.) can simply access the house plans directly on the web server. There would also be no need for customers to configure their firewall for port forwarding or setting up a VPN. Devices can also be authenticated with their tag. One of the disadvantages of the solution, is that a secure means of storing and accessing house data, becomes even more important when storing all data on an online cloud server. If we chose to use local servers and an online cloud architecture, then we will also need to build some means of exchanging data between the local servers and the cloud i.e a synchronization mechanism. This communication should also be secured using for example SSL/TLS.

Choice of tag technology NFC and QR codes were considered. Bar codes is an old technology that is not capable of storing very much information, and which is inferior to QR codes. QR Codes have the big advantage that it is a completely free technology to use, while NFC tags costs around 0,10 to 0,80 dollars pr. chip if you buy them in bulks (min. 500). The price depends on the desired storage capacity. Anyone can print out a QR tag (and in many different sizes) and place it wherever they want to, while NFC tags need to be shielded from rain and weather since it is an actual chip. Depending on how much error correction is desired in the QR code, it can store somewhere between 0,5 kb and 3 kb of data, while NFC tags on the market currently range between 64 bytes for the smallest ones and up to 888 byte for the newest NFC tag on the market; NTAG216. Any smart phone equipped with a camera is capable of reading a qr code, provided they have downloaded the app for it, while reading NFC tags requires new hardware not supported by the majority of smart phones currently on the market. The major advantage with NFC tags is the scan speed. Simply holding a scanner near a tag will almost instantly scan it, while the scanning process with QR codes can be a bit cumbersome holding the camera at the exact right angle etc. Furthermore NFC tags can be placed "under the hood", meaning they don't need to be visible on the product, while QR codes obviously need to be visible for the camera to read. Last of all, most NFC tags include built-in encryption, while QR codes offer no security and actually can be tampered with, to cause malicious behaviour. Ultimately QR tags were chosen, because they are free to use, easy to implement and is already a widely used technology.

3.4 Synchronization

A consequence of using local servers is that it necessitates a synchronization mechanism, since we will have a distributed system where data is stored in different places; at the central server and at the local servers. Whenever any changes happen to data on the central server, those changes need to be reflected on the local server in the house linked to the affected data. Synchronization is meant to happen in one direction only i.e from the central server to the local server. This is because the system should be a closed system, where creation of devices only happen through a web page and modification of devices only happen through the app. Therefore no synchronization should be performed in the other direction, meaning that the local server should always take the house plan from the central server and overwrite the locally stored one ensuring only one entry point for data, making the system easier to maintain and supervise. However AIT does not yet use this system and therefore we should ensure that the system is also backwards compatible with the work flow currently being used - a workflow that involves deploying the local servers with the house plans on them pre-installed. Therefore synchronization should be performed in both directions; from the central server to the local server, but also in the other direction. This means that if the local server has a device unknown by the central server, the central server will create that device and link it to that house.

A problem we are facing when synchronizing from the central server to the local server, is that we can't communicate directly with the local server, since it is located behind a firewall. Generally the firewall will prevent incoming packets, unless it can identify them as the response to an outgoing packet, or a special rule has been created e.g port forwarding (which we do not want). Therefore the central server cannot just notify the local server whenever changes happen. Instead the local server needs to regularly "check in" with the central server, to see if any devices have been added, deleted or edited. But one might ask how applications such as Skype then initiate a call, since the client receiving the call certainly hasn't requested this call. This has to do with a technique called "UDP Hole Punching" [Sch07]. Alice wants to send data directly to bob (and receive), but this is restricted by Bob's firewall. Instead Alice tells the Skype server (which Bob trusts) that she wants to talk with Bob. The Skype server forwards the message to Bob, upon which Bob "punches" a temporary hole in his own firewall for Alice to use and similarly the other way around. So essentially Alice uses a 3rd party (Skype) to authorize herself with Bob. This is a very rough simplification, but it shows us that communication initiated from outside a firewall is possible. And indeed this is what danish company Nabto has developed - A protocol that allows direct outside communication to a device behind a firewall.

Such a firewall punching technology would be very well suited for smart homes since it basically allows local servers to act as clients subscribing to changes made to devices linked to them. This would mean that the central server could notify the entire system (webpage, app and local server), whenever a change was made from a controller client. However, this system is patented by Nabto and is not free. Neither does AIT have their own implementation of it or anything similar. Therefore we have chosen the alternative, which is that all communication is initiated by the local server, which means that it regularly requests a synchronization with the central server, to see if there are any changes to the house plan.

3.5 Requirements

The main requirement for the system said in words is: "To ease the installation and setup of new smart houses for technicians", and as discussed the problem was the lack of remotely being able to modify data on HOMER. The system should enable the technician to deploy a house plan on a local server remotely, but also enable him to modify an existing house plan remotely. This spawned a discussion about how best we can achieve remote access to the local server, which led us to take the decision of using an online architecture, instead of dealing with port forwarding or VPN's. A few weeks into the project we decided that including the interaction part of house plans was too big for this project, and instead we chose to focus entirely on implementing an extendible structured online architecture for storing, adding, deleting and editing devices - the interaction part would be left for another project. The basic functionalities of the system would concern devices i.e adding, deleting, editing, linking and unlinking devices. We set out building the system for the technician as a tool to aid him when installing new smart houses, but along the way we realized that we wanted the system to be implemented in a way, so that it in a later project could be expanded, so private non-tech users (customers) can use the system themselves to install their own houses. Therefore we wanted to spread the functionality over 2 client controllers in the system, both of them for technicians, but one of them implemented to be adaptable in the future for customers only. A simple webpage should be implemented for the company to manage all their devices with, allowing technicians/admins to add, delete and edit devices in the system. An app should be implemented allowing technicians to link and unlink existing devices in the database with houses. To give the technician more control over the system allowing quick testing of new device setups, the add device and edit functions should also be available on the app as well as on the webpage. To facilitate communication with the HOMER platform an OSGI bundle should be developed to be deployed on HOMER.

3.5.1 Users

The overall users of the system are companies wanting to develop and sell smart home products. Below we will describe what users of such a system value..

Software Developer

Developers of the system will focus on how the features of the system are built rather than the actual features themselves, since it is them who are tasked with understanding, maintaining and extending the system. They prioritize a system with a good architecture, meaning a server with a clearly defined interface, that can easily be extended with new functions. A modularized system, where services, calls or components are reusable and a system where logics are grouped and placed in a structured and understandable manner. In addition to that the code should be easy to compile, build and test.

Technician

A technician cares mostly about the amount of features offered by the app, and that the app is working and stable. Usability of the app also means a bit to a technician since he doesn't want to waste his time with the app, while setting up a new smart home. The time it takes for data to be transferred from the app to the local server should also not be too long.

3.5.2 Requirements & quality parameters

The system will require 4 components to be developed. A central web server, an application for a mobile phone, a webpage and a software bundle for the HOMER platform. The software-centric qualities touched upon under the software developer user description, are here relevant for all parts of the system, since no matter which part we're considering, a software developer will appreciate good architecture when he's working with the system. So the software quality will not be commented upon under each section, though it is important for all components of the system.

Server

- It should be able to store devices in a database.
- It should provide clients access to devices in a usable manner i.e provide proper services to get devices.
- It should provide clients operations to: add, delete, edit, link and unlink devices.
- It should provide a library of common objects for clients (and itself) to use.

The server should be able to store large amounts of data as efficiently as possible as well as having a reasonable data access speed. In our case a fast access speed relies more on the I/O performance of the database, rather than a high transfer speed, since we're dealing with small chunks of data.

App

- It should enable a technician to log in to a house.
- It should present the devices on the central server associated with the house logged into, in a clear presentable manner.
- It should remain synchronized with the central server.
- It should enable users to link existing devices on the central server, to the house currently logged into, by scanning a QR-tag.
- It should enable users to unlink a device linked to the house currently logged into.
- It should enable technicians to quickly test new device setups, by manually adding new devices to the system and editing them.

A smooth user experience is important for the app, since it is trying to save the technician time. Users value that the app works as intended, does not crash/freeze and is easy to use. A nice UI giving a nice overview of the presented devices is valued. Besides from this, since the app is for technicians, it should be an accurate reflection of the system state and what is currently going on in the system, meaning that it should be synced with the devices on the webserver and it should show the technician the "state" of a device being synchronized. By this it is meant whether the device has been synchronized to HOMER yet or not.

Webpage

- It should present all device on the central server to the user.
- For each device it should display the corresponding QR tag linked to that device.
- It should remain synchronized with the central server.
- It should allow a technician to delete, edit and add devices to the system

The web page does not have so strict requirements for the user experience, since it is not for private users. Here it is more important that it works as intended and is robust. The amount of features offered is also somewhat important here, since admins and technicians need to have additional rights to manage the system.

HOMER bundle

- It should keep the devices on HOMER synchronized with the devices on the central server.
- Synchronization should happen in both directions i.e from central server to HOMER and from HOMER to central server.

The HOMER bundle should be stable and never crash even though it runs in the background all the time. It is important that it syncs correctly without losing any data during the sync process. The frequency at which synchronization takes place is important - we want it to happen often, but not too often causing unnecessary traffic. The synchronization should also not be too slow.

Besides from the 4 described components described above, the system requires that each device has a QR tag on it that stores the hardware id of that device.

CHAPTER 4

Design

This chapter will describe the architecture of the entire system, as well as design of each of the 4 components, their subcomponents and how they interact with each other. The focus is on the high level architecture and not code specific details.

4.1 Overall system architecture

The system is divided into four parts as can be seen in figure 4.1; the central server, the local server(s) running HOMER, an app and a webpage. The central server stores and manages all devices in the system in its JPA database, meaning that this database stores all devices for all houses in the system. Each house has a local server in it running an instance of HOMER. Local servers, mobile clients and browsers all communicate with the central server, by consuming the REST client API implemented by the central server.

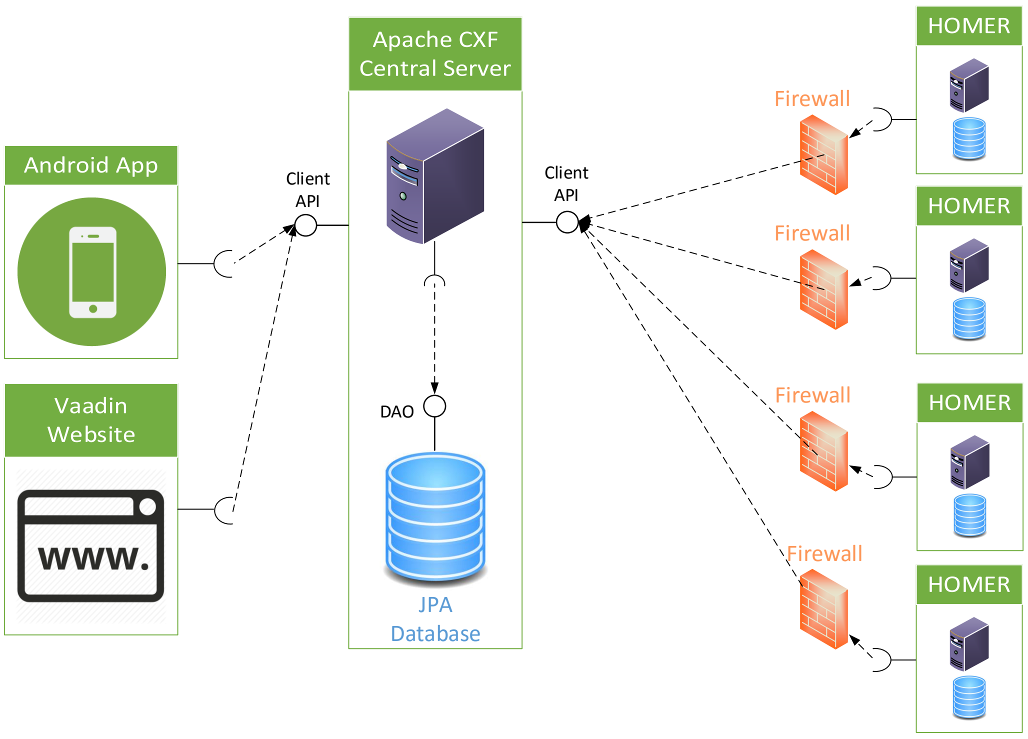


Figure 4.1: Overall system architecture

4.2 Central server

The central server was designed with a service oriented architecture in mind, relying heavily on the REST principles. All calls are either POST, GET, PUT or DELETE calls affecting 1 or more resources. The server is completely stateless, meaning that no client context is ever stored on the server and all information needed to execute a requested service is provided by the client. Furthermore the server and client are generally at all times sought to be completely separated having nothing to do with each other. The only link between them should be a uniform interface provided by the server that the client implements, as well as the common

This architecture is very modular and reusable achieving a very lose coupling in the system on two levels. One of them is between clients and the server. A client can call server commands independently of one another, and combine the results of the calls to do whatever the client wishes to do. This can be done synchronously and asynchronously depending on the clients needs. The second level exists between the implementation of the service interface and the device-service-DAO, where a service implementation can combine the results of any DAO calls (all sequentially executed), to obtain the desired result. This architecture decouples the server and the client so development can happen on both simultaneously and independent of each other. And within the server, development can happen simultaneously and independently of one another on the service implementation and the device-service-DAO.

4.2.1 Client API & server interface

The server and the clients are connected only through a uniform REST interface, making the methods defined in the client API the only means for clients to interact with the server. The methods defined by the client API, are implemented on the server through the APIService interface. This nicely encapsulates the server by defining the entry points into the code and is a good "first line of defence" against unwanted behaviour. The binding link between the client API and the APIService interface are the Universal Rsource Identifier's(URI's), which should be identical in the client API and the APIService interface. Each service call has one and it could look something like this: *"/devices/get/flatid"*, belonging to the a service call that will fetch all devices from the server with the provided flatid. In addition to this each service call also defines a data return type, that the client can rely on being returned, no matter how the call is initiated from the client. These return types should be used by all clients in the system (an by the server itself) and therefore they are stored in the commons library on the server. Service calls are defined both as synchronous and asynchronous, but clients should keep in mind that calling a service call synchronously locks the calling thread, which gives a bad user experience. Therefore the app never makes any calls synchronous, it is mainly the HOMER bundle that uses synchronous calls when executing the sync procedure. Optionally the service calls can also specify which parameters they need, which are also clearly defined entities defined in the commons library e.g. the call `editDevice(EditRequestWrapper device)` requires an `EditRequestWrapper` as a parameter, which is a common entity defined in the commons library.

4.2.2 Database

The database is a MySQL database created and managed by Java Persistence API (JPA) and run over an Apache server. It is a simple database, with just one table for all devices. There are a few extra tables for users, their roles and their permissions, but they were not created for this project, nor are they the focus of this project. The database is accessed through a DAO interface called *DeviceService*, offering the central server (which in relation to the database acts as a client) a set of methods defined in the DAO. The advantage of using a DAO is that we can offer a specific set of data operations to the user, without exposing the details of the database. Secondly we can also reuse the DAO if other components are introduced to the system that need database access. To help maintain the code, logics are kept as much as possible out of the DAO implementation, and are instead implemented in the server interface implementation. This is to have the logic gathered as much as possible at the same place in the system, helping software developers quickly get an overview of the code.

4.2.3 Commons library

The commons library contains common data structures for the entire system, along with a few utility classes used for parsing JSON objects to/from the server. Clients don't need to implement a JSON parser, because the client API is defined with Retrofit that does it for you. The common data structures mainly include response objects for the server to use when returning data to the clients, essentially acting as data contracts between the clients and the server. Besides from the response objects, the entities stored in the JPA database are also defined here; Device, User, Role, Permission (only Device is important for this project). What is very nice about having a commons library where the Device class is defined, is that all service and DAO methods don't need to know the contents of the Device class. They can simply specify the Device class as their parameter and/or return object and ignore what the Device contains, when passing the device object around in the system.

4.2.4 Device entity

The device class stores all information on a device and can be seen in figure 4.2. A device has a hardware id, which is a unique id that only that specific device can have. Whether or not the uniqueness of this hardware id will extend further than each vendors collection depends on the implementation of the system, but this id is presumed to be the unique id of a device in this system. This id should also be stored in a tag on the device, to link the physical device with its entry in the online database.

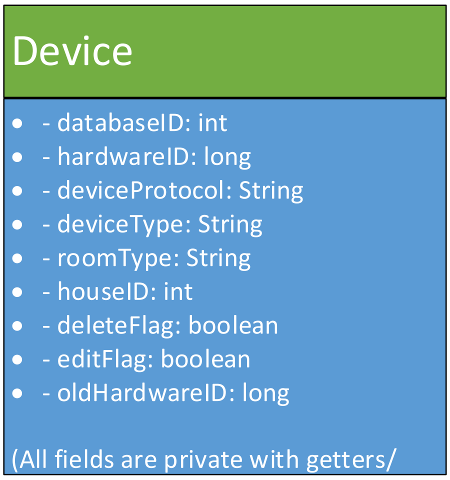


Figure 4.2: Device Entity

Besides from the hardware id, other mandatory fields that must be set when a device is created, are the device type and the device protocol. Since HOMER is developed with the ISO 11073 standards for devices, we have also used this for the device types in this system. The device type should therefore be specified according to this standard; an example of such a type is the motion sensor type: *MDC_AI_TYPE_SENSOR_MOTION*. The device protocol is also mandatory and should be specified according to the currently 12 different protocols supported by HOMER. The *deviceProtocol*, *deviceType* and *roomType* have been implemented with string types as can be seen on figure 4.2. In hindsight it would have been cleaner and better, to implement them as enumeration types matching the specified enumeration types on HOMER, but issues importing the library from the HOMER Core bundles, prevented us from doing this. The 3 properties; hardwareId, deviceType and protocol are mandatory properties and must be provided when creating an instance of the Device class. The last mandatory attribute is the roomid, which is included because the HOMER database requires it and because the app uses it to group devices under the rooms they are located in. Besides from these mandatory fields each device has a flatid (not mandatory upon ceation), linking that device to a certain flat, along with some flags and pointers used for synchronization which will be described in detail in section 4.6.1.

4.3 HOMER bundle

The purpose of the developed HOMER bundle is to facilitate data communication between the central server and HOMER, by keeping devices on HOMER synchronized with devices on the central server (and the other way around, due to backwards compatibilty as discussed in the analysis section 3.4. The bundle is deployed along with the HOMER core bundles on the local server.

4.3.1 Synchronization scheduler

The HOMER bundle is responsible for the synchronization between HOMER and the central server, since due to local firewalls, communication cannot be initiated in the other direction as discussed in 3.4. So HOMER is responsible for synchronizing itself with the server, not the other way around. Therefore the bundle makes use of a scheduler function from the HOMER Core framework, that makes the bundle runnable and executes its run() method at a regular interval. The entire synchronization flow is executed inside this run() method on every execution. To achieve more modularity in the system and to ease debugging of the synchronization mechanism, the synchronization code has been split up into several methods. These include: *getDevicesWithFlatID-FromServer(), deleteDevicesFromLocalDBMarkedForDeletion(), editOrAddDevicesFromLocalDBMarkedForEditing(), loadDevicesFromLocalDB(), addSyncedList-ToWebserver(), webserverCleanUp(flatid)* and together they make up the entire synchronization process executed in every run(). They will be described in detail in section 4.6. The scheduler timer (or synchronization timer) is set to execute the run() method every 5 minutes. However, if the app is checked into the house it most likely means that the house plan is being modified, and therefore we set a flag on the server, indicating that the app is being used. When the run() method executes next time, it will check whether this flag has been set and if it has, the synchronization timer will be set to 10 seconds for as long as the app is in use. This is to facilitate more frequent synchronization whenever a technician is modifying the house plan. The full synchronization procedure will be described in detail in section 4.6 and can be seen on figure on figure 4.7.

4.3.2 Device conversion

Some device conversion is sadly needed in the HOMER bundle, since the HOMER database uses a different kind of device type than the rest of the system. The original intention was to use the same class for defining devices in the entire system i.e HOMER and server would use the same common device class for this. However, the HOMER Core system currently uses two different objects for devices: **DBSensor** and **DBActuator**. The system is undergoing changes that are going to merge these two into one entity simply called DBDevice. Ideally our defined Device entity should have been identical to this DBDevice class, that will be implemented by HOMER in the future, in order to avoid any device conversion. But since these changes are not part of HOMER Core yet, we have defined a our own Device class as described earlier, and a device converter class has been implemented that converts between our Device entity and the **DBSensor** and **DBActuator** classes used by HOMER. Note that this device conversion is only needed when storing a device from the central server on HOMER, or when sending a device from HOMER back to the central server. The central server and all clients work with our defined Device class.

4.4 Mobile Android Application

The app is one of the two clients in the system, and its main responsibility is to facilitate linking and unlinking devices on the central server with individual houses. Currently the intended usage is for company technicians to use it as an aid when setting up new houses. Once the technician has checked into a house the app will fetch all devices from the central server with the given house id and the technician will be shown a list of those devices. Through the graphical user interface, the technician can link, edit and unlink devices on the app and these changes will be reflected on the central server and in the visual device list on the app thereafter.

The app starts up in the checkinActivity, asking the user to check in to a house. This is currently done by simply providing the house id of the desired house. Originally we had an idea about installing a scannable tag in each house that would allow the technician to check in to an apartment by scanning the tag. But although this was implemented in the app, the idea has since been scraped in favor of a user/password combination used to log into a house (although this has not been implemented). The thought behind this, is that if remote access is wanted, then a tag in the house will not help and hence one might as well use a user/pass combination all the time instead. This also supports future development, because in the future when the system has been further developed and made so user friendly that private users can install their own houses, then they will log into their own house with a user/password combination.

Once the user has checked into a house, the app session will be bound to that house, meaning that he will only be shown devices linked to that house and all actions he perform in the app will be done on that house. After checking in, the user will be taken to the main screen shown later in figure 4.3.

Before discussing the communication between the app and central server, we would like to briefly address why devices are linked in the app and not created. This is because devices are added to the system unlinked through the webpage, and then later linked to houses through the app. For the system this separation makes no difference, since we could just as easily have put this functionality in the app and then cut the web page out of the project. But for future purposes the adding and linking of devices to the system has been separated, because as mentioned above, private users should be able to install their own house in the future and at that point the app should only be for private users, while the webpage will be meant for the company to manage everything with. In the future the app should also only allow linking devices through scanning, since this function better encapsulates the system preventing faulty user input. Editing devices and manually adding devices to the system, are also features that would be taken out of the app. They are only meant for technicians to have access to.

4.4.1 Server-app communication

The app implements the client API provided by the central server, namely it implements 5 methods from this interface: *getDevicesByFlatIDFromServer(flatid)*, *linkDevice(device)*, *markDeviceForUnlink(device)*, *editDevice(device)* and *setSyncTimerFlag(boolean)*. All communication between the app and server, is initiated from the app which means we have a fully passive server in this relationship, that only responds on the requests given by the app. Calls from the app to the server are always performed asynchronously, which is almost always recommended to do between a client with a UI and a server, in order to free up the thread running the UI on the client. Therefore callbacks are implemented on all calls from the app to the server and one should hence keep in mind when developing on the app, not to rely on calls being performed sequentially.

Synchronization on the app is done manually by pressing the refresh button found in the action bar on the app. Alternatively we could have put this refresh call inside an asynchronous thread running in the background executing every 10 seconds or so. But waiting for this background process to do the synchronization every time a change happens on the app, seemed like an annoying solution. Lowering the synchronization timer too much (say to 2-3 seconds) would imply too much unnecessary synchronization traffic, and therefore the manual approach was favoured. The best solution, however, would be to have the server inform the app whenever a change happens on it. This would require creating a background process on the app listening and communicating with the server, while the server would need to store session information on each app instance currently logged in to a house, in order to be able to notify the correct observers. Ultimately due to time constraints this was not implemented, but would be a nice alternative to manually refreshing.

4.4.2 User interface

The main menu screen can be seen in figure 4.3. The user interface was not the focus of this project, but nonetheless we felt that it was important to have a decent UI that could give a good user experience. The UI is divided into two sections; the buttons on the left side and the list of all the devices connected to the house on the right. The list is a 3-layered expandable list, made to quickly be able to gain an overview of all devices in the house, the information associated with device and the rooms the devices are placed in.

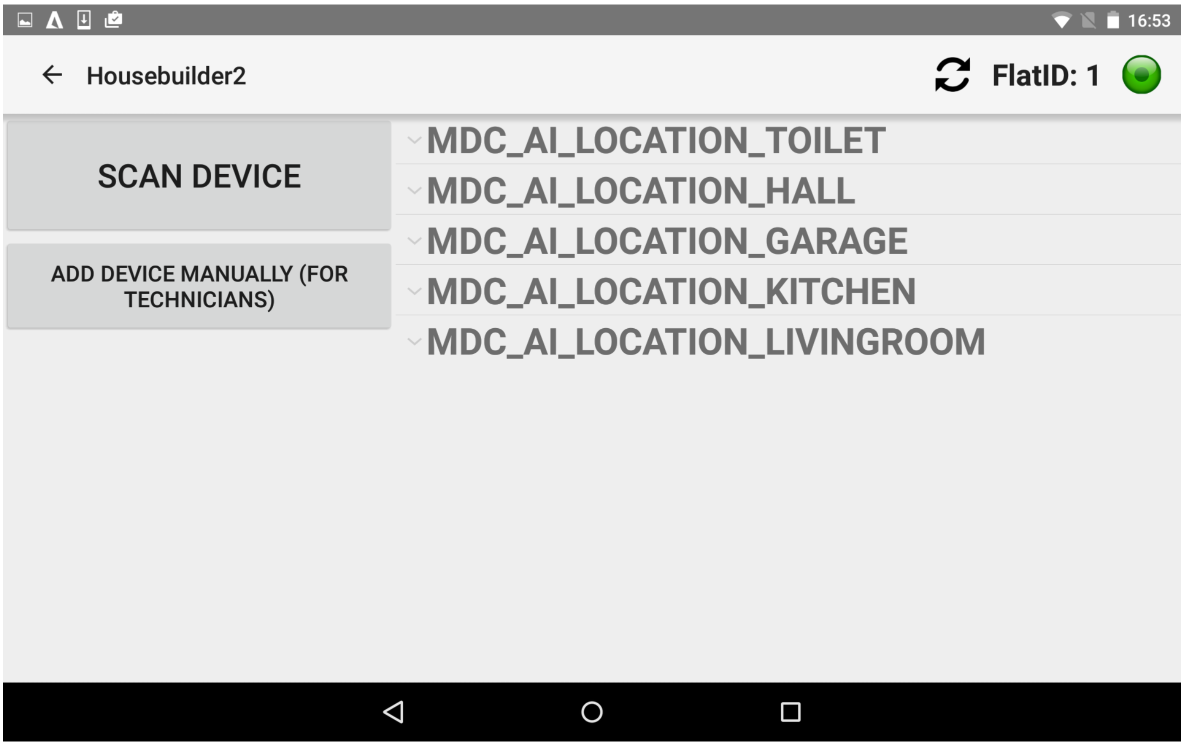
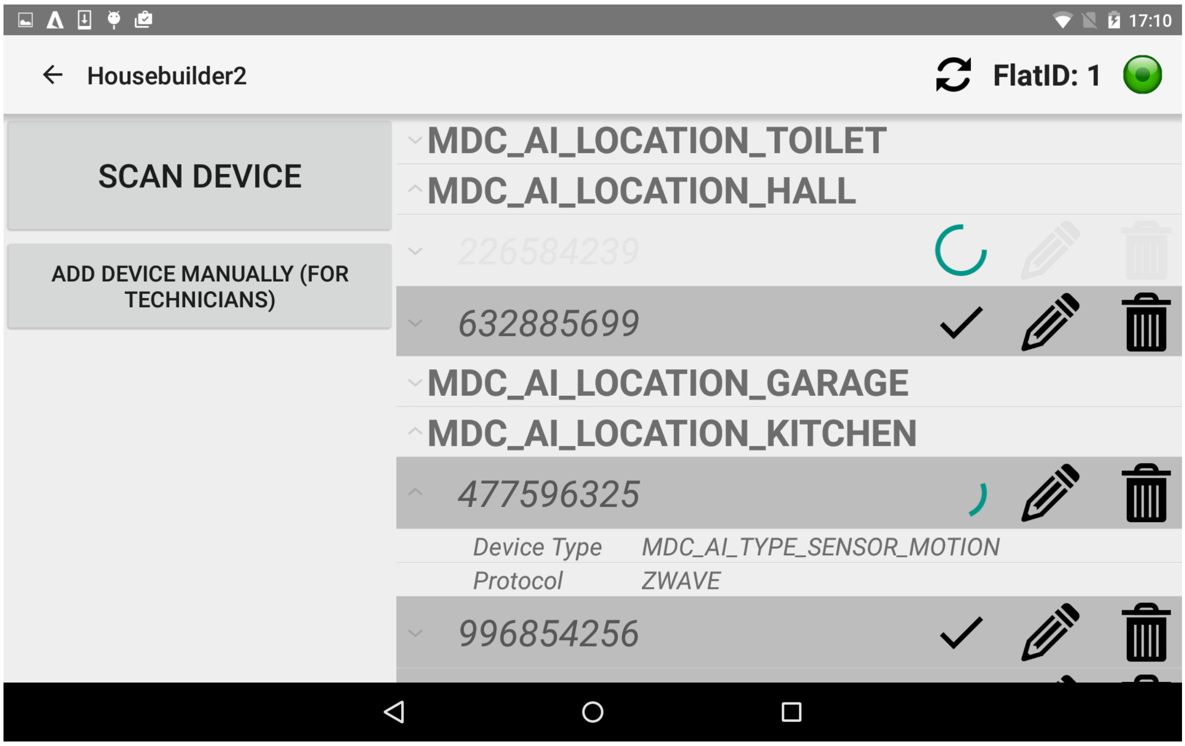


Figure 4.3: The main menu on the app

Currently the names of the rooms are just the according ISO 11073 standard names for rooms, while the devices are named after their hardwareID. They are named this way, with the assumption that the technician knows about these standards and will appreciate the standards. In the future when private users are going to use the app, the intention is to force the user to give a description of a device and a name of the room it is in, when scanning it. For example the device description could be "Table lamp" and the room name could be "Office". This will allow us to use this data in the UI, making it easier to recognize devices and the rooms they are in.

For the sake of simplicity and a good user experience, an extra effort was put into creating the list view on the right side of the screen. First of all we felt that a standard list view would be annoying to work with since androids basic list view is static and hence cannot be interacted with. Had we used this option, details on every device would have to be visible at all times (expanded all the time), which would make getting an overview very hard with many devices.

To solve this an expandable list which allows the user to expand and collapse details on individual devices was used. In this way the user can collapse devices that he does not wish to view the details on. Android has a built in function for expandable lists which is nice, but we wanted to further group devices under the rooms they belong to and make entire rooms expandable/collapseable, so that when the "Kitchen" is collapsed then all devices in the kitchen will no longer be shown. Furthermore with expandable lists the developer can programmatically control which devices are expanded/collapsed (for example we could expand devices that have just been edited, linked or added). This would mean, that we would need an expandable list containing rooms, with each room containing another expandable list of the devices present in it, essentially giving us a 3-layered expandable list as seen in figure 4.4. Level 1 contains rooms, level 2 the devices present in that room and level 3 the details on a device. 3-layered expandable lists is not a feature that is supported natively by android and therefore the layouts, xml-files, list-adapters and a cache had to be custom built which will be elaborated further on in the implementation section 5.4.1.



Device Type	MDC_AI_TYPE_SENSOR_MOTION
Protocol	ZWAVE

Figure 4.4: 3-layered expandable list

The device list is re-fetched from the server and redrawn whenever any call from the app to the server is carried out. The user can also manually request this with the refresh button in the action bar. For a technician when he links a device to a house through the app, it is important that he knows when the device has been synchronized to HOMER. The synchronization is something that happens in the background, but we want to visualize it. Therefore the delete- and edit flags, are checked whenever a redraw is initialized. If either flag is set, a spinning circle will appear on the device meaning it has not yet been synchronized to HOMER, additionally devices with their delete flag set will be greyed out and their buttons will be unclickable. Device 226584239 on figure 4.4 has its delete flag set, and therefore it will be drawn greyed out and with a spinning circle, while device 477596325 has its edit flag set and will therefore have a spinning circle on it. Devices with a tick on them are synchronized with HOMER.

Gathering data provided by the user is needed, when editing a device or adding a device to the system. For this, a subactivity has been implemented which acts as a device data form, where the user provides new data for a device he wants to add or edit (see figure 4.5). To minimize faulty user input, spinners have been used wherever the user should only be able to select an option from a given set of options, e.g in the case of device protocols and device types.

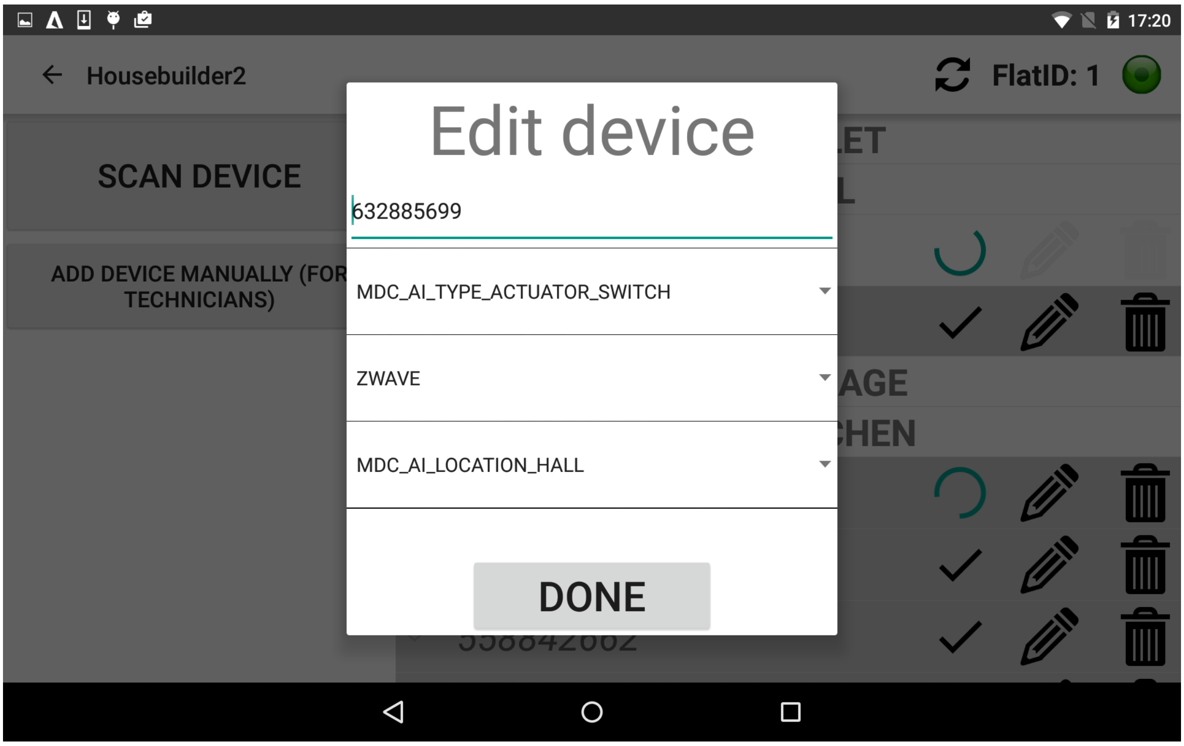


Figure 4.5: Device data form for editing and adding devices

In the case of editing a device, all data from the clicked device are passed through an intent to the edit subactivity. When the user is done entering device data, he taps the done button and the data is transferred to the main activity again through an intent. The main activity picks up the data from the intent through its *onActivityResult* method and then calls the appropriate server method (edit, add, link or unlink device).

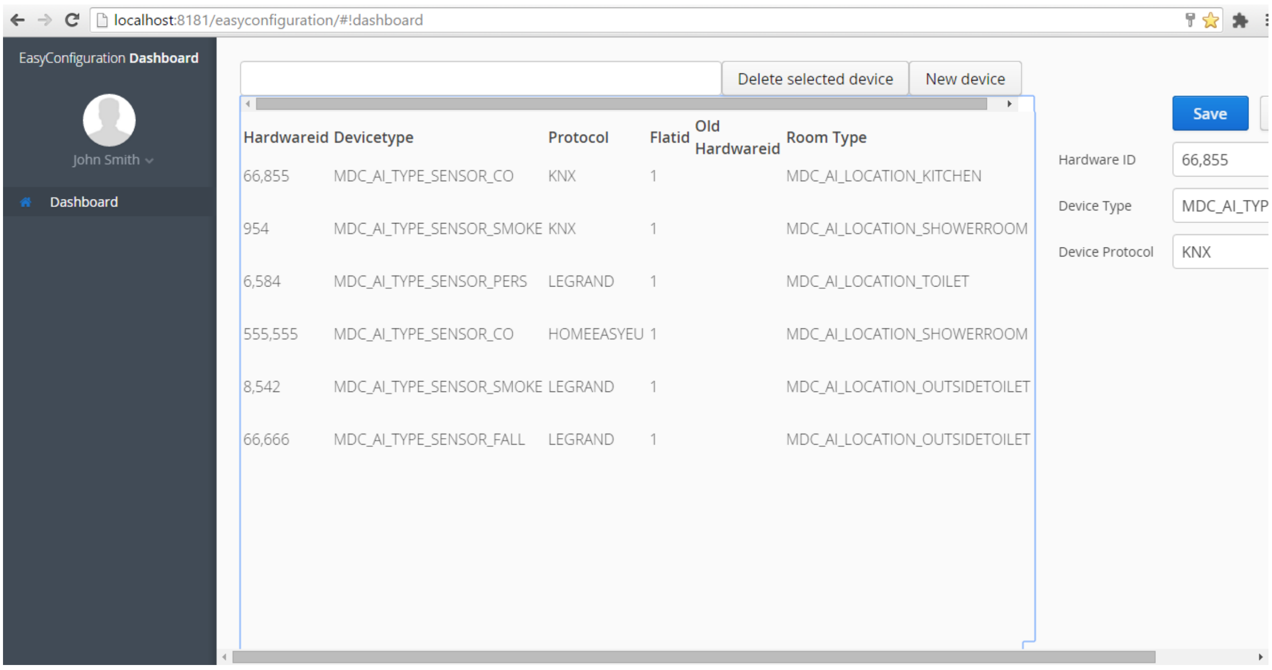
As mentioned earlier, if the system and the app becomes user friendly enough the app should only be for private users. In this case this edit/add screen should not be an option for the private user since he should only use the scan function to add devices with. Editing devices should not be an option for the private user as it leads to buggy behaviour. Private users shouldn't be concerned with protocols, device types, id's or anything like that. But as long as the app is used by technicians, functions like edit and add are needed, for example to be able to quickly add a dummy device with hardware id "123" and test it out.

4.5 Web page

The webpage is the second of the two controller clients in the system, and plays a smaller role than the app. Development on the app was always prioritized in this project over the web page, but nonetheless some platform to manage the system with was part of the requirements. The most important function of the web page is to create and delete devices in the system, and it is only meant for company technicians to use. It is the only point in the system where the user can permanently delete device records from the system. As mentioned previously, devices should be created with the web page and then linked/unlinked from houses using the app. A nice feature provided by the web page, is that it automatically generates a QR tag when a new device is created matching the hardware id of the device.

In terms of communication with the central server, the web page is simply another client implementing the client API provided by the server. Like with the app it is again important to implement server calls asynchronously, to free up the UI thread.

A screenshot of the webpage can be seen on figure 4.6. The UI of the webpage was not prioritized highly when developing the system. Adding devices to the system is done by clicking the *New device* button which resets the device form on the right side. When the device form has been filled out, clicking *save* will store the device on the central server. Clicking on a device record in the grid, will fill out the device form prepopulated with the data of the device. The data can then be edited, whereafter clicking the *save* button, again will save the device on the central server.


localhost:8181/easyconfiguration/#!dashboard
EasyConfiguration Dashboard
John Smith
Dashboard
Delete selected device
New device
Save

Hardwareid	Devicetype	Protocol	Flatid	Old Hardwareid	Room Type
66,855	MDC_AI_TYPE_SENSOR_CO	KNX	1		MDC_AI_LOCATION_KITCHEN
954	MDC_AI_TYPE_SENSOR_SMOKE	KNX	1		MDC_AI_LOCATION_SHOWERROOM
6,584	MDC_AI_TYPE_SENSOR_PERS	LEGRAND	1		MDC_AI_LOCATION_TOILET
555,555	MDC_AI_TYPE_SENSOR_CO	HOMEEASYEU	1		MDC_AI_LOCATION_SHOWERROOM
8,542	MDC_AI_TYPE_SENSOR_SMOKE	LEGRAND	1		MDC_AI_LOCATION_OUTSIDETOILET
66,666	MDC_AI_TYPE_SENSOR_FALL	LEGRAND	1		MDC_AI_LOCATION_OUTSIDETOILET

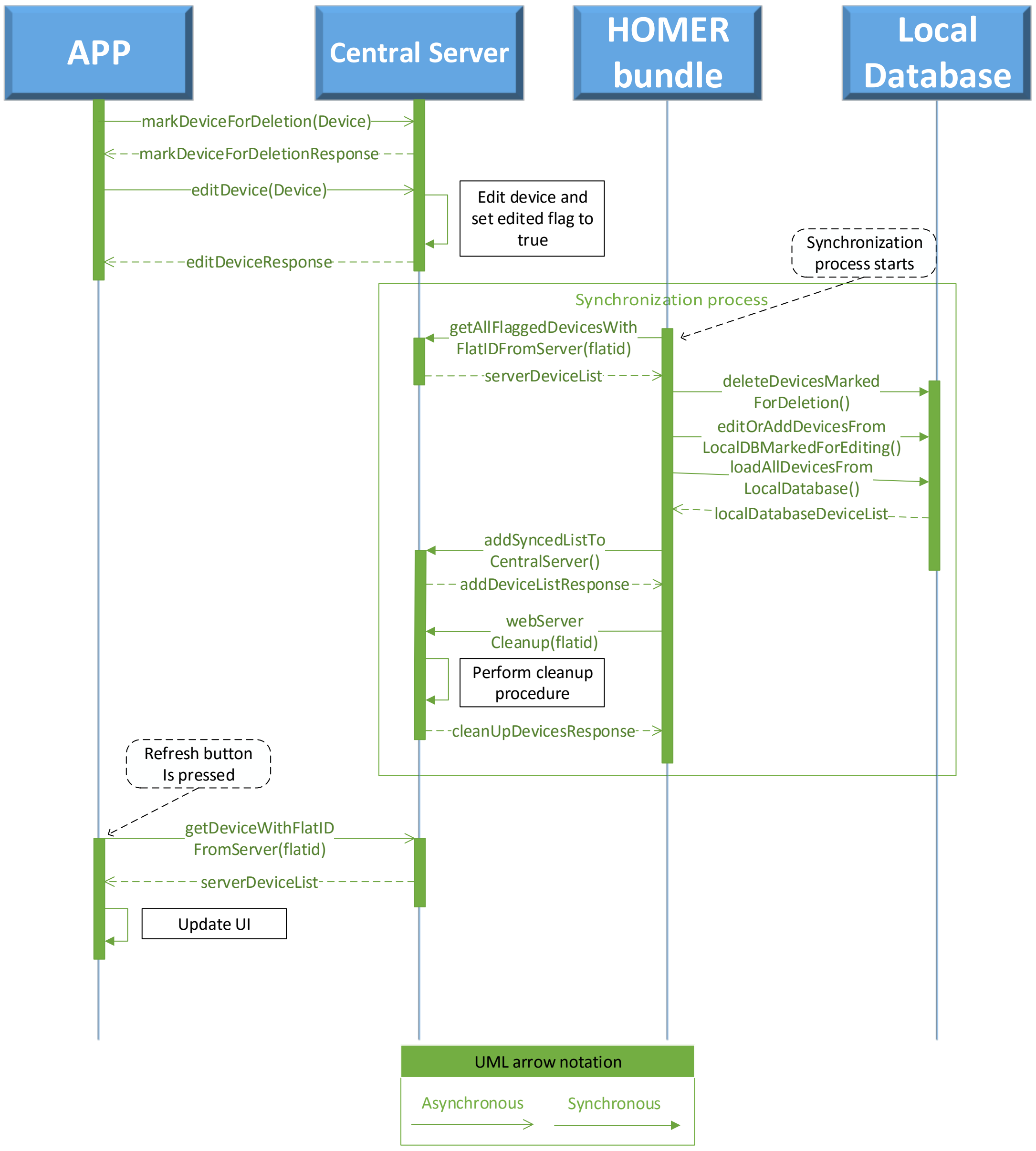
Hardware ID	66,855
Device Type	MDC_AI_TYP
Device Protocol	KNX

Figure 4.6: Screenshot of the webpage

4.6 Synchronization

When the app requests a change on the server, then the change should not be carried out immediately, but rather some mechanism should be used to indicate, that the given change is scheduled to happen. This is because if you simply delete the device on the central server without leaving any indication of what device was deleted, then HOMER doesn't know that the device was deleted. Therefore when the sync procedure runs again, HOMER will think it has a device the central server does not, and then the device will be added to the server again. To solve this a flagging mechanism has been used in the system, to signal when changes have have been scheduled to happen on a device.

While describing the synchronization in this section, figure 4.7 can be kept in mind as a reference to how the overall synchronization flow looks like.


APP
Central Server
HOMER bundle
Local Database
markDeviceForDeletion(Device)
markDeviceForDeletionResponse
editDevice(Device)
Edit device and set edited flag to true
editDeviceResponse
Synchronization process starts
Synchronization process
getAllFlaggedDevicesWith FlatIDFromServer(flatid)
serverDeviceList
deleteDevicesMarked ForDeletion()
editOrAddDevicesFrom LocalDBMarkedForEditing()
loadAllDevicesFrom LocalDatabase()
localDatabaseDeviceList
addSyncedListTo CentralServer()
addDeviceListResponse
webServer Cleanup(flatid)
Perform cleanup procedure
cleanUpDevicesResponse
Refresh button Is pressed
getDeviceWithFlatID FromServer(flatid)
serverDeviceList
Update UI
UML arrow notation
Asynchronous
Synchronous

4.6.1 Setting flags

Each device contains 3 flags; a deletion flag, an editing flag and an unlink flag, showing whether the device has been scheduled for deletion, editing or unlink. Only the webpage can schedule a device for deletion. The app can schedule a device for edit or for unlink. If a device has been scheduled for unlink or deletion, the device will not be immediately deleted or unlinked from a house, but rather the flag will be set. In the case that a client wants to edit a device, then the edit is immediately carried out and edit flag is set. Sometime later when HOMER initiates a sync procedure, it will know what to with devices depending on what flags are set. If a device has either its unlink or delete flag set, then HOMER simply deletes that device from its local database. If the edit flag is set on a device, it means one of 2 things; either the device already exists on HOMER's local database but needs to be edited, or the device does not yet exist in it and therefore needs to be added. Technically it is not needed to set the edit flag on newly added or linked devices, since HOMER can simply re-insert all the devices from the central server into its local database, overwriting the old values stored. However this would result in many unnecessary database insertions on every synchronization, since devices with no changes to them will also be reinserted. To facilitate editing the hardware id of a device, an edit flag is not enough, since if the hardware id of a device is overwritten then HOMER does not know which device to edit the hardware id on. Therefore an *oldhardwareid* field was added to the device, that is only set in case the hardware id is edited. If the hardware is edited, the *oldhardwareid* is set to the previous hardware id of the device, so HOMER has a pointer to the device it needs to edit. After HOMER has performed a synchronization, the *oldhardwareid* will be set to null in the cleanup step. Care should be taken, if the user edits the hardware id more than once before a sync has taken place. In this case one should make sure that the *oldhardwareid* still points to the original hardware id and not the 2nd hardware id as shown in figure 4.8.

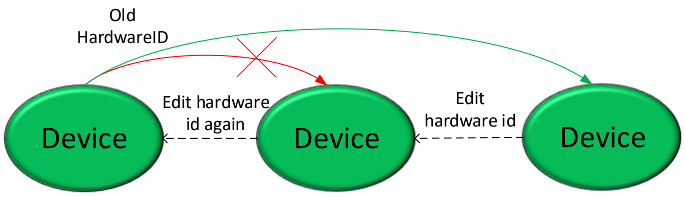


Figure 4.8: Maintaining the oldHardwareID pointer

The choice of having flags in each device to indicate changes, was weighed against having a list on the server indicating which devices are scheduled for deletion, editing and adding. The advantage of using a dedicated server list to indicate changes are that we don't have to have extra fields in our device class indicating this, which is a valid concern that might reduce the size of the a device object a bit. The drawbacks are that it adds a lot of unnecessary complexity since we would need to store the list in a separate table in the central server database (which also costs additional space). Additionally we would have to separately pass these lists around in the system, when passing device lists between the central server and HOMER and between the central server and the clients. Ultimately the flags were chosen because of the simplicity of the solution and because the infrastructure was already there. Devices as entire objects were already passed around in the system, so adding some fields to the device imposed no changes on the system. The little amount of space one could save was negligible compared to the added complexity.

4.6.2 Synchronization procedure

Synchronizing the devices on the central server with the devices on HOMER, can be viewed as a set reconciliation problem where we want to find $S_a \cup S_b$, with S_a being the devices on the server and S_b being the devices on HOMER. Our goal will be to solve the problem transferring as few devices as possible between HOMER and the central server. The first thought one should think, is that when HOMER initializes a sync procedure, then we only need to transfer the devices from the central server to HOMER, that have been modified/added since the last time the sync procedure took place. Therefore we assume that devices that does not have a flag set, already exists on HOMER and therefore we will only transfer devices to HOMER that have a flag set i.e edit, delete or unlink. This assumption is fair to make, because whenever any device is added to the system or linked to HOMER, then a flag is set that indicates this change and the only way to remove this flag is through the cleanup method which is only called by HOMER. The assumption does not hold, if devices for some reason are inserted by the admin directly into the database without using the DAO.

The following methods are called when a synchronization procedure is initialised by the HOMER bundle. All calls are performed sequentially by the HOMER bundle.

- *getAllFlaggedDevicesWithFlatIDFromServer().* This method retrieves all flagged devices from the central server with the flatid of the calling HOMER instance. Doing this allows us to easily synchronize, since we have all devices that have been changed on the HOMER side. After this call we are ready to start applying changes.

- *deleteDevicesFromLocalDBMarkedForDeletion().* Here we delete all devices in HOMER's local database that have been marked for deletion (has the delete flag set). HOMER's local database does not check for null pointers and it is therefore important to check if the device exists in HOMER before trying to delete it. This is relevant if the user has added a device and deleted it again before synchronization took place.
- *editOrAddDevicesFromLocalDBMarkedForEditing().* This method applies all edits from the central server to HOMER's database i.e all devices that have their edit flag set are either edited or added. If the device already exists in HOMER's database then the record is updated and otherwise if the device does not exist then it is added.
- *loadDevicesFromLocalDB().* All devices from HOMER's local database are loaded into the HOMER bundle. This method filters away devices with unknown device types and/or unknown protocol types.
- *prepareListForWebserver().* This method prepares a list of devices, to be sent back to the central server. It filters away all devices that were initially retrieved from the central server, by the *getAllFlaggedDevicesWithFlatID-FromServer()* call.
- *addSyncedListToWebserver().* Sends the list of devices prepared by the *prepareListForWebserver()* call, to the central server. Note that step is only necessary because we want to provide backwards functionality, as discussed in the analysis section 3.4.
- *webserverCleanUp(flatid).* This method tells the server to perform a cleanup on all devices belonging to the flatid given as parameter. A cleanup consists of resetting all flags and old hardware id pointers while also carrying out the actual changes on the central server (we can do this now that HOMER has been sync'ed). Edit and unlink flags are set to null and old hardware ids are set to null. Devices marked for deletion can now be permanently deleted from the system while devices marked for unlink will have their flat id set to -1.

At this point the synchronization is done and the central server and HOMER now both contain the same devices. Clients can now fetch a fresh copy of the devices from the central server.

Chapter 5

Implementation

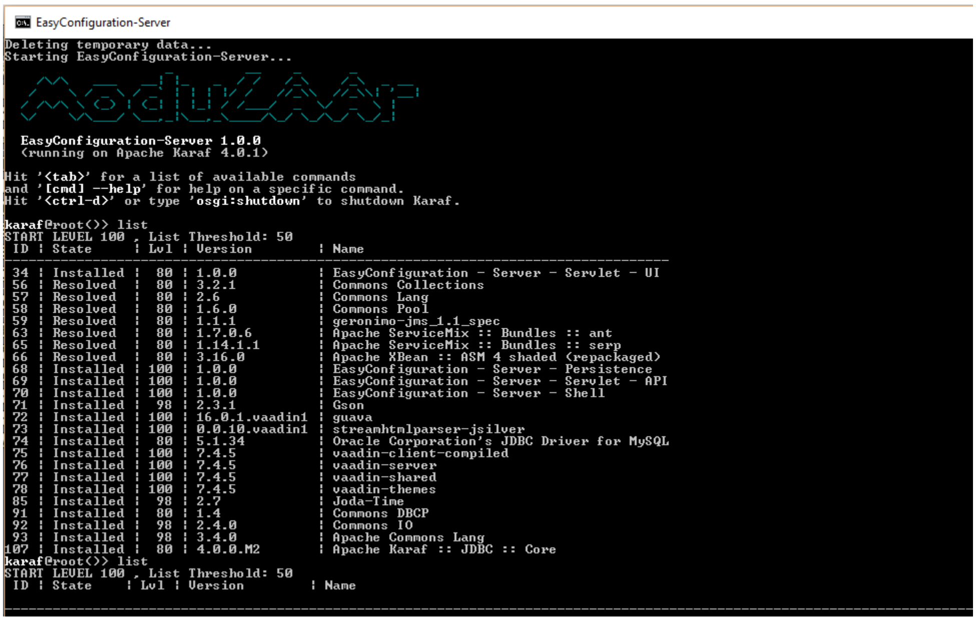
AIT's entire HOMER platform is developed to run in an OSGI container (Apache Karaf) and therefore they wanted the system for this project to be compliant with that. That meant that the software bundle developed in this project for HOMER, should be deployable as a bundle alongside the rest of all the other bundles running in HOMER. As for the central server, we also chose to implement this as a collection of bundles running in an Apache Karaf container. The instance of Apache Karaf running the central server, should not be confused with each of all the instances of Apache Karaf running on the local servers.

5.1 Development tools

Maven is a build management tool, that helps you build and manage dependencies in Java projects. Maven essentially gathers all build-related activities into one file for each folder in your project. This file is called the POM file (Project Object Model), wherein you specify what libraries your project depends on, how your project should be built (.jar, .war .ear OSGI bundle etc) and how your project should be packaged etc. For this project Maven has been used extensively as a dependency management tool, since when you have specified a dependency, then it automatically downloads all the libraries needed for that, including sub-libraries that those libraries depend on etc. Maven is also IDE independent, meaning it doesn't matter whether you are working in Eclipse, IntelliJ or NetBeans, the code will be compiled to the same. Maven offers many additional functions, but the above are some of the core functions used in this project.

Apache Aries Blueprint is used to manage bundles in an OSGI container. It acts a bit similar to the POM file, but only internally inside the OSGI container, specifying the dependencies between the bundles and ensuring that bundles are wired properly when bundles are loaded and unloaded at run-time. For example in this project, the blueprint file of the persistence bundle specifies that the DAO interface implementation should be exposed as a bundle for the server interface implementation to use.

Figure 5.1 shows the server bundles: Persistence, Servlet and shell running in Apache Karaf (the shell is only used as test client).



```
Deleting temporary data...
Starting EasyConfiguration-Server...

  EasyConfiguration-Server 1.0.0
  (running on Apache Karaf 4.0.1)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'osgi:shutdown' to shutdown Karaf.

karaf@root()> list
START LEVEL 100 , List Threshold: 50
 ID | State     | Lvl | Version          | Name
--------------------------------------------------------------------------------
 34 | Installed |  80 | 1.0.0            | EasyConfiguration - Server - Servlet - UI
 56 | Resolved  |  80 | 3.2.1            | Commons Collections
 57 | Resolved  |  80 | 2.6              | Commons Lang
 58 | Resolved  |  80 | 1.6.0            | Commons Pool
 59 | Resolved  |  80 | 1.1.1            | geronimo-jms_1.1_spec
 63 | Resolved  |  80 | 1.7.0.6          | Apache ServiceMix :: Bundles :: ant
 65 | Resolved  |  80 | 1.14.1.1         | Apache ServiceMix :: Bundles :: serp
 66 | Resolved  |  80 | 3.16.0           | Apache XBean :: ASM 4 shaded (repackaged)
 68 | Installed | 100 | 1.0.0            | EasyConfiguration - Server - Persistence
 69 | Installed | 100 | 1.0.0            | EasyConfiguration - Server - Servlet - API
 70 | Installed | 100 | 1.0.0            | EasyConfiguration - Server - Shell
 71 | Installed |  98 | 2.3.1            | Gson
 72 | Installed | 100 | 16.0.1.vaadin1   | guava
 73 | Installed | 100 | 0.0.10.vaadin1   | streamhtmlparser-jsilver
 74 | Installed |  80 | 5.1.34           | Oracle Corporation's JDBC Driver for MySQL
 75 | Installed | 100 | 7.4.5            | vaadin-client-compiled
 76 | Installed | 100 | 7.4.5            | vaadin-server
 77 | Installed | 100 | 7.4.5            | vaadin-shared
 78 | Installed | 100 | 7.4.5            | vaadin-themes
 85 | Installed |  98 | 2.7              | Joda-Time
 91 | Installed |  80 | 1.4              | Commons DBCP
 92 | Installed |  98 | 2.4.0            | Commons IO
 93 | Installed |  98 | 3.4.0            | Apache Commons Lang
107 | Installed |  80 | 4.0.0.M2         | Apache Karaf :: JDBC :: Core
karaf@root()> list
START LEVEL 100 , List Threshold: 50
 ID | State    | Lvl | Version         | Name

--------------------------------------------------------------------------------
```

Figure 5.1: Screenshot of server running in Apache Karaf

5.2 Central server

The central server has been implemented using Apache's CXF implementation of JAX-RS. To be OSGI compatible it is implemented as three separate OSGI bundles; a servlet bundle implementing the server interface, a persistence bundle implementing the DAO interface and a commons library bundle (loaded and exposed from inside the persistence bundle). Besides from these three bundles, the server has a client API library with matching URI end-points to the server interface.

5.2.1 Server interface & client API

The RESTful server interface has been implemented using the javax.ws.rs package. When implementing a method with the RESTful interface you at the very least need to specify what kind of call it is i.e GET, POST, PUT or DELETE and the Universal Resource Identifier (URI) of the call (see figure 5.1).

Listing 5.1: RESTful server interface methods implemented with the JAVAX.ws.rs package

```
@POST
@Path("/devices/add")
@AuthenticationNotRequired
public AddDeviceResponse addDevice(Device device) throws Exception;

@POST
@Path("/devices/editdevice")
@AuthenticationNotRequired
public EditDeviceResponse editDevice(Device device)throws Exception;

@GET
@Path("/devices/{hardwareid}")
@AuthenticationNotRequired
public GetDeviceResponse getDevice(@PathParam("hardwareid") Long
    hardwareid) throws Exception;

@GET
@Path("/devices/flatdevices/{flatid}")
@AuthenticationNotRequired
public DeviceListResponse
    getAllDevicesWithFlatID(@PathParam("flatid") int flatid) throws
    Exception;
```

The URI is the binding link between the central server and the clients and should uniquely define a service call. This is for example relevant in the case of the *getDevice* method, where part of the URI path is used as a parameter for the method. As can be seen from figure 5.1 the URI of the *getDevice* method is "/devices/hardwareid", where hardwareid is used as the parameter of the method interface. The other method *getAllDevicesWithFlatID()*, can't simply have the path "/devices/flatid", because it would conflict with *getDevice*'s path. All clients implementing the client API must in addition to the URI also specify the base URL used for all calls to the central server. This could look something like: "http://128.130.251.91:8181/cxf/easyconfiguration/api/1.0", which is prepended to the unique URI on a service call.

The original intention was to have the client API be identical to the JAX-RS interface used for the server, which would have given us just one single interface in the whole system, implemented by the server and used by the clients. RESTEasy is a JAX-RS implementation by JBoss intended for java clients, which is developed for exactly this purpose [Jbo16]. However, the android framework proved very uncooperative with the Jboss Resteasy technology, and therefore we switched to using Retrofit 1.9 instead, which is specifically developed as a framework for consuming API's on Android and Java platforms. This does, however, mean that we have 2 interfaces to maintain in the system, since the client API is a complete copy of the server interface. The only difference is that it is defined using Retrofit instead of JAX-RS, which means that the annotations differ a bit. Besides from these differences the URI's are still identical (they must be). As mentioned in the design section 4.4.1, we mainly want to use asynchronous communication. To do this, most service calls return to the client, using a callback pattern. The response entity returned from the server is returned inside this callbacks *success* method, or if the server interaction was unsuccessful then it returns in the *failure* method. We have used Retrofits v. 1.9 for this implementation, but it should be noted that Retrofit 2.0 was rolled out recently (October 2015) which changes the callback structure a bit, as well how the services are built. Retrofit has released a specific upgrade guide from 1.9 to 2.0 [Pö15].

When a client wants to consume the client API it can use the `ServiceGenerator` class implemented in the api.lib package on the server. The class has one single method called `createService` that is overloaded so it can be called with or without a username and password. No matter what, a client API must be provided, as well as a base URL used for building the path URI for each call with. The method will instantiate an OkHttp client and configure it with a timeout as specified, as well as log level as can be seen in code listing 5.2(there are many additional configurations that can be use). The `createService` method will return an adapter object capable of calling all the API methods specified in the provided client API. An example of instantiating such an object and making a call can be seen in figure 5.9.

Listing 5.2: Device class properties

```
final OkHttpClient okHttpClient = new OkHttpClient();
okHttpClient.setReadTimeout(httpTimeout, TimeUnit.SECONDS);
okHttpClient.setConnectTimeout(httpTimeout, TimeUnit.SECONDS);

RestAdapter.Builder builder = new RestAdapter.Builder()
        .setEndpoint(baseUrl)
        .setClient(new OkClient(okHttpClient))
        .setLogLevel(RestAdapter.LogLevel.FULL);

 //if user/pass was provided to method, then a so called
     RequestInterceptor is configured and added to the builder object

RestAdapter adapter = builder.build();

 return adapter.create(serviceClass);
```

5.2.2 JPA Persistence bundle

The database used by the central server is created and managed by OpenJPA's implementation of JPA (Java persistence API). When starting up the Apache Karaf framework for the central server, one of the first things to happen is the initialization of the MySQL database, its database drivers and the OpenJPA framework. OpenJPA looks for the persistence.xml file which has to be located in the META-INF directory in the classpath. It reads the configurations settings stored in there and links the database with the persistence bundle containing the DAO. This procedure happens during the startup of the Karaf OSGI framework. When the persistence bundle is completely loaded, its blueprint file will have initialized an *EntityManagerFactory* object in the DAO bundle for the DAO implementation to use, which will be further discussed in the DAO section 5.2.4.

The nice thing about using JPA is that once it is set up and configured, one doesn't need to be concerned with creating and managing a database. JPA does it automatically for you, provided that you have annotated the classes to be included in the database correctly. With JPA you can also specify whether fields of the classes should be mandatory, unique, public or private and you can specify the relationships (1-1, 1-n, n-n etc.) between classes and this will be reflected in the created database by JPA. Normally when changing a class in the commons library on a server, you would also need to reflect this change in the database. With JPA the database is automatically recreated and managed when the Device class is changed. And unless you change the constructor of the device class, then the clients will also not need any updates. This has made extending the Device class slowly, a very easy process.

5.2.3 Common entities

The common library includes response objects, entities and a utility library for the JSON converter class. The device class seen in figure 4.2 is just a POJO, but contains many extra annotations and imports to make it work with the JPA database. These annotations tell the JPA framework that this class is an entity and should be created as a table in the database with a given name. Each field in the class has a *Column* annotation, specifying the name of the field in the database, specifying whether the field is unique and whether it should be nullable. Id fields for example are usually unique and not nullable, which means we want to make sure to specify that in the column annotation. The device protocol and device types must both be provided but neither are unique and therefore they have nullable = false but do not specify being unique. The minimum required fields to create a Device instance are: hardware id, device type, device protocol and room type. Figure 5.3 shows how the *hardwareid* and *devicetype* fields are specified in the device class, with all their annotations used by the JPA framework.

Listing 5.3: Device class properties

```
@Column(name = "hardwareid", nullable = false, unique = true)
 @SerializedName("hardwareid")
 @Expose
 private Long hardwareid;

 @Column(name = "devicetype", nullable = false)
 @SerializedName("devicetype")
 @Expose
 private String devicetype;
```

Response objects in the common library are there to have wrappers for everything instead of parsing raw types. They also act as data contract ensuring that clients can rely on receiving their response payload in certain format. So for example instead of having the server calls *addDeviceList(deviceListWrapper)* and *getAllDevicesWithFlatID(flatid)* return a list of devices they instead return a *DeviceListResponse* object that contains a device list, a success boolean and a message. All response objects are implemented as serializable objects, because they need to be serialized before being sent back to the clients.

5.2.4 DAO Implementation

The DAO implementation consists of a range of methods offering various database operations to the user. The general structure of a DAO method is shown in figure 5.4, showing a simple DAO method.

Listing 5.4: *addDevice()*: a simple DAO method

```
@Override
 public Boolean addDevice(final Device device) {
   final EntityManager entityManager = factory.createEntityManager();
   final EntityTransaction entityTransaction =
       entityManager.getTransaction();

     try {
         entityTransaction.begin();

        entityManager.persist(device);

        entityTransaction.commit();
     } catch (Throwable ex) {
      log.error("Error during interaction with database", ex);
      return false;
     }
     return true;
 }
```

Each DAO methods need an *EntityManager* object to create an *EntityTransaction*. While the Karaf framework is initializing all the bundles in the system, the OpenJPA framework together with the persistence bundle, make sure to initialize an *EntityManagerFactory* object inside the DAO class. Each DAO method will use this *EntityManagerFactory* object to instantiate an *EntityManager* object for them. Once this has been done we can start a transaction using the `entityTransaction.begin()` call and the transaction can be closed again calling `entityTransaction.commit()`. In between these two calls, data operations and queries should be placed. In the case of the simple *addDevice()* method shown on figure 5.4, the only operation used is a simple persist call, asking the JPA to create and manage the given object, which in this case is a new device.

Figure 5.5 shows the *unlinkDevice()* method, which is a bit more advanced. The method is supposed to fetch a given device, set its flatid to -1, reset its unlinkflag by setting it to false and store the device again with the new changes applied. This requires a query which is defined using the createQuery call, providing as parameters the SQL query to be executed and the object entity to return. When retrieving the result from a query you can specify whether you are expecting a list of elements or a single element, by using either `Query.getSingleResult()` or `Query.getResultList()` calls. Finally when the device has been retrieved, we set the desired fields on the device and call merge(), which is used for over-writing existing data entries.

Listing 5.5: *unlinkDevice()*: a normal DAO method

```
@Override
    public Device unlinkDevice(Device device) {
      final EntityManager entityManager = factory.createEntityManager();
      final EntityTransaction entityTransaction =
          entityManager.getTransaction();

      Device deviceResult = null;
        try {
            entityTransaction.begin();

            final Query q = entityManager.createQuery("select d from
                Device D where d.hardwareid = :hardwareid", Device.class);
            q.setParameter("hardwareid", device.getHardwareid());
            deviceResult = (Device) q.getSingleResult();

            deviceResult.setFlatid(-1);
            deviceResult.setUnlinkFlag(false);

           entityManager.merge(deviceResult);

           entityTransaction.commit();
       } catch (Throwable ex) {
         log.error("Error during interaction with database", ex);
         return null;
        }
        return deviceResult;
    }
```

Edit device method

Implementing a method for editing devices on the central server, turned out to not be as straight forward as simply editing another property. Originally the edit method had implemented in a similar way to the *cleanDevice()* method, where you simply fetch the stored device, overwrite the stored fields on it and store it again. To facilitate changing hardware ids, you have to store a pointer to the old hardware ID somewhere in the system. Therefore an *oldHardwareId* field was added to the device, storing the old hardware id of the device (if the hardware id had been modified).

However the old hardware id might not always be set and the way the device should be edited will differ depending on whether the old hardware id has been set or not. Therefore the DAO contains 2 different methods to edit a device; one of them is called *editDeviceNormal* and does it the straightforward way i.e retrieving the specified device using it's hardware id and then overwriting its values. This edit DAO call should be used when editing a device, that has not had its hardware id set. The other DAO method to edit a device is called *editDeviceHardwareID* and should be used when the hardware id has been edited, which means the old hardware id has been set. In this case, the DAO method will look through the devices records in the database and see if any of their hardware id's match with the old hardware id of the edited device - not the new hardware id of the device. When the device record has been found, all its fields will be overwritten with the new edited values and the hardware id of the device record will be updated to the new hardware id, while the old hardware id of the device record will be set to the old hardware id of the edited device as was illustrated on figure 4.8.

The minor exception is if the hardware id of the device has been edited more than once before a synchronization has taken place. In this case querying the hardware id's of the device stored in the database, for the old hardware id of the edited device, wont yield any results, since the id we are looking for is located in the old hardware id field of one of the devices in the database. The DAO method therefore needs to look through the devices records in the database and see if any of their old hardware id's match with the old hardware id of the edited device.

Logics for deciding when to use the *editDeviceNormal* or *editDeviceHardwareID* should be kept out of the DAO and instead be placed in the server interface implementation, where all logic is placed. All the methods implemented in the DAO are defined by the persistence bundle as a bean called deviceService and exposed for the server interface implementation to use. A blueprint file for the persistence bundle specifies that the Karaf framework should expose the deviceService.

5.2.5 Server interface implementation

The class *APIServiceImpl* implements the RESTful server interface and is the execution entry point of all calls coming from clients implementing the client API. Data sent between clients and the server is formatted as JSON objects and therefore need to be deserialized when received by the server and serialized when sent from the server. For clients, Retrofit does this for them, but the server interface is not (and can not) be defined with Retrofit. However, AIT had a ready-made implementation of Google's GSON library that was used by the server. The Blueprint file for the servlet bundle specifies this GSON library as a provider when starting up the bundle. The Blueprint file also specifies the DAO interface from the persistence bundle as an interface to be implemented by the *APIServiceImpl* class. In this way when Karaf starts up, first the persistence bundle is loaded as described in section 5.2.2, followed by the servlet bundle containing the *APIServiceImpl* class.

Logics should be kept as much as possible in the *APIServiceImpl* class instead of in the DAO, and therefore it is the APIServiceImpl class's responsibility to always check whether a device exists before performing an operation it. Checks for whether devices exist are not done in the DAO. To show how a service call defined in the server interface is implemented, the code for a few calls will be shown and described below.

A simple example is the *getDevice(hardwareid)* method as shown in figure 5.6. Most method implementations have the general structure, that they first check if the *deviceService* is up and running (remember the deviceService is exposed by the persistence bundle), then they check if the device(s) they are about to perform an operation on exist, then they carry out the operation and finally they return an appropriate response object. *getDevice()* follows this pattern nicely.

Listing 5.6: *getDevice(hardwareid)*: a simple servlet method

```
@Override
@AuthenticationNotRequired
public GetDeviceResponse getDevice(final Long hardwareid) throws
    Exception {
  if (deviceService != null) {
    if (deviceService.existsDevice(hardwareid)) {
      Device device = deviceService.getDevice(hardwareid);
      return new GetDeviceResponse(true,"Succesfully got
          Device",device);
    } else {
      log.error("Device with hardwareid '" + hardwareid + "'
          doesn't exist");
      return new GetDeviceResponse(false,"Unsuccesful! device
          payload is null! ",null);
    }
  }
  return new GetDeviceResponse(false,"Unable to get device. Device
      service is null",null);
  }
```

Another method not shown here is the *addDevice()* method, which works oppositely, so if the device already exists, it returns an *AddDeviceResponse* object, saying that the device already exists. If it does not already exist, the DAO method addDevice is called: `deviceService.addDevice(device)` and afterwards it returns successfully.

The *webserverCleanUp()* method seen on figure 5.7, adds a bit more complexity. The responsibilities of this method is to reset flags and pointers and carry out the specified operations on the central server. This method is only called from HOMER after a sync procedure has taken place.

The method iterates all devices on the central server, with the flat id of the calling HOMER instance. These devices are retrieved calling the DAO method `deviceService.getAllDevicesWithFlatID(flatid)`. Thereafter the devices are iterated and if the deleteFlag is set, we check if the device exists and if it does we call the DAO method `deviceService.deleteDevice(currDevice)` to delete it. A similar pattern is followed for devices marked for unlink: if they exist we call `deviceService.unlinkDevice(currDevice)` which sets the flat id of the device to -1 and sets the unlink flag to false. Devices marked for edit, simply have their edit flag set to false and their old hardware id set to null.

Listing 5.7: *webserverCleanUp()*: a normal servlet method

```java
public CleanUpDevicesResponse webserverCleanUp(final int flatid) throws
    Exception {

  if (deviceService != null) {
    List<Device> allDevicesWithFlatID =
        deviceService.getAllDevicesWithFlatID(flatid);

    List<Device> deletedDevices = new ArrayList<Device>();
    List<Device> editedDevices = new ArrayList<Device>();
    List<Device> unlinkedDevices = new ArrayList<Device>();

    for(int i = 0;i<allDevicesWithFlatID.size();i++){
      Device currDevice = allDevicesWithFlatID.get(i);

      //If device is marked for deletion then delete it
      if(currDevice.getDeleteFlag()){
        if(deviceService.existsDevice(currDevice.getHardwareid())){

          Device deletedDevice =
              deviceService.deleteDevice(currDevice);
          if(deletedDevice != null){
             deletedDevices.add(deletedDevice);
          }else{
             log.warn("Device with hardwareID: " +
                 currDevice.getHardwareid() +
                   " should have been deleted, but was not!" );
          }
       }
    }

    //If device is marked for unlink then unlink it
    if(currDevice.getUnlinkflag()){
       if(deviceService.existsDevice(currDevice.getHardwareid())){

          Device unlinkedDevice =
              deviceService.unlinkDevice(currDevice);

          if(unlinkedDevice != null){
             unlinkedDevices.add(unlinkedDevice);
          }else{
             log.warn("Device with hardwareID: " +
                 currDevice.getHardwareid() +
                   " should have been unlinked, but was not!" );
          }
       }
    }

    //if device is marked for edit then reset the edit flag and set
        oldHardwareID to null
    if(currDevice.getEditflag()){
       if(deviceService.existsDevice(currDevice.getHardwareid())){

          Device editCleanedUpDevice =
              deviceService.cleanDevice(currDevice);

          if(editCleanedUpDevice != null){
             editedDevices.add(editCleanedUpDevice);
          }else{
             log.warn("Device with hardwareID: " +
                 currDevice.getHardwareid() +
                   " should have been edited, but was not!" );
          }
       }
    }
 }
 return new CleanUpDevicesResponse(true,"Deleted
         "+deletedDevices.size() + " device(s). "
          + "Edited " + editedDevices.size() + " device(s)",
              deletedDevices, editedDevices,unlinkedDevices);
  }
  return new CleanUpDevicesResponse(false, "Unable to delete or edit
      devices from webDB. DeviceService down", null,null,null);
}
```

5.3 HOMER synchronization bundle

The synchronization bundle for HOMER, is made to start up as a bundle alongside the rest of the HOMER bundles running on the local server in the house. The blueprint file for the bundle specifies 3 classes loaded from HOMER Core, to be initiated by the OSGI container and loaded into the synchronization bundle when the bundle starts up:

- DataAccess. This is a HOMER Core database implementation used to communicate with the local database. A call could look like`dataAccess.deleteSensor(id)` (dataAccess being the name identifier for DataAccess).
- ConfigurationService. This class is used to access a configuration file locally stored on a HOMER instance. This file contains many variables related to the specific instance of HOMER that is running. We use it to access the system id of the running HOMER instance, which can be considered the flatId.
- Scheduler. A class used for making bundles runnable executing a run() method at regular intervals. This is used to make to make the synchronization process execute regularly at a specified interval. It executes every 10 seconds when the app is in use i.e changes are happening to the house plan, and every 300 seconds otherwise.

The HOMER synchronization bundle consists of just two classes; the main class and a helper class. The main class contains the synchronization methods used for the synchronization and is at the same time also the runnable class executing the synchronization every 10 seconds. Additionally it implements the client API from the server. The second class, DeviceConverter, is a helper class used for device conversion between the Device entity defined in our system and the DBSensor/DBActuator classes used by HOMER's local database.

5.3.1 Synchronization methods

Synchronization takes place inside the *run()* method, called by the scheduler in regular intervals. The contents of the *run()* method is shown below. Whenever the *run()* method is called, synchronization takes place which involves calling the following methods in order:

1. *getAllFlaggedDevicesWithFlatIDFromServer()*;
2. *deleteDevicesFromLocalDBMarkedForDeletion()*;
3. *editOrAddDevicesFromLocalDBMarkedForEditing()*;
4. *loadDevicesFromLocalDB()*;
5. *prepareListForWebserver()*
6. *addSyncedListToWebserver()*;
7. *webserverCleanUp()*;

It should be noted that because HOMER uses DBSensor and DBActuator, it not only means that we need device conversion, but it also means that lot of code has to be duplicated to fit for both types. Whenever possible, this duplicated code has been left out of this report and rather described.

The first call,*getDevicesWithFlatIDFromServer()*, is a really simple method that just calls the `easyConfigurationAPIClient.getAllDevicesWithFlatIDSynchronous(flatid)` method of the client API and stores the result in a local array list as well as in Hash Map for efficiency purposes.

The code for *deleteDevicesFromLocalDBMarkedForDeletion()* can be seen in figure 5.8. The code iterates all the devices from the central server fetched in the previous step, and checks if the delete flag is set on them. If it is, we check if the device is a sensor or an actuator and continue appropriately. The code is mirrored for sensors and actuators, since only method names and parameters differ - the structure of the code remains the same. After the device type check, we want to find the device in the local database. This is however, not so easy since HOMER cannot look up devices on a hardware id - only on it's own internally generated database id. Therefore we fetch all devices of the appropriate type from the local database and iterate them until we find a device with a matching hardware id. Once this device has been found we delete it on the local database by calling `dataAccess.deleteSensor(dbIDForDeletion)` (sensor is replaced with actuator if we are dealing with an actuator). A detail is that on the device that is about to be deleted, we check whether the oldHardwareId has been set. If it has then we need to search the local database for the oldHardwareId instead of the hardwareId.

Listing 5.8: *deleteDevicesFromLocalDBMarkedForDeletion()*: a method used for synchronization

```
private void deleteDevicesFromLocalDBMarkedForDeletionOrUnlink(){

   for(int i = 0;i<devicesWithFlatIDFromServer.size();i++){
      Device currDevice = devicesWithFlatIDFromServer.get(i);

      if(currDevice.getDeleteFlag() || currDevice.getUnlinkflag()){
         if
              (DeviceType.isSensor(DeviceType.getDeviceType(currDevice.getDevicetype())))
              {

            List<DBSensor> localDBSensors = dataAccess.getSensors();
            Integer dbIDForDeletion = 0;

            for(int j = 0;j<localDBSensors.size();j++){

               Long idToCompareOn = currDevice.getHardwareid();

               if(currDevice.getOldHardwareid() != null){
                  idToCompareOn = currDevice.getOldHardwareid();
               }

               if((long)localDBSensors.get(j).getHardwareId() == (long)
                    idToCompareOn){

                  //Bingo! we found the corresponding sensor in the
                       LocalDB

                  dbIDForDeletion = localDBSensors.get(j).getSensorId();
                  dataAccess.deleteSensor(dbIDForDeletion);
               }
            }
         }else
            if
                (DeviceType.isActuator(DeviceType.getDeviceType(currDevice.getDevicetype()))
                {

               //mirrored code for actuators
            }
      }
   }
}
```

The *editOrAddDevicesFromLocalDBMarkedForEditing()* method is by far the biggest and most complex of the synchronization methods, but is too big to include here. Structurally it is very similar to the delete procedure described above. The method iterates the device list retrieved from the server and if a device has its edit flag set, while neither of the unlink or delete flags are set, then we want to edit it in the local database. Again we need to find it the locally stored copy in same way as described for the delete procedure because we can't look up devices on their hardware id. When the device has been found, we apply the changes and save the object in the database again. A problem we might encounter, is if the device type of the device has been edited e.g. from an actuator to a sensor. In this case we will not find the device, in the local sensor table and instead we need to search the local actuator table. If the device could not be found in either table, it means the device is newly created/linked and in that case, it should be added to the local database.

5.3.2 Device Converter

The Device converter class is responsible for converting between the DBSensor/DBActuator classes used by the HOMER database and the Device class used by the rest of the system. This is done through 4 different methods: *DeviceTODBSensor*, *DeviceTODBActautor*, *DBSensorToDevice* and *DBSensorToDevice*. The first 2 methods are mirrored for sensors/actuators and the last 2 methods as well. Conversion from a Device to a DBSensor/DBActuator is very simple, since DBActuator and DBSensor contain all information that a Device does plus more. Remember that a device just needs the hardwareId, device protocol, device type and room type to be created. Therefore we simply create a new instance of device and populate these 4 fields with information from the DBSensor/DBActuator. So conversion in this direction, involves throwing away unneeded data.

Conversion from Device to DBSensor/DBActuator involves supplying some dummy data to create the DBSensor/DBActuator, since those entities contain extra fields that the Device entity does not. The dummy data involves inputting either null, an empty string, 0 or false in the unnecessary fields. The DBSensor/DBActuator contain a room id pointing to a Room entity, while a Device just contains a room type. Therefore when translating from Device to DBSensor/DBActuator we need to ensure that a room entity in the database exists with a matching room type to the Device and if not then create a new room entity with the given room type.

5.4 App

The app was developed for the Android platform and programmed in Java. The app consumes the client API offered by the server using Retrofit 1.9, which as mentioned, conveniently also takes care of serializing/deserializing objects to/from JSON format.

The android app consists of 2 activities; the main activity and the checkin activity. In addition to this, comes the 3 minor dialog activites; edit and add device activity, as well as the select room activity. Each activity consist of some number of visual elements, all stored in the resource folder of the android app project, as well as the logics using those visual elements implemented with Java in the activities.

5.4.1 Main Activity

As mentioned in the design section, the Main activity uses 5 methods from the client API to communicate with the central server. Those methods are accessed through the API adapter object created by Retrofit using the *ServiceGenerator* class. An example is shown below where we instantiate an adapter object using Retrofit and afterwards make an asynchronous call using the object 5.9.

Listing 5.9: Pattern of an API call using adapter object from Retrofit

```
//APIInterface.class : client API offered by server.
//API_BASE_URL      : String defining the base URL of the server.
//DeviceListResponse : Response type for getAllDevicesWithFlatID()
    call.

APIInterface clientProxy =
    ServiceGenerator.createService(APIInterface.class,API_BASE_URL,
    userName, userPassword);

clientProxy.getAllDevicesWithFlatID(currentFlatid, new
    Callback<DeviceListResponse>() {

   @Override
   public void success(DeviceListResponse response,Response ignored) {
      // Do stuff if call is succesful
   }

   @Override
   public void failure(RetrofitError retrofitError) {
```

```
            Log.d(TAG, "Failure, retrofitError" + retrofitError);
        }
    });
```

On the left-side vertical pane we have a scan button and a button to create devices in the system. All scan-related functionality was implemented using Zxing QR-tag library borrowed from here (https://github.com/zxing/zxing).

3-layered Expandable list

The 3-layered expandable list on the right side of the UI is not a view supported natively by android. Android includes a normal expandable list, where each group element in the list contains a list of children. In order to achieve a 3-layered list, we had to put an expandable list inside another expandable list as seen on figure 5.2. Since this is not supported by any Android library, all draw-related and size calculation-related methods that normally run in the background and that one normally doesn't have to be concerned with, had to be implemented from scratch. Consider the following scenario: The user expands a level 1 group element called (A) and immediately all 6 of its children should be shown, which means we need to calculate the size of those children plus the group element. Then the user chooses to expand 3 of the 6 children and again we recalculate the size and update the UI. Now if group element (A) is collapsed, then we again need to recalculate the size and redraw the UI, but this time we want to remember which children were already expanded. In this way, we are saving the state of each group element, so next time (A) is expanded, then the 3 children will also automatically be expanded. A basic implementation of this was borrowed from here http://mylifewithandroid.blogspot.co.at/2011/02/3-level-expandable-lists.html. The implementation provides the background calculations done when lists are expanded/collapsed, as well as a cache for storing the state of each element in the whole list. In this way we also avoid creating new views all the time, which is an expensive operation on Android.

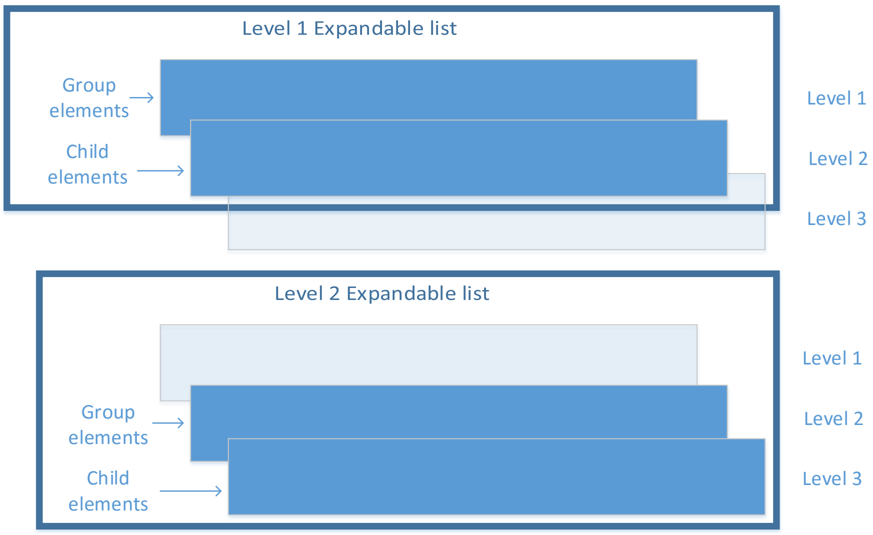


Figure 5.2: 3-layered Expandable List

The first level of the 3-layered expandable list, is a list of all the rooms in the house, where the layout of each element, is defined by a simple xml file with just an android textView in it. The children of a room are devices, which each are implemented as an expandable list. Each device element is defined by the secondlevellayer_expandablelist.xml file in the resource folder, and it includes edit- and delete buttons as well as a sync status icon as can be seen on figure 4.4 from the design section. The children of a device, are the properties linked to that device i.e device type and protocol. When the user taps room, the *getGroupView* method of that room element is called and needs a view returned to it. First the cache is checked if it stores a view state for that group element. If it doesn't, then a new expandable listView is created as well as an expandable list adapter for this view. This expandable list will contain all the devices of that room and will be saved in the cache. Almost the same procedure happens when a device group element is pressed; again we check the cache and use the state it has, and if it has nothing stored then a new static list is created to contain the properties of the device.

Figure 5.3 illustrates the role of an adapter (in our case a listView adapter). Generally an adapter works as a middle man, between a view and its data source, converting the data from the data source to UI elements usable by the listView.

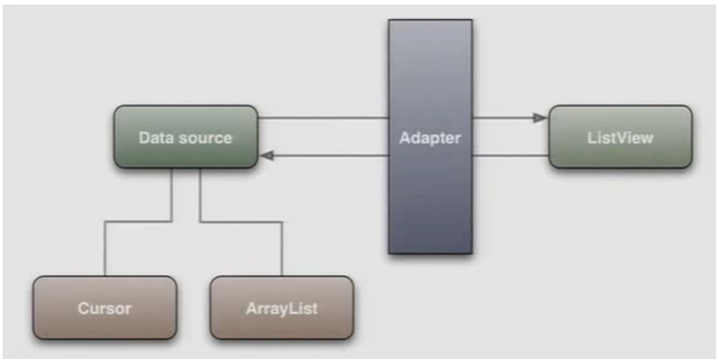


Figure 5.3: Adapter role (figure borrowed from video source: https://www.youtube.com/watch?v=N6YdwzAvw0A

When implementing buttons in android, it is usually good code practice to declare the button listener statically inside the xml element defining the button, and then implement the listener in the activity containing the button. This is how it is done for the *scan* and *add device manually* buttons, as well as for the checkin button in the checkin activity. However, child elements of expandable lists are dynamically created, which means that the views for the devices are not created before the room group element is clicked on. Therefore we cannot implement button listeners for buttons on devices in the Main activity, since we don't know how many buttons there, but moreover they have not been created yet by the time the Main activity is initialised. Instead they should be initialized when the *getGroupView()* method is called, in the second level expandable list adapter. The *getGroupView()* method is where the list elements are created first time, and it is also the place to implement logics concerning the UI of each device group element, e.g whether the device should be greyed out (delete flag is set), or whether a spinner or a tick be should be shown next to the device (edit flag set or not set).

A device currently only has 2 properties connected to it, which might seem like too little information on a device. This is because a lot of other information on the device is indirectly shown through the UI. The delete flag is reflected in whether or not the device group element is greyed out, while the value of the edit flag is reflected in whether or not the spinner or a tick is shown next to the device group element. There is no need to show the flat id of devices, since they all belong to the flat that the app is currently checked into, which is shown in the action bar. The room type of the device can be seen from the room it is grouped under, and the hardwareId of the device is the name shown on the device group element. A future plan is to force the user to provide a description of the device when he adds it, and then use this description as the name of the device instead of the hardwareId. In this case the hardware id of the device should be shown as a device property when it is expanded.

5.4.2 Checkin Activity

The checkin activity is the start up activity in the app and displays a simple screen with a text field and a button. The text field is for entering a flat id and the button can then be pressed to log in to that flat. In the future a user/pass combination should be to replace the checkin procedure, but for now this has not been implemented. When the check in button is pressed ,the Main activity is started and the flat id typed in by the user is passed along to the main activity, to be shown in the action bar.

Besides from allowing the user to login, the checkin activity also shows a progress dialog upon start up and performs a *testConnection()* call in the background to the server. If the call is successful the dialog disappears, meaning that there is connection to the central server. If connection to the server was successfully initiated, the checkin activity then calls *setSyncTimerFlag(true)*, indicating that the app is now in use and that the sync timer should therefore be set to 10 seconds instead of 300 seconds.

5.4.3 Edit, Add & select room activities

The edit, add and select room activities are really very simply activities that are very similar. They are data gathering forms implemented as a sub activities to the Main activity. When the add device or edit device is called, the add device or edit device activity is appropriately started and prompts the user for data. The edit activity is used when the edit button is pressed, and it prepopulates the data fields with the data from the device that the user wants to edit, while the add activity shows all the data fields empty. When the user has entered/edited the data he wants to, he presses the *Done* button and the data is passed to the main activity through an android intent. The data is picked up by the main activity in its *onActivityResult()* method; request code 1 is for the add activity, request code 2 is for the edit activity and request code 3 is for the select room activity.

Spinners have been implemented to select device type and device protocol. This prevents erroneous user input, and helps the technician to know what his choices are. Implementation wise, it means that the app needs to store all possible device type values, as well as device protocol values. If a protocol or device type is added or removed from the system it needs to be reflected in this list.

As mentioned in the design section, the add or edit activities should not be used at all if private users are using the app. Instead the scan function should completely configure the device in the system, without the user having to know anything about what the properties of the device are. This improves the user experience tremendously for a private user and nicely encapsulates the system preventing erroneous input or even malicious input.

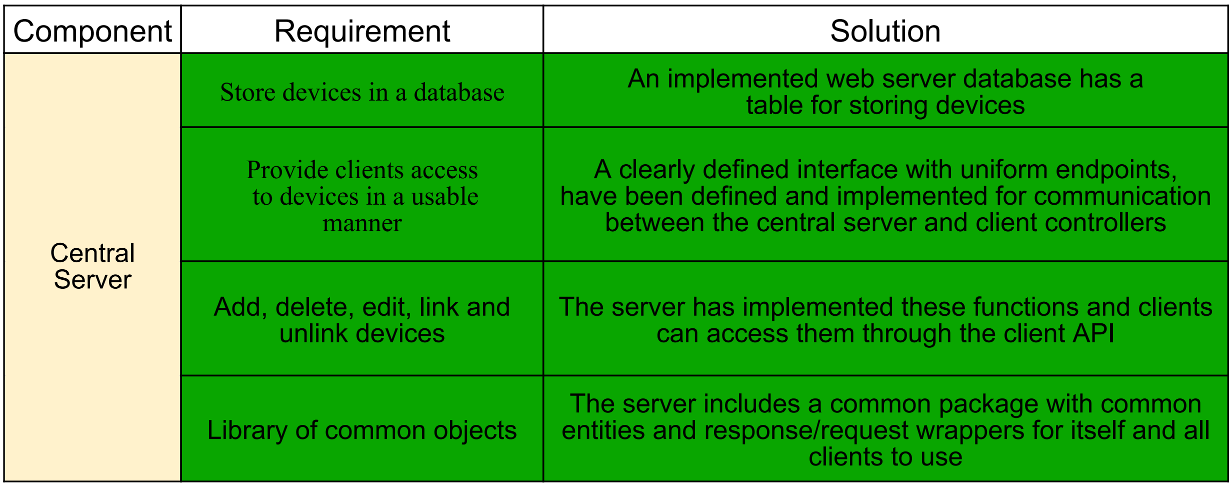
Chapter 6

Evaluation

The implemented system will be evaluated based upon the requirements discussed in section 3.5. For each of the 4 components, the requirements to them will be discussed and additionally the quality parameters will be addressed as well.

6.0.4 Central server

Figure 6.1 shows the requirements for the central server.



Component	Requirement	Solution
Central Server	Store devices in a database	An implemented web server database has a table for storing devices
	Provide clients access to devices in a usable manner	A clearly defined interface with uniform endpoints, have been defined and implemented for communication between the central server and client controllers
	Add, delete, edit, link and unlink devices	The server has implemented these functions and clients can access them through the client API
	Library of common objects	The server includes a common package with common entities and response/request wrappers for itself and all clients to use

Figure 6.1: Requirements to the central server

All of the requirements for the server have been satisfactorily fulfilled - it is capable of doing what it was planned to do. In addition to meeting the requirements for it, it has an extendible structure, because the services on it have been decoupled to a high degree. If a developer wants to implement a new call, he simply defines it in the interface and implements the call in the *APIServiceImpl* class, where all the logic should be placed. If the call needs database access, he can use the database functions already defined in the *DeviceService* interface, or define a new DeviceService call and implement it in the *DeviceServiceDAOImpl* class.

Besides from just the services related to adding, deleting, linking, unlinking and editing, the server also includes many utility and helper calls that are convenient when implementing new services.

Stress tests were conducted by making rapid asynchronous calls from the app to the server and from a simple shell to the server. Table 6.1 shows the results between the app and the server. The interval time is the amount of time the calling thread waits between each call, and the rest of the columns show the amount of calls conducted.

Table 6.1: Stress testing the app and the central server

Interval	5 calls	20 calls	100 calls	1000 calls
0 ms	App crashes	App crashes	App crashes	App crashes
50 ms	No crash	App crashes	App crashes	App crashes
100 ms	No crash	No crash	No crash	No crash

Table 6.1 shows that the app is the weak point, between the server and the app. Making 5 or more amount of asynchronous calls with no delay in between will crash the app, because the calls are asynchronous and therefore the amount of threads will pile up and either exceed the amount of cores allowed by the JVM, or cause the app to run out of available memory. Only when enough delay is inserted (100 ms) can the already called asynchronous calls be released from threads fast enough to allow new calls.

Table 6.2: Stress testing the central server with a shell

Delay	5 calls	20 calls	100 calls	1000 calls
0 ms	No crash	No crash	No crash	No crash
50 ms	No crash	No crash	No crash	No crash
100 ms	No crash	No crash	No crash	No crash

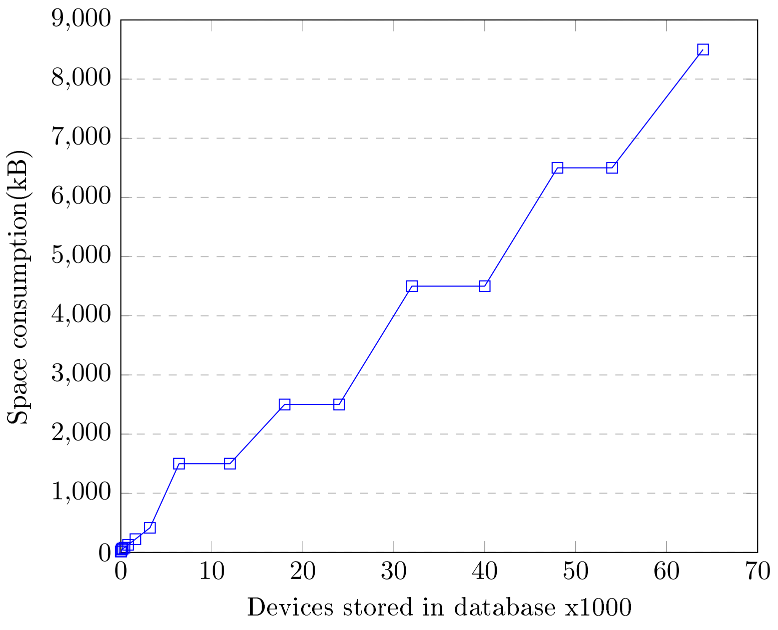
Table 6.2 shows us that the server basically never crashes when the calls are coming from a simple shell. Even with no interval at all the server never crashes. This could be because JAX-RS has built-in flood protection. However, it should be noted, that the stress tests was done on two computers on the same network. The latency will be higher when making calls over the internet.

Table 6.3 and figure 6.2 show the space consumption of the device table in relation to the amount of devices stored in it.

Table 6.3: Space consumption and amount of devices stored in database

Devices	0	50	100	200	400	800	...	12000	18000	32000	48000	64000
Space (kB)	16	16	48	64	80	128	...	1500	2500	4500	6500	8500

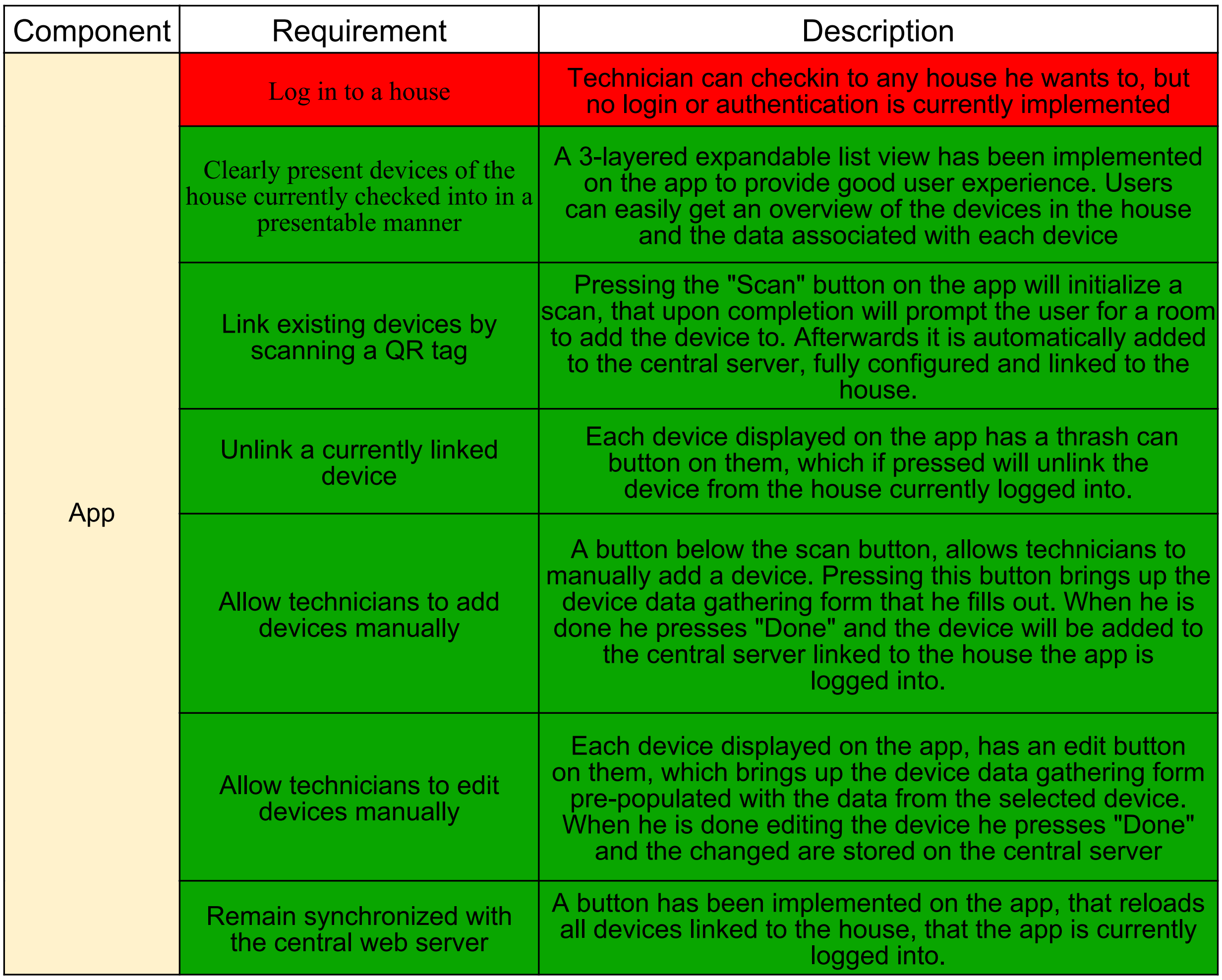
Figure 6.2: Space consumption in relation to amount of devices stored in database



Despite the plateaus, the space consumption does seem to grow linearly with the amount of devices. The plateaus on the graph probably occur due to lots of extra memory being allocated to the table to avoid reallocation too often. This is because the reallocation is an expensive operation, in which (depending on the specific implementation) the size of the current table is doubled and afterwards all the elements from the full old table, are copied to the new table. All space consumption tests were carried out by importing a standard device many times into the SQL database using the PHPMyadmin front end. All space consumption values were also read by querying PHPMyadmin for the size of the Device table.

6.0.5 App

Figure 6.3 shows the requirements for the Android app.



Component	Requirement	Description
App	Log in to a house	Technician can checkin to any house he wants to, but no login or authentication is currently implemented
	Clearly present devices of the house currently checked into in a presentable manner	A 3-layered expandable list view has been implemented on the app to provide good user experience. Users can easily get an overview of the devices in the house and the data associated with each device
	Link existing devices by scanning a QR tag	Pressing the "Scan" button on the app will initialize a scan, that upon completion will prompt the user for a room to add the device to. Afterwards it is automatically added to the central server, fully configured and linked to the house.
	Unlink a currently linked device	Each device displayed on the app has a thrash can button on them, which if pressed will unlink the device from the house currently logged into.
	Allow technicians to add devices manually	A button below the scan button, allows technicians to manually add a device. Pressing this button brings up the device data gathering form that he fills out. When he is done he presses "Done" and the device will be added to the central server linked to the house the app is logged into.
	Allow technicians to edit devices manually	Each device displayed on the app, has an edit button on them, which brings up the device data gathering form pre-populated with the data from the selected device. When he is done editing the device he presses "Done" and the changed are stored on the central server
	Remain synchronized with the central web server	A button has been implemented on the app, that reloads all devices linked to the house, that the app is currently logged into.

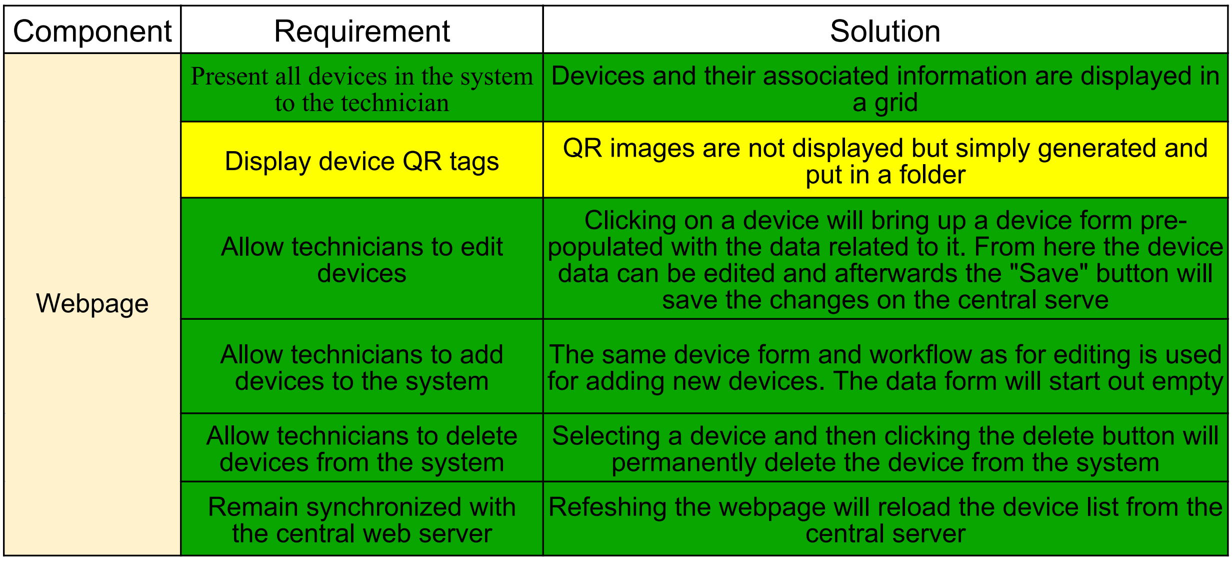
Figure 6.3: Requirements to the app

A proper login authentication mechanism was not implemented due to time constraints. A simple checkin procedure replaces it. An important feature about the app is that the user quickly can get an overview of the devices in the house and where they are placed. This has been achieved by using the 3-layered expandable list, grouping devices together under rooms they belong to and remembering the collapsed/expanded states of the list elements. Users are able to get detailed information about the devices present in the house, and they can see the state of the synchronization in the system from the UI. The names of devices displayed on the app, are currently simply their hardware id's. This could be improved by forcing customers to give a description of the device when they add/link it to the house. There is also plenty of room for improvement when it comes to the graphics of the app e.g better looking buttons, boxes, layouts and frames, but for a prototype the user experience has reached an acceptable level.

The synchronization for the app has been designed and implemented with AIT's needs in mind, which were a single technician going out into houses and setting them up. For this, a manual update procedure was sufficient i.e. tapping the refresh button to sync the app. This does not work very well if multiple users are logged into a house and are working simultaneously on configuring the house plan. In this case synchronization must happen instantly every time an update happens, to ensure that users are always seeing the latest state of the system. Automation of the sync procedure could be done by having a background thread on the app and webpage subscribe to changes from the server, meaning that we no longer have a passive server, but rather one that notifies clients about changes. However, as the sync function works now, it does not crash the system if a user has not synced the app in a while and tries performing an operation on a device that does not exist or has been updated. The database operation simply fails and execution continues, without anything happening on the central server. Additionally the app will call refresh, after execution from the failed call returns to the app and the user will then see the updated state.

6.0.6 Webpage

Figure 6.4 shows the requirements for the webpage.



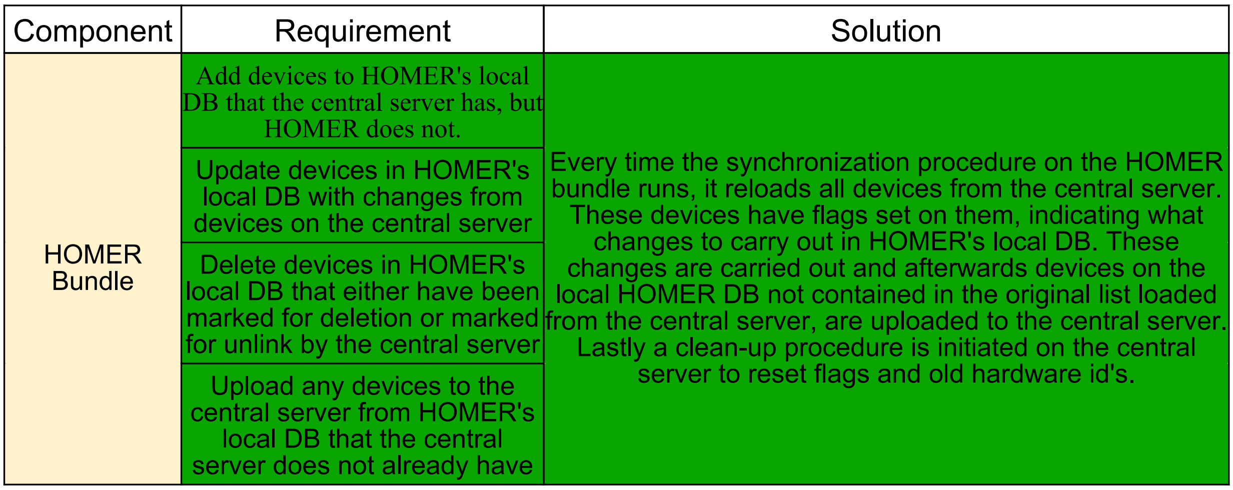
Component	Requirement	Solution
Webpage	Present all devices in the system to the technician	Devices and their associated information are displayed in a grid
	Display device QR tags	QR images are not displayed but simply generated and put in a folder
	Allow technicians to edit devices	Clicking on a device will bring up a device form pre-populated with the data related to it. From here the device data can be edited and afterwards the "Save" button will save the changes on the central serve
	Allow technicians to add devices to the system	The same device form and workflow as for editing is used for adding new devices. The data form will start out empty
	Allow technicians to delete devices from the system	Selecting a device and then clicking the delete button will permanently delete the device from the system
	Remain synchronized with the central web server	Refeshing the webpage will reload the device list from the central server

Figure 6.4: Requirements to the webpage

The basic functionality on the webpage works i.e. adding devices to the system, deleting them from the system and editing them. However, the UI is very rudimentary and could use a lot of extra work. Whenever a new device is created and added to the system, a QR code is created storing the hardware id of the device. The QR tag is not displayed on the webpage, but instead simply stored in a folder. Like with the app, syncing could also be automated where the webpage subscribes to changes from the central server.

6.0.7 HOMER synchronization bundle

Figure 6.5 shows the requirements for the HOMER bundle.



Component	Requirement	Solution
HOMER Bundle	Add devices to HOMER's local DB that the central server has, but HOMER does not.	Every time the synchronization procedure on the HOMER bundle runs, it reloads all devices from the central server. These devices have flags set on them, indicating what changes to carry out in HOMER's local DB. These changes are carried out and afterwards devices on the local HOMER DB not contained in the original list loaded from the central server, are uploaded to the central server. Lastly a clean-up procedure is initiated on the central server to reset flags and old hardware id's.
	Update devices in HOMER's local DB with changes from devices on the central server	
	Delete devices in HOMER's local DB that either have been marked for deletion or marked for unlink by the central server	
	Upload any devices to the central server from HOMER's local DB that the central server does not already have	

Figure 6.5: Requirements to the HOMER bundle

The HOMER bundle is capable of doing exactly what it was planned to do and is developed as an OSGI bundle, deployable alongside the HOME Core bundles running in a house. We have tried to reduce the amount of devices being transferred between the central server and HOMER as much as possible, but we have not achieved a linearity between the amount of devices that needs synchronization and the actual number of transferred devices. Neither could this be achieved if we had used timestamps in the system and the reason is given here.

Say we have a set **A** with 100 elements in it and another set **B** with 100 elements in it (not necessarily the same) and that these sets should always remain synchronized. If only 1 element changes in one set, we don't want to transfer all the elements to the other set for synchronization, but rather only the affected element. A timestamp solution, where all elements are marked with timestamps, achieves this by looking at the timestamp of all elements and only transferring the elements that have timestamps newer than last time a sync took place. Our implemented solution achieves the same, since it will only transfer elements that have had their flag set. Elements that have been synced with the other set, will have had their flags reset and will therefore not be transferred next time a sync procedure takes place. So both of these solutions, will transfer only the affected elements from set **A** to set **B**. The problem arises because we want the synchronization to also happen in the other direction i.e set **B** also needs to transfer changed elements to set **A**. Set **B** does not know which elements set **A** has, it only knows that set **A** definitely has the 1 element that was already transferred. Therefore set **B** will need to transfer all 99 elements to set **A** to ensure that **A** is not missing any elements from **B**.

The problem could be solved by relaxing the requirements for the synchronization a bit. Instead of requiring that synchronization is done in both directions on every sync procedure, we could instead only sync in both directions when the system starts up. In this way we would have ensured, that all devices that have been pre-deployed on HOMER will have been transferred to the central server in the beginning. Afterwards, it is a fair to assume, that devices will only be added to the system through the central server, making synchronization from HOMER to the central server unnecessary. A balance between the approaches might prove fruitful i.e once pr. 24 hours a full sync procedure in both directions will be done.

Chapter 7

Conclusions

A cloud architecture for smart homes running the HOMER platform by AIT, has been introduced and implemented. The architecture achieves remote access to data stored on local servers in smart homes, by foregoing the usage of port forwarding, VPN's or other remote access techniques and instead storing all smart-home related data on a central database server.
The implemented system consists of 4 components; a central web server with a database, an OSGI bundle running on the local server in the smart home and an app and a webpage to interact with the server. The two clients can log into a house and display all the devices present in that house. Using the clients a user can add devices to the house logged into, delete devices from it, edit devices in it and link or unlink devices to the house. These changes will then be reflected on the central server. The bundle running on the local server will frequently synchronize the devices on the local server, with the devices on the central server. This ensures that the actions done on the client controllers, are reflected on the local server running in the smart home. Users are able to scan a QR code on devices, whereafter the device will automatically be configured in the system.
Clients communicate with the server through a client API, which is implemented on the server as a RESTful service oriented architecture. This decouples clients and servers to a high degree, making it easy for software developers to maintain the system. Development on clients and servers, can happen simultaneously and almost independently of one another.

The work done in this thesis relies on an application layer platform running in houses, enabling the developer to abstract from the underlying protocols. Such an application layer has been the subject of much research of the last decade and for this project the HOMER platform developed by AIT was used.

7.1 Future work

The biggest missing piece in this thesis, is the interaction between devices, which was deliberately omitted from the project. The system has been built to manage devices, but the system does not store any interaction configuration between the devices e.g. motion sensor **A** should turn on lamp **B**. When this has been implemented, the system would be complete and ready to be tested in real houses.

In the future the system shouldn't be meant to aid technicians in setting up houses, but actually allow private users to set up their own house. Currently the app allows the technician to create, add and edit devices in the system. These functions should be removed, only allowing the private user to link and unlink devices from/to the house - never to edit or add devices. Furthermore he should only be able to link a device to the house, by using the scan function. This nicely encapsulates the system preventing faulty user input and it also spares the user the need to know about device protocols, device types, hardware id's etc, improving the user experience.
However, before implementing these changes, the app would need to be made even more user friendly than it already is. An authentication mechanism is also needed for users to log into a house.

Lastly, it would be interesting to actually move the running HOMER instance from the local server, onto the the cloud, so the smart home software is actually running on the cloud server, rather than on the local server. The disadvantage is if the internet in the house is not working, then the smart home no longer works. To solve this, the cloud could be used as the primary source for running the smart home software, while having a backup stored on the local server.

Appendix A

Source code

The source code can be found on Github at https://github.com/Jenne577/Easyconfiguration.git.

Bibliography

[4Co13] 4Control. 4 control warns users against port forwarding. URL: http://www.cepro.com/article/control4_warns_dealers_about_port_forwarding, 2013. [Online; accessed 22-January-2016].

[Ash13] Skip Ashton. Zigbee's new ip specification for ipv6 6lowpan wireless network designs. URL: http://www.embedded.com/design/connectivity/4419558/Zigbee-s-new-IP-specification-for-IPv6-6LoPAN-wireless-network-designs, 2013. [Online; accessed 22-January-2016].

[BMST10] Jeppe Brønsted, Per Printz Madsen, Arne Skou, and Rune Torbensen. The homeport system. In *Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE*, pages 1–5. IEEE, 2010.

[Car15] John Carlsen. Home automation systems review! URL: http://home-automation-systems-review.toptenreviews.com/, 2015. [Online; accessed 22-January-2016].

[CGH$^+$02] Ed Callaway, Paul Gorday, Lance Hester, Jose A Gutierrez, Marco Naeve, Bob Heile, Venkat Bahl, et al. Home networking with ieee 802. 15. 4: a developing standard for low-rate wireless personal area networks. *IEEE Communications magazine*, 40(8):70–77, 2002.

[DGV09] Simon Duquennoy, Gilles Grimaud, and Jean-Jacques Vandewalle. The web of things: interconnecting devices with high usability and performance. In *Embedded Software and Systems, 2009. ICESS'09. International Conference on*, pages 323–330. IEEE, 2009.

[GTW10] Dominique Guinard, Vlad Mihai Trifa, and Erik Wilde. *Architecting a mashable open world wide web of things*. ETH, Department of Computer Science, 2010.

[GYYL09] Khusvinder Gill, Shuang-Hua Yang, Fang Yao, and Xin Lu. A zigbee-based home automation system. *Consumer Electronics, IEEE Transactions on*, 55(2):422–430, 2009.

[HMK+05] Sumi Helal, William Mann, Jeffrey King, Youssef Kaddoura, Erwin Jansen, et al. The gator tech smart house: A programmable pervasive space. *Computer*, 38(3):50–60, 2005.

[Hom13] HomeSeer. Introducing myhomeseer remote access service! URL: http://board.homeseer.com/showthread.php?t=167137, 2013. [Online; accessed 22-January-2016].

[Hom16] HomeSeer. Homeseer supported protocols. URL: http://www.homeseer.com/faq.html, 2016. [Online; accessed 22-January-2016].

[Jbo16] Jboss. Jax-rs 2.0 client api. URL: https://docs.jboss.org/resteasy/docs/3.0-beta-3/userguide/html/RESTEasy_Client_Framework.html, 2016. [Online; accessed 22-January-2016].

[Kas14] Jacob Kastrenakes. The dumb state of the smart home. URL: http://www.theverge.com/2014/1/24/5336104/smart-home-standard-are-a-mess-zigbee-z-wave, 2014. [Online; accessed 6-November-2015].

[KBY+12] Ji Eun Kim, George Boulos, John Yackovich, Tassilo Barth, Christian Beckel, and Daniel Mosse. Seamless integration of heterogeneous devices and access control in smart homes. In *Intelligent Environments (IE), 2012 8th International Conference on*, pages 206–213. IEEE, 2012.

[KP12] Andreas Kamilaris and Andreas Pitsillides. A restful architecture for web-based smart homes using request queues. Technical report, Citeseer, 2012.

[Mer14] Rick Merritt. Nest nurtures new iot protocol. URL: http://www.eetimes.com/document.asp?doc_id=1323094, 2014. [Online; accessed 22-January-2016].

[MTMT06] Vittorio Miori, Luca Tarrini, Maurizio Manca, and Gabriele Tolomei. An open standard solution for domotic interoperability. *Consumer Electronics, IEEE Transactions on*, 52(1):97–103, 2006.

[Pö15] Marcus Pöhls. Retrofit 2 — upgrade guide from 1.9. URL: https://futurestud.io/blog/retrofit-2-upgrade-guide-from-1-9, 2015. [Online; accessed 22-January-2016].

[PRL$^+$08] Thinagaran Perumal, Abdul Rahman Ramli, Chui Yew Leong, Shattri Mansor, and Khairulmizam Samsudin. Interoperability for smart home environment using web services. *International Journal of Smart Home*, 2(4):1–16, 2008.

[Quo14] Quora. Apple homekit protocol stack. URL: https://www.quora.com/What-protocol-does-HomeKit-use-to-communicate-with-its-devices, 2014. [Online; Online; accessed 22-January-2016].

[Sch07] Jurgen Schmidt. How skype & co. get round firewalls. URL: http://www.made4biz-security.com/log/2007/02/jrgen-schmidt-hole-trick-how-skype-co.html, 2007. [Online; accessed 22-January-2016].

[Sma16] SmartThings. Smartthings speak 3 protocols. URL: https://www.smartthings.com/compatible-products, 2016. [Online; accessed 22-January-2016].

[VF02] Dimitar Valtchev and Ivailo Frankov. Service gateway architecture for a smart home. *Communications Magazine, IEEE*, 40(4):126–132, 2002.

[Wac02] Kenneth Wacks. Home systems standards: achievements and challenges. *Communications Magazine, IEEE*, 40(4):152–159, 2002.

[Wik15a] Wikipedia. Iso/ieee 11073 — wikipedia, the free encyclopedia, 2015. [Online; accessed 6-November-2015].

[Wik15b] Wikipedia. Videotape format war — wikipedia, the free encyclopedia, 2015. [Online; accessed 6-November-2015].

[WKLA13] Ehsan Ullah Warriach, Eirini Kaldeli, Alexander Lazovik, and Marco Aiello. An interpaltform service-oriented middleware for the smart home. *International Journal of Smart Home*, 7(1):115–141, 2013.

[Wro14] Daniel Wroclawsk. We're losing the war for the smart home - our tech is smarter than ever, so why is the industry so dumb? URL: http://smarthome.reviewed.com/features/were-losing-the-war-for-the-smart-home, 2014. [Online; accessed 6-November-2015].

[Zig15] ZigBee. New zigbee ipv6 protocol. URL: http://www.zigbee.org/zigbee-for-developers/network-specifications/zigbeeip/, 2015. [Online; accessed 22-January-2016].