NTNU – Trondheim
Norwegian University of
Science and Technology

UbiBazaar: App Store for the Internet of Things

Simon Stastny

Master in Information Systems
Submission date: June 2015
Supervisor: Babak Farshchian, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Recently the world witnesses the emergence of Internet of Things (IoT) technologies, as new commercial IoT products by both large tech companies and new startups are coming to market daily. The recent boom in hardware prototyping platforms such as Arduino or Raspberry Pi also contributes to the development in IoT field, as user innovators (such as Makers, tech enthusiasts and researchers) can build their IoT systems and prototypes easily and cheap.

Despite the development efforts by the latter, we feel the user innovation in IoT has not reached its full potential. We believe that this is caused by the means the user innovators reach the end users. Our research shows that there is a lack of efficient and usable software distribution channels where user innovators could offer their software the the end users, share and collaborate with other innovators.

This project proposes an app store for the Internet of Things, as a means of software distribution in IoT, a tool connecting user innovators to end users and other innovators. This proposed system should help the user innovators with distributing their app to the masses and the users to install the apps easily to their devices, just as easy as they are used to install apps on smartphones.

In the scope of our project we performed surveys consisting of questionnaires and interviews with researchers and member of Maker communities, we analyzed those and derived requirements for our system based on the findings. Subsequently, we proposed an extensible architecture for the system to be built upon, created a paper prototype and evaluated it with a focus group and implemented a working software prototype than can be used to IoT apps to Raspberry Pi devices using Docker. Both the architecture and the prototype have been evaluated accordingly.

The findings from the prestudy show a big variance between used IoT platforms and require the app store to support a multitude of different platforms and be extensible to support new platforms as they come to market. Other challenges found were relating to device configuration and context in which the devices are used.

The projects is in the stage of a working prototype, however more work needs to be done before this can be launched as a product.

Preface

This report documents the work done on project *UbiBazaar: App store for the Internet of Things*, which was written as a master thesis at Department of Computer and Information Science (IDI) at NTNU in the spring semester of 2015.

I would like to express my gratitude to my supervisor, professor Babak A. Farshchian, who provided me with valuable feedback and guidance through the project, and put me in contact with other people whose feedback I highly appreciate.

Contents

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**

2 Method **3**

2.1 Prestudy 4

2.1.1 Interviews 4

2.1.2 Literature review 5

2.1.3 Questionnaire 6

2.2 Design and evaluation of the paper prototype 7

2.2.1 Design of the paper prototype 7

2.2.2 Evaluation of the paper prototype 7

2.3 Software prototype design 7

2.3.1 Design of the software prototype 8

2.3.2 Evaluation of the software prototype 8

3 Findings and derived requirements **9**

3.1 Findings 9

3.1.1 Platforms 10

3.1.2 Deployment 11

3.1.3 Configuration 13

3.1.4 Distribution 14

3.2 Derived requirements 15

3.2.1 Platform support 15

3.2.2 Device capabilities and compatibility 15

3.2.3 App store requirements 16

4 State of the art **19**

4.1 Git deployment 19

4.2 Snappy Ubuntu Core 20

4.3 Docker 20

4.4 Resin.io . . . 22
4.5 µC Software Store . . . 23
4.6 Overview of covered requirements . . . 24

5 Design and implementation **25**
5.1 Proof of concept implementation . . . 25
5.2 Architecture . . . 28
5.3 Solution concept for Raspberry Pi . . . 29
5.3.1 Design decisions . . . 30
5.4 Paper prototype . . . 31
5.5 Software prototype . . . 32
5.5.1 Database . . . 33
5.5.2 Core library . . . 34
5.5.3 API . . . 34
5.5.4 Web frontend . . . 35
5.5.5 Ahab (installation manager) . . . 36

6 Evaluation **39**
6.1 Prototype evaluation with focus group . . . 39
6.2 Extending to support Arduino deployment . . . 41
6.3 Dockerizing CoSSMic for Raspberry Pi . . . 42
6.4 Overview of covered requirements . . . 45

7 Discussion **47**
7.1 Limitations of the current solution . . . 47
7.2 Future work . . . 48
7.3 Conclusion . . . 49

References **51**

Appendices

A API Documentation **55**

B Scenarios **59**
B.1 Scenario 1 . . . 59
B.2 Scenario 2 . . . 60

List of Figures

2.1 Model of the research process by [Oat06] 3
2.2 Iterations of the project 4

3.1 Reported usage of platforms by Makers in our questionnaire. 10

4.1 Snappy architecture. Source: [Can] 20
4.2 Differences between Docker and traditional virtualization. Source: [Docd] 21
4.3 Workflow of Resin.io. Source: [Res13b] 22
4.4 Selected screenshots of µC Software Store. Source: [EST13] 23

5.1 Landing page and filtering. 25
5.2 Adding a new app in two steps. 26
5.3 App detail and device detail. 26
5.4 App installation and user profile. 27
5.5 Device overview and adding a new device. 27
5.6 User login and registration dialogs 27
5.7 Proposed architecture. 28
5.8 Domain model for the proposed architecture. 28
5.9 Architecture of prototype solution for Raspberry Pi. 30
5.11 Welcome screen of the paper prototype. 32
5.12 Database model of our implementation. 33

6.1 Pairing instructions in prototypes. 40
6.2 App detail in prototypes. 40
6.3 Support of Arduino support in the proposed architecture. 42

List of Tables

4.1 Coverage of requirements by the state-of-the-art technology. 24

6.1 Evaluation in correspondence to requirements. 45

A.1 API: Methods provided by App service 55
A.2 API: Methods provided by Installation service 56
A.3 API: Methods provided by Device service 56
A.4 API: Methods provided by Manager service 57
A.5 API: Methods provided by User service 57

Chapter 1 Introduction

The emerging field of Internet of Things (IoT) gets a lot of attention recently as virtually every day we can see new products coming to market, large tech companies unveiling new technologies and new start-ups forming. Despite these efforts, IoT has not really taken off yet, as most of the technologies are under heavy development and the products are mostly at early stages of adoption.

As we pointed out before [Sta14], the importance of lead users and user innovation for user adoption was shown by previous research [UH88]. As was shown, the lead users of products and technologies face the same needs as the rest of the users, but they face them much earlier and their feedback can be used to improve the product or a technology so it is more mature by the time the majority gets to adopt it. The innovations developed by the users innovators can in many cases be viewed as enriching the product and attractive for both the users and the manufacturer of the product (such as in the case of apps for smartphones). The same way user innovation successfully drove user adoption in cases of other technologies, we believe it may help in the case of IoT as well.

As we mentioned previously [Sta14], we feel that one of the factors that harms the IoT user innovation is the absence of tools for easy software deployment and distribution. User innovators developing new software for their systems lack ways how to distribute the software to potential users easily.

In the previous work [Sta14], we stated how the development in the IoT undergoes a big shift from being dominated by large commercial players to being open to everyone, facilitated by the advent of small affordable open source electronic platforms, such as Raspberry Pi, Arduino, Intel Edison, BeagleBoard, mbed or ARTIK.

This shift that IoT is undergoing has also been aided by the emergence of so called “maker culture” that promotes creative use of technologies and build upon the principles of openness and knowledge sharing. The electronic platforms mentioned above are very popular in the maker community and are used in widely used in hobby IoT projects by makers.

Many of the maker IoT projects are in fact complex systems integrating commercial IoT products as well as their own hardware and software into complete systems. These makers, we believe, are the true user innovators in IoT, bringing innovation in form of new products (or prototypes) as well as experiencing integration challenges ahead of the rest of the users and developers.

For the reasons stated above we focus in this research on the makers and we try to facilitate the user innovation coming from the maker community.

This project aims to drive user adoption in IoT through facilitating user innovation. We aim to identify the main challenges for IoT software deployment and distribution as experienced by makers, and propose a technical solution addressing those challenges. We believe that the idea of app stores, very successful in the field of smartphones, may very well be applied in the context of IoT.

Although the notion of app stores for IoT exists in previous research [KK10, Dav11, MM12, SCBZ11], it has never been put into practice as a tool for deployment and distribution of user-made software. Proposing such one and implementing it is the main contribution of our research.

Our main research question therefore is:

> **RQ:** *“How can introduction of app stores improve IoT software deployment and distribution for user innovators?”*

The main contributions of our project comprise of a set of requirements for an IoT app store, an architecture proposition and a software prototype solution. Other contributions are a state-of-the-art literature review on software deployment and distribution, survey to investigate related challenges that makers face, a review of state-of-the-art solutions in the field of IoT software deployment and distribution.

Chapter 2 Method

This research project spanned over two semesters and has been performed in several consecutive iterations (as defined by [Oat06]), some with sub-iterations, and each building upon the results of previous iterations.

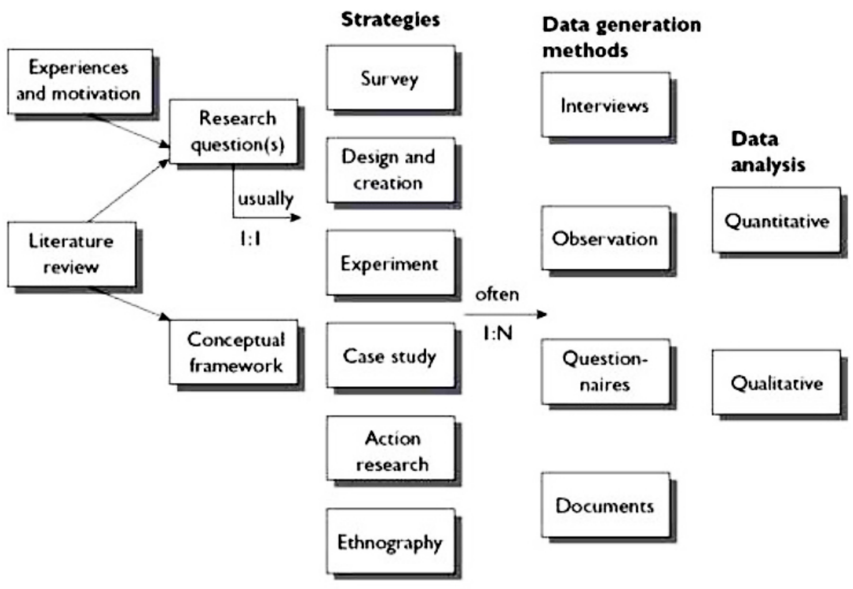


Figure 2.1: Model of the research process by [Oat06]

This is the case for our prestudy (i.e. iteration 1), which comprises of three iterations itself. The design and creation iterations (iterations 2 and 3) are however considered two separate iterations in our project, as they are the main contributions and we want to focus on each of them separately.

The figure 2.2 shows the relations between the iterations of our research project. As we followed the design cycle twice, first time for a paper prototype and second time for a software prototype, our research project consisted of these four consecutive iterations:

1. Prestudy, consisting of *a*) Interviews *b*) Literature review *c*) Questionnaire

2. Design and evaluation of a paper prototype

3. Design and evaluation of a software prototype

4. Final evaluation

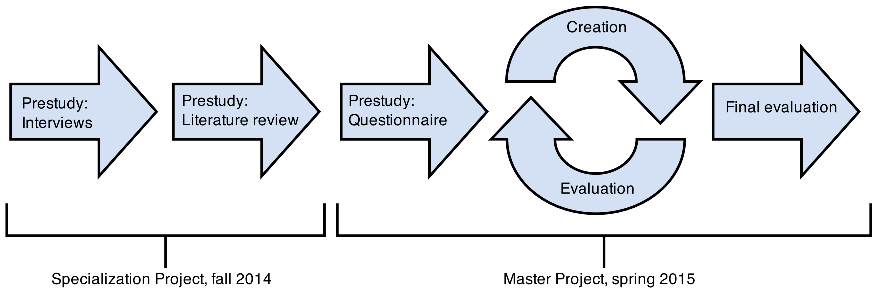

Prestudy: Interviews
Prestudy: Literature review
Prestudy: Questionnaire
Creation
Evaluation
Final evaluation
Specialization Project, fall 2014
Master Project, spring 2015

Figure 2.2: Iterations of the project

2.1 Prestudy

As mentioned above the purpose of this project is to help user innovation in IoT, represented by the development efforts by the maker community, by improving the way software is deployed and distributed. To come up with a contribution to improve these areas, first requires us to understand the needs of the user innovators and the challenges they face during the development, software deployment and distribution. To address this a prestudy needed to be performed to investigate the phenomena.

The prestudy has been performed in three iterations: *a*) a pilot round of interviews, giving us insight into the nature of challenges *b*) a literature review *c*) questionnaire with follow-up interviews.

2.1.1 Interviews

In the first part of the prestudy, taking place in autumn 2014, we scheduled a serie of interviews with two acquainted researchers and one local maker. This was a fundamental part of the prestudy and was designed to give us insight into the nature of challenges that makers face, and platforms that are used for development. The data collected in this part of the prestudy helped us to prepare a questionnaire for the last part of the prestudy, as well as to set up a query for our literature review.

The findings from the first set of interviews are described in detail in the previous report [Sta14], including a short description of the used platforms and functional descriptions of the developed IoT systems.

2.1.2 Literature review

To understand the challenges of software deployment and distribution we carried out a state-of-the-art literature review. The method of the literature review as well as the detailed findings are described in [Sta14]. We will only provide a short summary of the method here.

Since IoT is an emerging field and there is a lack of literature related directly to it, we considered also literature of related fields, such as ubiquitous, pervasive and mobile technologies, to be relevant. The nature of these fields resemble the one of IoT to some extent and the challenges that apply in them might be relevant in the field of IoT as well.

Listing 2.1: Scopus query to identify relevant literature

```
TITLE-ABS-KEY
  (
    (   "Internet of Things"    OR "Internet-of-Things" OR "IoT"
     OR "Web of Things"         OR "Web-of-Things"      OR "WoT"
     OR "pervasive comp*"       OR "ubiquitous comp*"   OR "smart"
     OR "Arduino"               OR "Raspberry Pi" )
  AND
     (  "software deployment"   OR "deployment of software"
     OR "deploying software"    OR "deploy software"
     OR "software installation" OR "installation of software"
     OR "installing software"   OR "install software"
     OR "software distribution" OR "distribution of software"
     OR "distributing software"
     OR "application store"     OR  "software store"    OR "app store"
     OR "application market"    OR  "software market"   OR "app market"
  )
AND ( LIMIT-TO ( LANGUAGE , "English" ) )
```

To find relevant literature, we also considered works focused on software and mentioning Arduino or Raspberry Pi as of interest. We also included several common terms used for software deployment and we constructed a query that we then used in Scopus to find relevant literature. Due to the large number of works and little relevancy in most of them, we introduced critetia for inclusion and exclusion. As a results, we only considered relevant works that related to *a*) deployment challenges in the above mentioned fields *b*) effects of app stores in IoT.

In total 11 works were review in the literature review [And00, BCFP14, Dav11, GVAGM10, HM06, KK10, MM07, MM12, SKS05, SCBZ11, ZWH$^+$06].

2.1.3 Questionnaire

After analysing the data from the first part of the survey, we published an online questionnaire, and asked respondents from several maker groups around the globe to fill it. In this questionnaire and interviews, we tried to investigate the means of software deployment, distribution, and collaborative development, as well as which platforms are popular to use among makers. Additional interviews with those respondents have been carried out eventually.

The questionnaire was posted to 8 maker communities and hackerspaces:

1. Trondheim Makers (Trondheim, Norway)
2. Maker Faire Oslo (Oslo, Norway)
3. MADE - Festival for Makers (Roskilde, Denmark)
4. Brmlab: Hackerspace Prague (Prague, Czech Republic)
5. Noisebridge Hackerspace (San Francisco, USA)
6. Maker Faire (global)
7. Sudo Room (Oakland, USA)
8. Bergen Hackerspace (Bergen, Norway)

Of these communities, 3 published the questionnaire on their facebook pages [1] [2] [3] and 2 on a mailing list [4] [5] . A total of 11 members from these communities responded to the questionnaire.

The full questionnaire is disclosed in the appendix of [Sta14]. It consisted of 9 required questions inquiring about platforms used by the respondents in their IoT projects, what challenges and limitations they face during deployment and distribution, and how do they collaborate and share.

[1]https://www.facebook.com/TrondheimMakers/posts/1539370499646893
[2]https://www.facebook.com/MakerFaireOslo/posts/721443127975522
[3]https://www.facebook.com/madebyorangemakers/posts/1597810183770621
[4]https://brmlab.cz/pipermail/brmlab/2015-February/008031.html
[5]https://www.noisebridge.net/pipermail/noisebridge-discuss/2015-February/046095.html

As the last step of the survey, respondents of the questionnaire were also asked to leave contact information, and the leads were used to schedule follow-up interviews. Despite 5 respondents have left contact information, only 1 was reachable and still open for an interview, which we carried out.

In the follow up interview, we inquired about collaboration on maker projects and technical details of some of the answers in the questionnaire. Possible improvements of deployment were discussed to details.

2.2 Design and evaluation of the paper prototype

After the prestudy, the findings were compiled and a set of requirements was derived from them, describing functionality desired from the system and architectural requirements. We selected which requirements will we focus on and these were used as an input for two design and evaluation iterations of our project. The first of the iterations set to design and evaluate a paper prototype of the system.

2.2.1 Design of the paper prototype

As a first step of the design iteration, we created a set of two scenarios that would represent the workflow of users in the system. Then we proceeded to create a paper prototype of our system supporting those scenarios.

Paper prototyping is a technique for quick and cheap prototyping, which suited our needs as we needed to validate our ideas and concepts in a timely manner. To design the paper prototype, we used Balsamiq Mockups [Bal] software.

The details on the scenarios and the prototype itself are to be found in section 5.4.

2.2.2 Evaluation of the paper prototype

After the scenarios and the prototype were designed, a focus group was called in and presented with the idea of an app store for the IoT. The paper prototype was shown and was walked through with the scenarios. The input from the focus group was collected and analyzed. The results of the focus group evaluation is in section 6.1.

2.3 Software prototype design

After the paper prototype was evaluated and the concepts and ideas reconsidered with the evaluation feedback, we proceeded to create a software prototype of the system, and subsequently to evaluate the prototype.

2.3.1 Design of the software prototype

To design a software prototype, we decided to implement the essential of the requirements and carry our a working software system that can be used as described in the scenarios. The description of the design and implementation can be found in section 5.5.

2.3.2 Evaluation of the software prototype

After the software prototype was designed, we evaluated some of the aspects of the prototype namely *a*) ease of adding an app *b*) extensibility for new platforms. The evaluations can be found in section 6.3 and section 6.2.

Chapter 3

Findings and derived requirements

In this chapter, we will summarize findings from literature and our data collection and derive requirements for the proposed system.

3.1 Findings

We present our findings divided into four subsections: findings on platforms, findings on deployment challenges, configuration challenges and findings on distribution challenges.

The findings on platforms used in IoT development by makers give us insight into what major platforms do we need to consider when investigating deployment and distribution challenges, and also what platforms to focus on when proposing a technical solution to address those. Supporting multiple platforms is a necessary requirement due to the rich variety of IoT platforms that will remain till the foreseeable future.

The remaining thrww, findings on deployment, configuration and distribution challenges, relate to each other to a great extent. We mentioned earlier in [Sta14] that deployment and distribution of software are two different concepts. In the scope of app stores, deployment and configuration of software are actually inherent parts of the distribution process. This is given by the nature of app stores, which are designed for simplicity to use rather than control over the process. We can see this in mobile app stores, where seeing an app in the catalog and having it installed are just a click away, without configuration being necessary in most cases.

In app stores, the distinction between distribution and deployment is typically not of an interest to the end user, and as such this trade-off between aspects of simplicity to use and user control is strongly inclined to of simplicity of use. It might turn out that this trade-off should be balanced differently in the case of IoT development as it assumes higher user expertise and will to spend more time and effort.

If we consider deployment to be a part of distribution, it is clear that challenges in deployment can cause challenges in distribution. These should always be considered with respect to each other.

3.1.1 Platforms

The survey showed that platforms such as Arduino and Raspberry Pi are used for prototyping IoT systems. Android turned out to be another popular platform used for development.

The second survey showed that in the immense number of platforms, Arduino and Raspberry Pi are the most prolific, as they were used by all respondents but two and three respectively. Some respondents also mentioned use of other platforms, such as Intel Edison, BeagleBone, Teensy, Atmel AVR, Spark, BasicATOM or ARM mbed. Vast majority of the projects are using devices of more than one platform. See figure 3.1 for an overview of reported platform usage by makers in our questionnaire.

Overall, Arduino and Raspberry Pi seem to be by far the most used platforms for IoT development in the maker community.

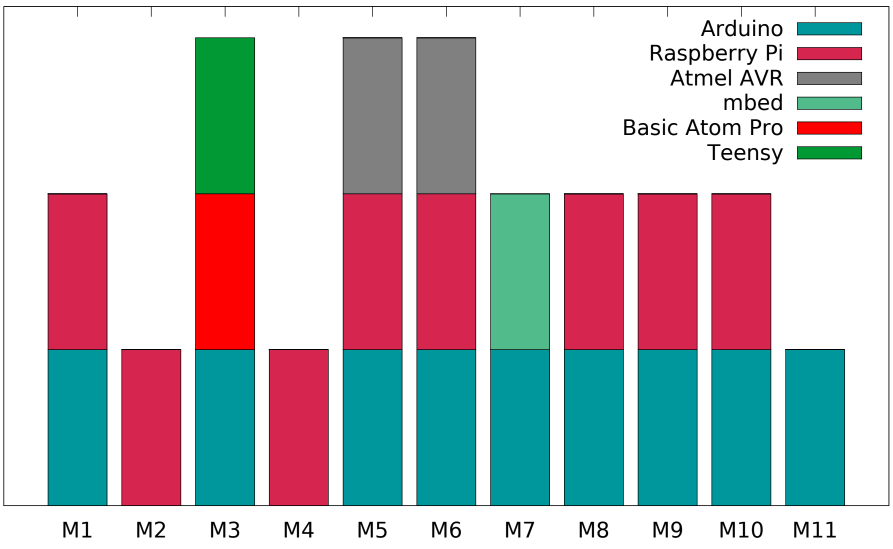


Figure 3.1: Reported usage of platforms by Makers in our questionnaire.

The wide variety of different platforms is seen by some of the research as actually harming the development in IoT. The heterogeneity can cause a vendor lock-in, as was noted by [SCBZ11] who mentioned that "The progress of Internet of Things is hampered by a fragmented world of standards and players in the field. The different and many proprietary platforms can easily lock an end user to a solution that may be outdated quickly."

Another quite interesting problem is the infinite number of device variations. As devices in IoT are usually more than just a simple board – they are equipped with sensors, actuators and other perisherals – each particular device on a certain platform can have different capabilities and these need to be considered when resolving compatibility of devices with apps.

3.1.2 Deployment

The survey shows that the wide variety of platforms gives space to an even wider variety of deployment tools and mechanisms. While some platforms require use of specialized tools to program device, other platforms require users to come up with their own ad hoc means of deployment. Some platforms also require a physical access to the device, while some enable remote deployment over network or wirelessly.

As many of the developed system consist of more platforms, of which some may require a specialized tools for deployment, we can see that deployment might not be trivial. The notion of heterogeneity of environments as a problem for deployment in IoT was noted by several respondents in the survey, as well as mentioned in the related literature several times.

Raspberry Pi

As we mentioned previously in [Sta14]: "Raspberry Pi is a credit card-sized single-board computer [Ras14], that can run ARM builds of several Linux distributions. As such, any software that runs on Linux on the ARM platform can be ran on Raspberry Pi."

Installation of operating system To run a Raspberry Pi, you need to equip it with an SD card with an installed operation system of your choice. This typically also involves installing required Linux packages etc. Several respondents of our survey noted that preparation of the this is the most time-consuming part, and that there certainly is room for improvement.

One participant noted that the approach of manual preparation does not scale well and should be reconsidered for industry scale applications: "If you're going to produce thousands of Raspberry Pis that are customized for this project then you would produce them all with this image that already has this house-monitoring software installed."

Another participant estimates this approach could speed up the deployment six-fold for his particular application: “Typical deployment takes about six hours [...] If you already have a prepared image for the Raspberry Pi, then the entire deployment takes about one hour.”

Version control system (VCS) deployment Deployment using VCS such as git seems to be a popular option for devices with Internet access and capability to compile or interpret the source code. Deployment using git (a popular VCS tool) was used by several respondents for deployment to Raspberry Pi devices.

This approach is quite popular, as developers already are using VCS to keep their source code, and it is quite easy to set up. Detailed description of git deployment can be found in section 4.1.

Docker As another way of improving deployment to Raspberry Pi, one participant mentioned *dockerizing* (i.e. prepare an image for Docker) the applications and then running them as Docker containers on the Raspberry Pi: “[the application on Raspberry Pi] could be dockerized, the Java component, the GUI, the application that stores the different power which is a web application written in PHP. This all can be packaged as a docker file, kind of an installation script and then once someone takes a Raspberry Pi and installs docker and get this docker file, he will have the service running. However there is still pairing with the smart plugs to be done.”

Docker [Doca] is a platform for automation of application deployment inside software containers, making use of Linux Containers (LXC), a virtualization environment in Linux. It allows developers to assemble whole application into a container that can be easily run by anyone either locally or deployed in the cloud, without having to deal with installing all dependencies, as they are all packed in the container as well. A detailed description of Docker is provided in section 4.3.

Arduino

Arduino devices are usually programmed using Arduino IDE running on a PC to which they are connected via a USB cable. Arduino IDE is a free and open-source software, and supposedly very easy to use. This method of software deployment was prevalent in respondents using Arduino devices.

Arduino, as well as other platforms, are also using various lower-level approaches to device programming, such as In-Circuit Serial Programming (ICSP) protocol using tools such as AVR Downloader/UploaDEr (AVRDUDE). This approach was also mentioned by some respondents, but it seems to be of a far less popularity nowadays and only used bu those who relish in working with low level programming.

Typically IoT systems comprise of devices of various platforms of which some, including Arduino, require specialized build tools in order to deploy software. Getting rid of these tools, or incorporating them into one universal tool was mentioned by the participants.

The need of physical access to a device when deploying software was, however, raised as a major challenge by participants. This is typically a problem for devices that are placed in inaccessible or distant areas, or on customer premises. Deployment over network (wired or wireless) was mentioned as a possible improvement.

There exist other means of deployment to Arduino devices, such as deploying over BlueTooth, which is described in section 4.5. Some respondents also mentioned using a wireless network controllers such as ATWINC1500 for the purposes of wireless deployment.

3.1.3 Configuration

Related works frequently mention the challenges related to post-deployment configuration of software. As we mentioned in [Sta14], many IoT systems are by definition formed by a number of interconnected Things, which in many cases need to be paired, registered, or configured in some other way in order to be able to work together. In many IoT systems, configuration might a hard nut to crack, as since the Things in many cases comprise of devices with no interface that user can use to configure it. [And00]

The problem of configuration also emerged in the survey. Participants were mentioning the need of manual configuration that takes place after the software is deployed onto the target device.

As we found out in previously in [Sta14], the related works such as [MM07, HM06, ZWH$^+$06, SKS05] suggest addressing this challenge by self-configuration, one of the four aspects of self-management by [KC03], proposing a vision of autonomic computing: "Self-configuration Installing, configuring, and integrating large, complex systems is challenging, time-consuming, and error-prone even for experts", all of which actually apply to IoT systems.

The related works also suggest that self-configuration should be achieved through context-awareness, as mentioned by [MM07] as a means of achieving self-adaptation to changes in the environment: "Since pervasive systems are likely to be long lived and dynamic, they will demand deployment, and redeployment, solutions that are able to adjust to the system's changing execution context continuously." As other means of achieving self-configuration, research suggests to use service discovery and semantic descriptions that can be interpreted by the deployment tool.

3.1.4 Distribution

As noted earlier, this projects aims to propose an app store solution for the IoT, as a distribution channel for IoT software. It is worth noting that such a product does not exists as of now and as the survey shows the distribution of IoT software is instead done manually as downloading and copying files or repositories.

Some works suggest, that the above mentioned heterogeneity of platforms is the reason why we do not have distribution channels for IoT software artifacts such as [MM12] who mention "The IoT industry doesn't have a unified hardware and software platform. It is a network of heterogeneous hardware (i.e. Things), people and services. This greatly complicates the creation of distribution channels for software applications."

However, previous research such as [KK10, Dav11, MM12, SCBZ11] suggests that app stores might be a feasible solution for IoT, and as such should be investigated.

It was mentioned above that distribution of IoT software is mostly done manually as of now. This means that the end user manually fetches the software (downloads a binary package or source code) and then manipulates with it as they wish (typically deploy the software or do changes to source code and then build and deploy).

All respondents said that they use source code repositories, such as Github or Bitbucket, to share source code of their software. Some only use private repositories to version the source code or collaborate with other developers, while others use it as a way of releasing the sources for use by the general public. Distributing software in form of source code is a means of exercising the principle of openness, as it makes it possible to make changes, both technically and legally (in most cases).

Some respondents also mentioned distributing software in form of binaries or built packages. While these are easier for an end user to use (as they do not need to build them first), it might be impossible for the user to make changes. This is of course specific to the nature of the packages.

Apart from source code repositories, respondents frequently mentioned that they share the software on community forums or through their personal website or blogs.

Two of the respondents also mentioned they publish their software artifacts on software store, but failed to specify where. As these mentioned use of Raspberry Pi platform the stores in question could possibly be Pi Store [Ind15].

3.2 Derived requirements

As noted in the chapter 1 the system that we are proposing is an app store for the IoT with which we aims to address the lack of software distribution channels in IoT, considering the challenges identified in the findings above. In this section, we revisit the above mentioned findings and use them to derive requirements for our proposed system. Having a clear set of requirements will help us to define the extent of the solution we are attempting to propose, the desired functionalities and limitations.

3.2.1 Platform support

The findings show that one of the biggest challenges in IoT is the variety of platforms and devices, especially as new platforms are coming to market. For the system we propose to be successful, it needs to support a multitude of different platforms and be extensible to support new platforms in the future.

> **Requirement 1:** *"App store should support multiple platforms."*

> **Requirement 1.1:** *"App store should be extensible to support new platforms without changes to the architecture."*

The findings also show us that while there is a wide variety of platforms, it is mainly two most prolific ones, Raspberry Pi and Arduino, that are used most frequently. There is no use in supporting a large number of platforms unless we cover the most used ones.

> **Requirement 1.2:** *"App store should support remote deployment to Raspberry Pi."*

> **Requirement 1.3:** *"App store should support deployment to Arduino without needing to physically connect the computer to Arduino and use the Arduino build tool."*

3.2.2 Device capabilities and compatibility

In mobile app stores such as Google Play, the store provider can resolve the compatibility of an app and a device by simply looking whether the device (which is one of a finite number of known model types) supports the API level needed by the app. In IoT this approach is generally not applicable, as each device can be different and the app store can not a priori know the capabilities of every device possible.

It is therefore needed to come up with a framework that can formally describe the device capabilities and let the system use this to resolve the compatibility of apps and devices. These capabilities are ideally to be discovered automatically by the system, but may need to be edited manually be the user in case the discovery does not get it right.

Requirement 2: *"App store should be aware of the device's capabilities."*

Requirement 2.1: *"App store should be able to discover the capabilities of the device automatically."*

Requirement 2.2: *"App store should it make possible for the user to fix the capabilities information manually."*

Requirement 2.3: *"App store should use the device capability information to resolve app compatibility."*

3.2.3 App store requirements

The proposed system should also support the features and functionality that are to be expected from a modern app store. The features required here were from a large part inspired by well known app stores such as Google Play. This will help make the app store easy to use, especially for people who are familiar with other app stores.

Requirement 3: *"App store should provide expected app store functionality and features."*

As the main contribution of our project is bringing a means of installation of apps on the devices, this is also one the important group of requirements.

Requirement 3.1: *"App store should support installing apps on devices."*

Requirement 3.1.1: *"App store should support registering devices."*

Requirement 3.1.2: *"App store should support a single-click installation of apps."*

Requirement 3.1.3: *"App store should support a single-click uninstallation of apps."*

Modern app stores are edge-dominant systems. This means that the success of them largely depends on the input from the uses. It is not the provider of the service who creates the content, but rather the so called "prosumers", productive consumers. This means that users not only can download and install apps, they can also be able develop their own apps and use the app store as a platforms for publishing them.

Requirement 3.2: *"App store should support adding own apps."*

Requirement 3.2.1: *"App store should support adding updates to existing apps."*

The feedback from the paper prototype evaluation, described in subsection 2.2.2, brought up the issue of categorizing the apps. In case the system scales to a store with thousands of apps, an effective means of categorization, filtering and search needs to be put in place.

Requirement 3.3: *"App store should support filtering, categorization and search for apps."*

Requirement 3.3.1: *"App store should support categorizing apps into nested categories."*

Requirement 3.3.2: *"App store should support filtering apps by attributes (platform, author)."*

Requirement 3.3.3: *"App store should support search over apps."*

The social context of app stores was also brought up in the the paper prototype evaluation, and is further described in subsection 2.2.2. The requirements derived from the feedback are:

Requirement 3.4: *"App store should support user feedback on apps."*

Requirement 3.4.1: *"App store should support user comments on apps."*

Requirement 3.4.2: *"App store should support user rating of apps."*

The store of course needs to feature a functional user management system and means of authentication and authorization. This requirement is inherent and is only mentioned *pro forma*.

Requirement 3.5: *"App store should support user registration and login."*

Chapter 4
State of the art

This chapter gives an overview on the state-of-the-art methods, technology and solutions in software deployment in the IoT field that are related to the requirements proposed for our system. It provides an analysis of their advantages, disadvantages and ideas that can be utilized in our project. Table 4.1 provides an overview of coverage of the requirements.

4.1 Git deployment

Deployment using git (and other VCS) are popular even outside the IoT field, for instance as a deployment method to cloud Platform as a service (PaaS) such as Heroku [Her] or OpenShift [Red].

Even though this deployment method is quite popular, some point out that VCS systems were not designed for deployment purposes and might not be suited for such, as they are for example unable to keep track of file permissions or empty directories, which might be an issue in some cases.[Cha]

There are several workflows how deployment via git can be implemented [Cha]. In any case, device needs to be set up in advance in order to be deployed to.

Even though of deployment is used often by the developers themselves, it might not be an appropriate deployment model for use by regular users, as they would need to set up the environment, including all the dependencies and configuration needed, which might turn out to be complicated in some cases. We decided this way of deployment was not suitable for our prototype, in which we wanted to demonstrate the ease of deployment.

4.2 Snappy Ubuntu Core

Another technology to be considered is Snappy Ubuntu Core.[Can] According to [Can15] Snappy Ubuntu Core is a new rendition of Ubuntu with transactional updates - a minimal server image with the same libraries as today's Ubuntu, but applications are provided through a simpler mechanism. The Snappy system provides a new mechanism of managing apps, having them isolated from each other, and updating them with transactional delta updates.

This Ubuntu distribution was ported to ARM devices and is one of the third-party distributions you can download from the official Raspberry Pi webpage [Ras].

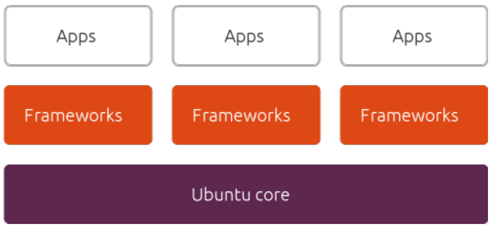


Figure 4.1: Snappy architecture. Source: [Can]

This technology has been introduced only recently and even though the technology has a certain potential it seems it has not taken off yet. We feel the community and documentation is not yet at the level of other, more popular technologies.

4.3 Docker

Docker is a platform building on top of LXC (Linux Containers), a virtualization environment in Linux. It facilitates running multiple isolated containers under a single system.[1] There is a subtle difference between a traditional virtualization which runs a separate OS inside a virtualized sandbox (such as one used in VirtualBox) and Linux Containers, which reuse the underlying system and only bring a layer of needed dependencies. This difference is best explained by figure 4.2.

Thanks to recent development efforts by Resin.io [Res13a] and by Hypriot [Hyp15], Docker now has been ported to ARM platform, and therefore Raspberry Pi, rendering Docker a feasible deployment method for Raspberry Pi.

To run an app inside a Docker container, one must first dockerize it. This means prepare a docker image that can be instantiated into a container. This is typically done by creating a so-called Dokerfile, which is a text file of a special syntax

[1]Technically detailed description can be found in [Docd]

containing instructions on building the image. The official Docker documentation [Doc15] describes it as “[...] a text document that contains all the commands you would normally execute manually in order to build a Docker image.”

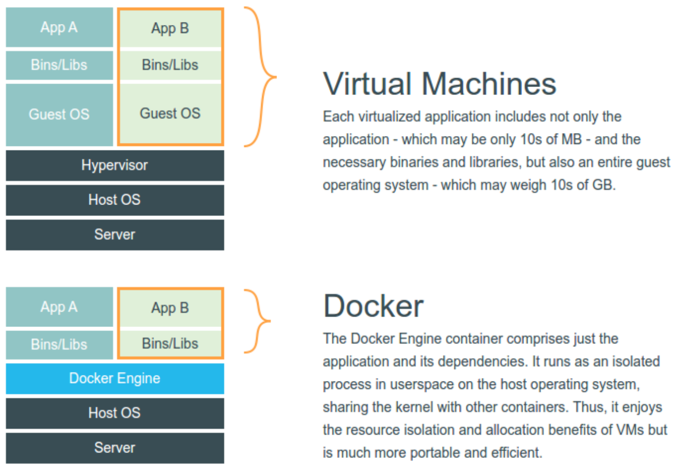

App A
App B
Bins/Libs
Bins/Libs
Guest OS
Guest OS
Hypervisor
Host OS
Server
Virtual Machines
Each virtualized application includes not only the application - which may be only 10s of MB - and the necessary binaries and libraries, but also an entire guest operating system - which may weigh 10s of GB.
App A
App B
Bins/Libs
Bins/Libs
Docker Engine
Host OS
Server
Docker
The Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.

Figure 4.2: Differences between Docker and traditional virtualization. Source: [Docd]

Docker images are layered. Each step in the Dockerfile translates to another layer, as the previous image is taken, instantiated into a container, the new step is applied to it, snapshot of container is taken and stored as another image, depending on the underlying one. The layers are thin, but too many of them in an image can be a possible performance bottleneck. Dockerfile also needs to specify its parent image, which can in turn be another image built from a stack of layers, and builds on top of another parent image. The parent image can also be a base image, which is an image without parent, which architecture needs to correspond with the architecture of the processor on which the container is supposed to run. It is therefore not possible to run a 64-bit image on a 32-bit processor or an ARM image on a x86 processor.

Resin.io, the company behind the product described in section 4.4, published a guide to building and installing Docker on Arch Linux for ARM in November 2013 and updated it several times since, with Docker v0.8.0 being the latest (as of May 2015) release in February 2014.

Hypriot (a project going under a rather poetic slogan “Docker Pirates ARMed with explosive stuff, roaming the seven seas in search for golden container plunder.”) releases SD card images of a modified Raspbian distributions with Docker installed. Their first known release on February 8th 2015 was using Raspbian Wheezy and Docker 1.4.1, the latest one (as of May 2015) release on April 16th is using the latest

Raspbian Jessie and Docker 1.6.0.

Docker features a remote API [Docb], that is REST-like and can be used bu other applications. It can be used either locally (bound to UNIX socket) or remotely (when bound to TCP socket). This technology seems to be feasible for our implementation of Deployment to Raspberry Pi, in large part thanks to the development efforts by Hypriot and Resin.io.

4.4 Resin.io

Unlike Hypriot, which is an initiative dedicated to providing Docker enabled Raspbian SD card image, Resin.io is a business providing a platform for developers which can be used to deploy software to devices such as Raspberry Pi, Beaglebone Black and (experimentally) Intel Edison [Res15]. According to their website [Res15] they are constantly working on supporting more platforms in near future.

Even though Resin.io is also deploying apps on IoT devices, the idea is different from the one of UbiBazaar. Resin.io is not an app store, but merely a bridge between software developers pushing code to a source code repository and devices that are running the software built from that code. It allows developers to deploy their code to devices easily, not to release their apps for the wide public, or for the public to install the apps on their devices.

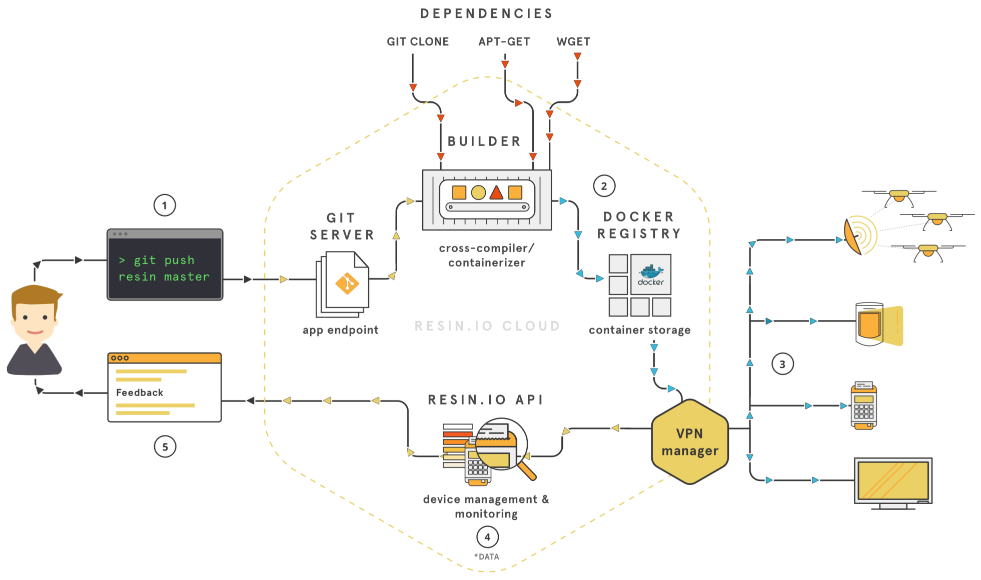


Figure 4.3: Workflow of Resin.io. Source: [Res13b]

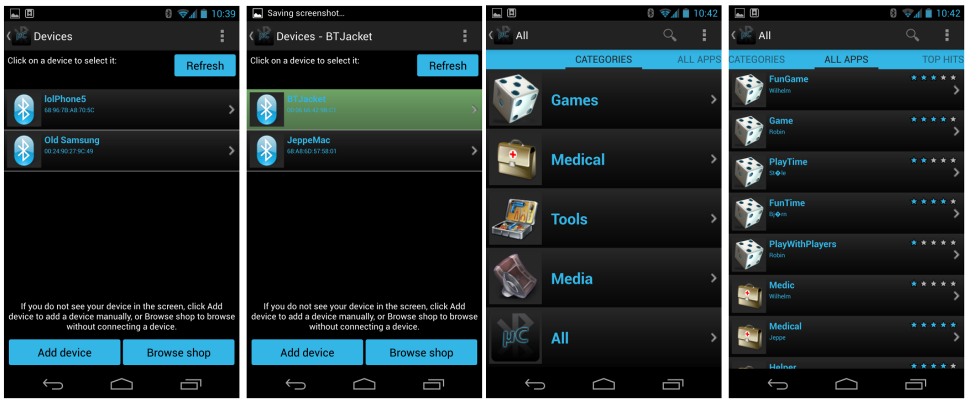
When code is pushed into a code repository in Resin.io infrastructure, it is built into a Docker image, which is in turn published to a Docker repository from which the devices are fetching it and starting the images as container. The whole process is described in detail in [Res13b].

Resin.io currently only supports having one app deployed on a device at once. The idea of UbiBazaar differs, as we aim to support running multiple apps on the same host. The idea of cross-compiling the image on a build server outside the target device also seems to be a feasible one, as it will save time for the end user.

4.5 µC Software Store

µC Software Store was developed as a student project at NTNU. The project's aim was originally to develop an app store from which apps can be installed to devices based on microcontrollers such as Arduino. The focus of the project shifted to "use Bluetooth instead of a wired connection to program an Arduino device". [EST13]

In the scope of the project, students implemented an Android app, that uses a BlueTooth shield (RN-42) to enable remote programming of an Arduino device. [New14] The system manages pairing of devices over BlueTooth and deploys software over-the-air. It features a fixed set of apps, which can be installed, but no new apps can be added and shared with other users, so the social dimension of app stores is unfortunately eliminated.


If you do not see your device in the screen, click Add device to add a device manually, or Browse shop to browse without connecting a device.

If you do not see your device in the screen, click Add device to add a device manually, or Browse shop to browse without connecting a device.

Figure 4.4: Selected screenshots of µC Software Store. Source: [EST13]

While the app only implements STK500v1 protocol for over-the-air deployment to Arduino, [EST13] claims the app has a modular design and can be easily extended to support additional protocols for deployment. This makes it a great inspiration in case of eventual implementation of Arduino deployment in UbiBazaar.

4.6 Overview of covered requirements

This section provides an overview of how our requirements were covered by the reviewed state of the art technology and solutions. We chose to focus on the deployment method perspective of our system and have not investigated the methods of discovering device capabilities and resolving compatibiloity.

Rq	State of the art	Findings
1	Resin.io (section 4.4) supports several platforms such as Raspberry Pi, BeagleBone Black, Intel Edison. All with the same deployment mechanism (Docker).	There are platforms for which Docker is not a feasible solution, such as Arduino. So to address the requirement, we need to do more than Resin.io is doing.
1.1	Several ways how to deploy apps on Raspbery Pi exists, such as *a*) Git *b*) Ubuntu Snappy *c*) Docker. Resin.io (section 4.4) is using Docker.	For Raspberry Pi, Docker seems to be a good choice of deployment method and technology.
1.2	µC Software Store (section 4.5) does over-the-air deployment of apps using BlueTooth.	Solution works, reuse of existing code is possible.
1.3	— not applicable	
2	— not investigated	
3	— not applicable	

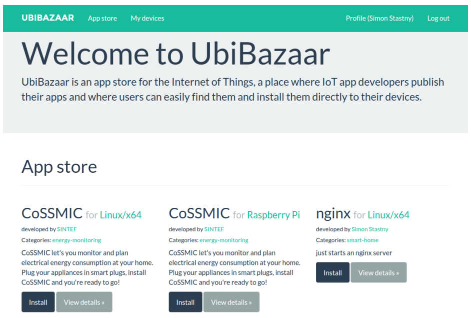
Table 4.1: Coverage of requirements by the state-of-the-art technology.

Chapter 5
Design and implementation

This chapter describes the proposed solution from an architecture and a concept, to a paper prototype to a software prototype. It also provides reasoning why some of the requirements from section 3.2 were not considered in the concept and architecture or carried out in the prototypes.

5.1 Proof of concept implementation

As noted in the chapter 1 and in section 3.2 we propose an app store for the IoT that should become a distribution channel for IoT software. This tool should facilitate user innovation in IoT by providing a platform where makers can publish their IoT apps and other users and makers can install those to their own devices. We believe this is a solution to the lack of distribution channels in IoT software, and can replace the current means of sharing software (that is making source code public in Github, etc.) with a method more efficient and user friendly.



Welcome to UbiBazaar

UbiBazaar is an app store for the Internet of Things, a place where IoT app developers publish their apps and where users can easily find them and install them directly to their devices.

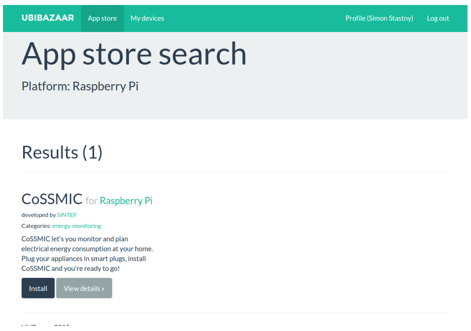
App store

CoSSMIC for Linux/x64
developed by SINTEF
Categories: energy-monitoring
CoSSMIC let's you monitor and plan electrical energy consumption at your home. Plug your appliances in smart plugs, install CoSSMIC and you're ready to go!

CoSSMIC for Raspberry Pi
developed by SINTEF
Categories: energy-monitoring
CoSSMIC let's you monitor and plan electrical energy consumption at your home. Plug your appliances in smart plugs, install CoSSMIC and you're ready to go!

nginx for Linux/x64
developed by Simon Stastny
Categories: smart-home
just starts an nginx server

(a) Landing page.



App store search

Platform: Raspberry Pi

Results (1)

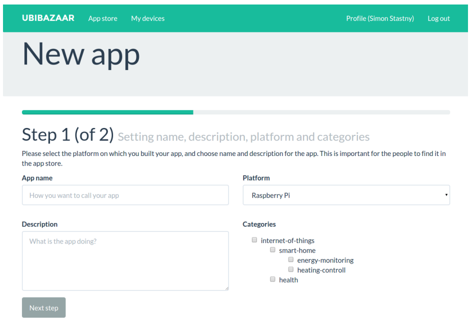
CoSSMIC for Raspberry Pi
developed by SINTEF
Categories: energy-monitoring
CoSSMIC let's you monitor and plan electrical energy consumption at your home. Plug your appliances in smart plugs, install CoSSMIC and you're ready to go!

(b) Filtering apps by platform.

Figure 5.1: Landing page and filtering.

In the end this project we designed and implemented a working software prototype of an IoT app store for Raspberry Pi devices and apps, installed as Docker containers.

The app store has a web frontend which can be accessed by users through a web browser, and which can be used to install apps to Raspberry Pi devices with just a few mouse clicks.



New app

Step 1 (of 2) Setting name, description, platform and categories

Please select the platform on which you built your app, and choose name and description for the app. This is important for the people to find it in the app store.

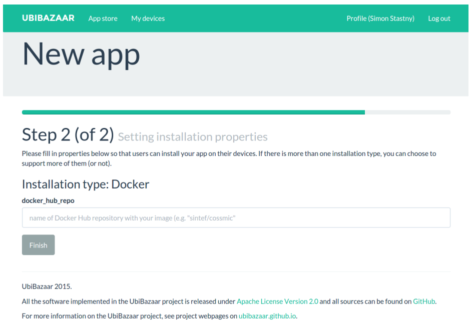
App name: How you want to call your app

Platform: Raspberry Pi

Description: What is the app doing?

Categories:
- internet-of-things
  - smart-home
    - energy-monitoring
    - heating-controll
  - health

(a) Step 1.



New app

Step 2 (of 2) Setting installation properties

Please fill in properties below so that users can install your app on their devices. If there is more than one installation type, you can choose to support more of them (or not).

Installation type: Docker

docker_hub_repo: name of Docker Hub repository with your image (e.g. "sintef/cossmic"

UbiBazaar 2015.

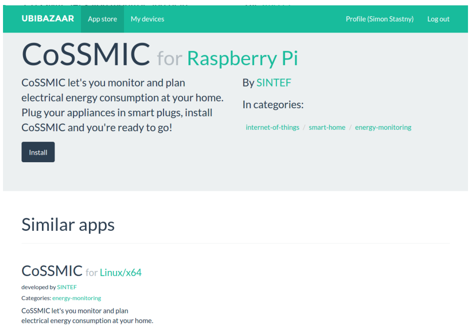
All the software implemented in the UbiBazaar project is released under Apache License Version 2.0 and all sources can be found on GitHub.

For more information on the UbiBazaar project, see project webpages on ubibazaar.github.io.

(b) Step 2.

Figure 5.2: Adding a new app in two steps.

The apps store was specifically designed for *a*) users who want to publish their own apps *b*) users who want to install apps published by others. It supports all the functionality needed from adding a new app, to registering a device, to having the app installed on the device.



CoSSMIC for Raspberry Pi

CoSSMIC let's you monitor and plan electrical energy consumption at your home. Plug your appliances in smart plugs, install CoSSMIC and you're ready to go!

By SINTEF

In categories: internet-of-things / smart-home / energy-monitoring

Similar apps

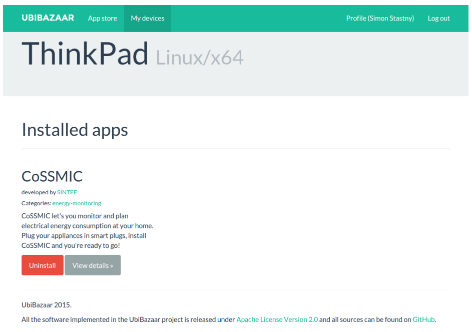
CoSSMIC for Linux/x64

developed by SINTEF

Categories: energy-monitoring

CoSSMIC let's you monitor and plan electrical energy consumption at your home.

(a) App detail.



ThinkPad Linux/x64

Installed apps

CoSSMIC

developed by SINTEF

Categories: energy-monitoring

CoSSMIC let's you monitor and plan electrical energy consumption at your home. Plug your appliances in smart plugs, install CoSSMIC and you're ready to go!

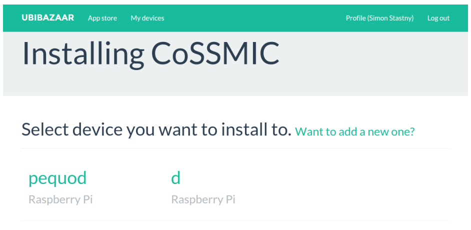
UbiBazaar 2015.

All the software implemented in the UbiBazaar project is released under Apache License Version 2.0 and all sources can be found on GitHub.

(b) Device detail.

Figure 5.3: App detail and device detail.

Some of the functionality of the app store is only available to authenticated users. While user who is not authenticated in can browse through the app store, filter the apps by attributes and categories and see their details, a user who authenticates can in addition *a*) register devices they own (figure 5.5b) *b*) install apps to registered devices (figure 5.4a) *c*) add new apps to the app store (figure 5.2).


UBIBAZAAR App store My devices Profile (Simon Stastny) Log out

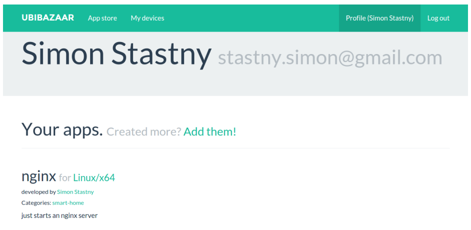
Installing CoSSMIC

Select device you want to install to. Want to add a new one?

pequod
Raspberry Pi

d
Raspberry Pi

(a) Selecting a device to deploy app to.


UBIBAZAAR App store My devices Profile (Simon Stastny) Log out

Simon Stastny stastny.simon@gmail.com

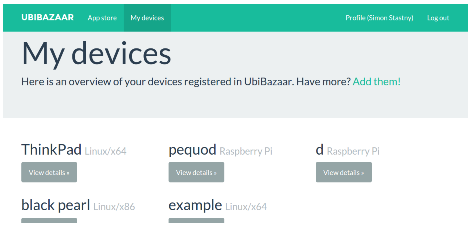
Your apps. Created more? Add them!

nginx for Linux/x64
developed by Simon Stastny
Categories: smart-home
just starts an nginx server

(b) User profile.

Figure 5.4: App installation and user profile.

User can at any time see the devices they have and what apps are installed on them, and can uninstall the apps from devices with a single mouseclick.


UBIBAZAAR App store My devices Profile (Simon Stastny) Log out

My devices

Here is an overview of your devices registered in UbiBazaar. Have more? Add them!

ThinkPad Linux/x64
View details »

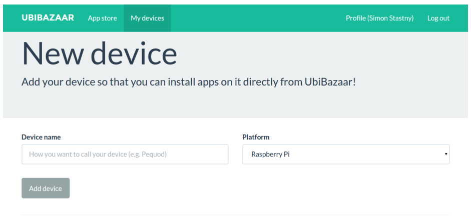
pequod Raspberry Pi
View details »

d Raspberry Pi
View details »

black pearl Linux/x86

example Linux/x64

(a) Device overview.


UBIBAZAAR App store My devices Profile (Simon Stastny) Log out

New device

Add your device so that you can install apps on it directly from UbiBazaar!

Device name: How you want to call your device (e.g. Pequod)

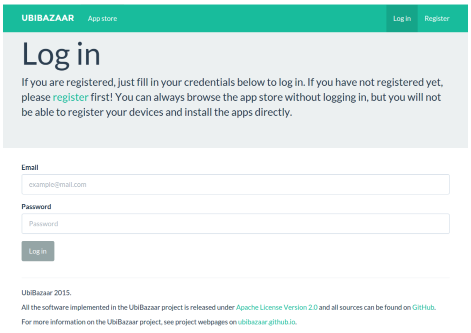
Platform: Raspberry Pi

Add device

(b) Adding a new device.

Figure 5.5: Device overview and adding a new device.

To authenticate, user first needs to register using the user registration form (figure 5.6b), providing a username and password of choice. Subsequently, user logs in through login form (figure 5.6a) with the credentials provided on registration.


UBIBAZAAR App store Log in Register

Log in

If you are registered, just fill in your credentials below to log in. If you have not registered yet, please register first! You can always browse the app store without logging in, but you will not be able to register your devices and install the apps directly.

Email: example@mail.com

Password: Password

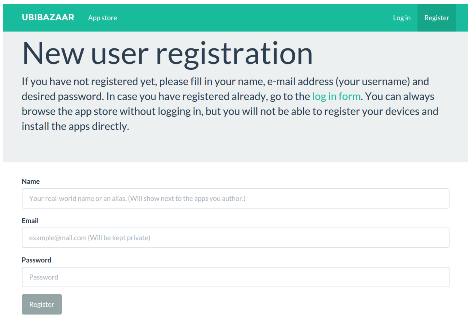
Log in

UbiBazaar 2015.

All the software implemented in the UbiBazaar project is released under Apache License Version 2.0 and all sources can be found on GitHub.

For more information on the UbiBazaar project, see project webpages on ubibazaar.github.io.

(a) Login dialog.


UBIBAZAAR App store Log in Register

New user registration

If you have not registered yet, please fill in your name, e-mail address (your username) and desired password. In case you have registered already, go to the log in form. You can always browse the app store without logging in, but you will not be able to register your devices and install the apps directly.

Name: Your real-world name or an alias. (Will show next to the apps you author.)

Email: example@mail.com (Will be kept private)

Password: Password

Register

(b) Registration dialog.

Figure 5.6: User login and registration dialogs

5.2 Architecture

As stated in the requirement 1.1 in section 3.2, the system should support future extensions for new platforms. To accommodate this the system was divided into separate modules so that it can be extended later on. Overview of the architecture is provided in figure 5.7.

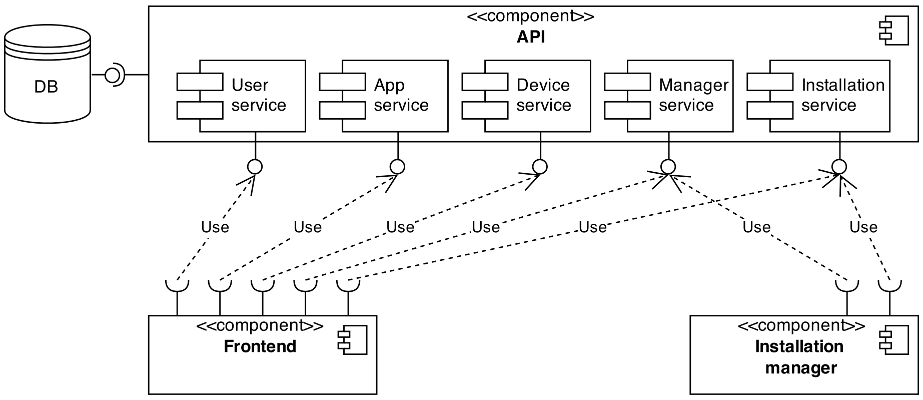


Figure 5.7: Proposed architecture.

In general, the three main types of components in the system are *a*) API *b*) Frontend *c*) Installation Manager.

We decided to fully separate the frontend from the API so new types of frontends can be implemented at a later stage, such as native mobile apps etc. This also means following the principle of loose coupling.

Domain model To back the ideas drawn in the proposed architecture we created a domain model that deals with the concepts that are to be used in system built in that architecture. A schematic diagram of the domain model can be seen in figure 5.8. Even though this diagram is oversimplified, we feel it shows the most important concepts and relationships among them.

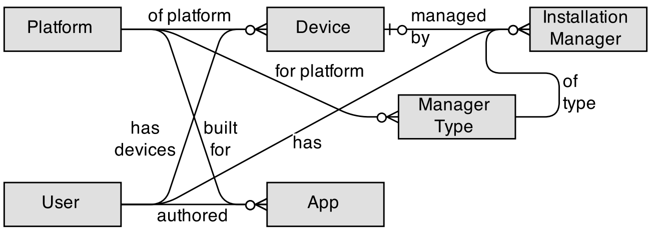


Figure 5.8: Domain model for the proposed architecture.

API is the component of the system that is responsible for the data. Typically, it deals with reading and storing data from a database and serves it to the other components. There typically is one API working over one database, but this setup might differ if the project is ever implemented as a distributed network of interconnected app stores. The API is responsible for keeping the data on users, apps, devices, installation managers and installations, each of them separately, following the principle of separation of concerns. Should there be any extension requiring to keep new types of data, these should be kept by the API. The API should also be equipped by security mechanisms for authentication and authorization over the kept resources. The API should be easy to integrate upon, and RESTful approach is preferable. An overview of the methods the API needs to provide can be found in paragraph 5.2.

Frontend is the component of the system that the users are interacting with to manipulate the data - registering themselves, adding new apps, registering their devices, requesting app installations onto devices. The component provides a user interface, a layer between the user and the API. It can take form of a web page, a smartphone app or a desktop application.

Installation manager is the component which manages installations of apps onto devices. It works with the API and makes sure that apps are installed on devices they are supposed to. It may or may not require user interaction, depending on the installation method. This might take form of a client residing on each individual device, or a server application, managing a number of devices remotely over the Internet, or even a smartphone app managing devices locally using BlueTooth.

5.3 Solution concept for Raspberry Pi

Even though the requirements in section 3.2 call for supporting both Raspberry Pi and Arduino deployments, we decided to only demonstrate implementation of one of them - Raspberry Pi (requirement 1.2). We will nonetheless describe in section 6.2 how Arduino deployment could be implemented and how the architecture is fit to support such extension, complying with requirement 1.3.

While the architecture proposed in section 5.2 is more of a general one, this section deals with architecture for a prototype solution that we propose, having Raspberry Pi in mind. The figure 5.9 gives an schema of the architecture when Raspberry Pi is considered to be the target platform. Design decisions backing this architecture are described in details below.

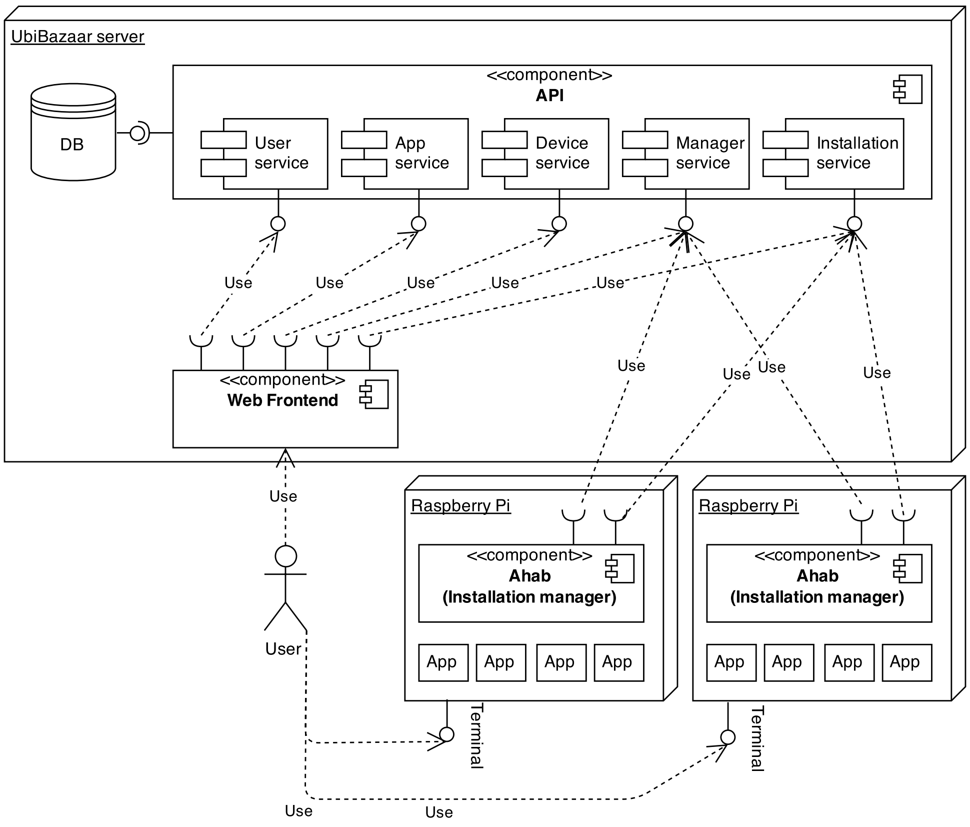


Figure 5.9: Architecture of prototype solution for Raspberry Pi.

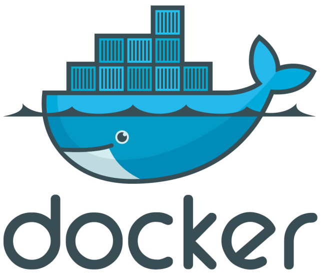
5.3.1 Design decisions

There were several design decisions made at this point and this section presents a brief overview of the rationale behind them.

Docker as the installation method As stated above, the main focus of this prototype is installing apps on Raspberry Pi devices. We decided to take the approach of dockerizing the apps and installing them as Docker containers. This will render some additional effort on the app makers, who needs to dockerize the app, but is in turn easy for the user to install such an app.

Manager-per-device strategy Using Docker Remote API [Docb] was chosen considered as a way how to manage apps installations on a device. Binding the Docker to a TCP socket and using the API remotely from a server service was briefly considered, but deemed inapplicable, as it would require the device to be reachable from the outer network, which might not fit many network setups in which IoT applications are used. It was decided that the system will be using the Docker Remote API locally from the device. For this to work, a client/manager needs to be running on that device interacting with both the Docker in that particular device and with the UbiBazaar API. This means that for each Raspberry Pi device, there needs to be this client installed. The relationship is therefore 1:1 between managers and Raspberry Pi devices.

(a) Captain Ahab fighting Moby Dick. Illustration by I. W. Taber. [Mel51]



(b) Docker logo. Source: [Doca]

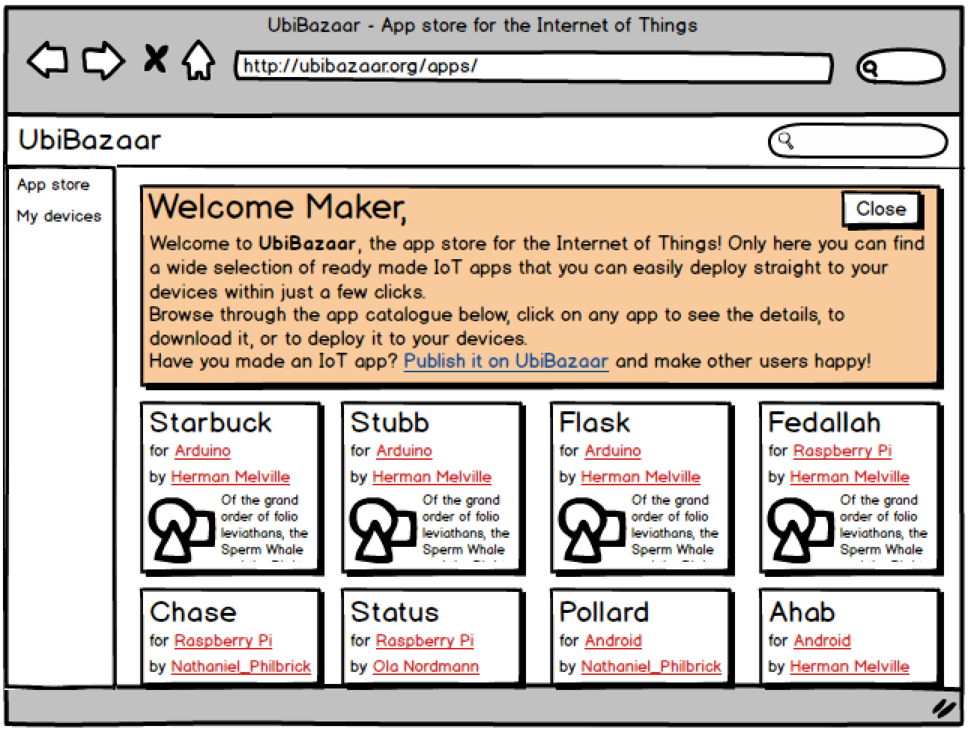
Naming the installation manager The name of installation manager for Raspberry Pi was decided to be *Ahab*, after the eponymous fictional captain of a whaling ship from Herman Mellville's masterpiece novel, Moby Dick [Mel51]. Captain Ahab is known for being mono-maniacally obsessed with Moby Dick, the most fierce whale known to man; the whale that deprived him of his leg and, later on, cost him his life. As the installation manager for Raspberry Pi is designed to build on top of Docker, and Docker has a whale in its logo[1], we find it fitting that the software would be named after the probably worlds best known whaler, despite his tragic destiny.

5.4 Paper prototype

As described in section 2.2, we created a paper prototype of the proposed system before implementing it. We did this to be able to validate our ideas quickly and have them evaluated externally.

To design a paper prototype, we have first come up with two user scenarios that represent the main tasks we expect the users to perform with the system, namely *a*) adding a new app *b*) registering a device *c*) installing an app. The full scenarios can be found in appendix B.

[1]The whale in the logo is in fact called *Moby Dock*, a pun on both Moby Dick and Docker. Source: https://blog.docker.com/2013/10/call-me-moby-dock/


UbiBazaar - App store for the Internet of Things

http://ubibazaar.org/apps/

UbiBazaar

App store

My devices

Welcome Maker,

Close

Welcome to **UbiBazaar**, the app store for the Internet of Things! Only here you can find a wide selection of ready made IoT apps that you can easily deploy straight to your devices within just a few clicks.
Browse through the app catalogue below, click on any app to see the details, to download it, or to deploy it to your devices.
Have you made an IoT app? Publish it on UbiBazaar and make other users happy!

Starbuck
for Arduino
by Herman Melville
Of the grand order of folio leviathans, the Sperm Whale

Stubb
for Arduino
by Herman Melville
Of the grand order of folio leviathans, the Sperm Whale

Flask
for Arduino
by Herman Melville
Of the grand order of folio leviathans, the Sperm Whale

Fedallah
for Raspberry Pi
by Herman Melville
Of the grand order of folio leviathans, the Sperm Whale

Chase
for Raspberry Pi
by Nathaniel_Philbrick

Status
for Raspberry Pi
by Ola Nordmann

Pollard
for Android
by Nathaniel_Philbrick

Ahab
for Android
by Herman Melville

Figure 5.11: Welcome screen of the paper prototype.

A clickable “paper” prototype was then designed in Balsamiq Mockups [Bal] that supported those scenarios. [2]

5.5 Software prototype

For all the components implemented in the scope of the software prototype, we used Java as the programming language of choice. This decision was made due to the previous experience with the Java platform and libraries and due to openness that this project aimed for in case this software prototype will be polished and becomes a reference implementation.

[2]See the clickable PDF prototype on https://www.dropbox.com/s/019iqa8nsorxerl/prototype_scenario_1.pdf?dl=0 (scenario 1) and https://www.dropbox.com/s/8pnc5exhzhpoqbr/prototype_scenario_2.pdf?dl=0 (scenario 2).

5.5.1 Database

It was decided to use a traditional Relational Database Management System (RDBMS) to persist the data in the system. This is to ensure integrity among the relations between individual entities we persist.

MariaDB When choosing a particular RDBMS implementation, we decided to use MariaDB [Mar], a community developer fork of popular MySQL. These two are however compatible and interchangeable, so this choice does not impose a strict limitation on eventual future work which would like to extend UbiBazaar yet use MySQL instead.

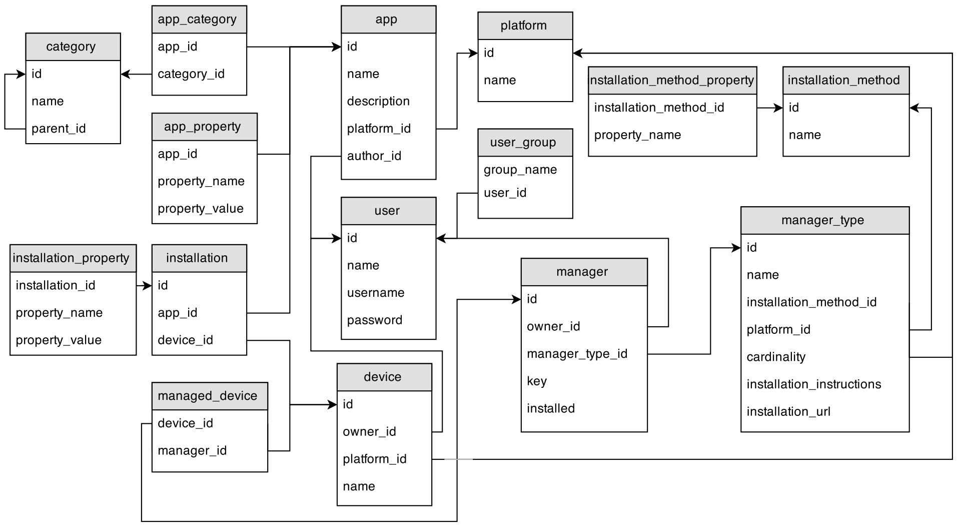


Figure 5.12: Database model of our implementation.

Database model The domain model, as described in paragraph 5.2 was revisited and a database model was derived from it, considering necessary linking tables. Diagram of the derived database model is shown in figure 5.12. The database model is using foreign keys heavily, as well as unique constraints and composite keys. We decided that each non-linking table will have a primary surrogate key (speaking the SQL language, the column is defined as `‘id‘ varchar(32) NOT NULL`) and natural keys such as username will only be considered unique keys.

5.5.2 Core library

Entities (such as App, Device, User, Installation and other) are used throughout all the components - API, frontend, installation manager - and as these components are all built in Java, we found it useful to extract these entities into a common library that then can be reused in all of the components.[3]

5.5.3 API

As the system is designed for reuse by new types of installation managers and possibly new types of frontends, our main concern was to implement the API in a way it is easy to extend, easy to implement and easy to use. Implementing it in RESTful manner, which is a de-facto standard in modern APIs, was a clear choice.[4]

Glassfish As this component is a server software, we decided to run it on a Java application server Glassfish 4, which is also a Java EE 7 environment and supports newest versions of APIs for servlets, JSON processing, Java API for RESTful Web Services (JAX-RS), Java Authentication and Authorization Service (JAAS), Java Database Connectivity (JDBC) as well as other features expected from an application server.

JAAS To secure the RESTful resources, we made use of the basic authentication method through the JAAS mechanism. JAX-RS has built-in support for JAAS and can inject security context to the secured resource. Declaration of secured resources is done in `web.xml` . The proposed system recognizes two types of subjects *a*) users *b*) installation managers. These differ not only in the privileges they should have in the API but also to the way they authenticate themselves. While users are authenticating with a username and password, installation managers authenticate with a unique ID and a generated random key. To accommodate this we implemented an own JAAS module that Glassfish can use. This also enabled us to hash passwords with an algorithm of our choice - with bcrypt.[5]

Maven UbiBazaar API is using Apache Maven [Apa] for build automation and dependency management. Maven follows the "convention over configuration" principle and makes it really easy to set up a project that has many dependencies without needing to download those and their dependencies manually.

JAX-RS To implement RESTful services easily, it was decided to make use of JAX-RS, namely of the version 2.0. The ease of implementing a RESTul service using this technoilogy is best illustrated in code listing 5.1.

[3]Source code and setup guide can be found on https://github.com/ubibazaar/core.
[4]Source code and setup guide can be found on https://github.com/ubibazaar/api.
[5]Source code and setup guide can be found on https://github.com/ubibazaar/jaas.

Listing 5.1: Example RESTful resource

```java
package org.ubicollab.ubibazaar.api.resources;

import javax.ws.rs.*;
import com.google.common.*;
import com.google.gson.*;

@Path("ping")
public class PingResource {

  @GET
  @Produces(MediaType.APPLICATION_JSON)
  public String ping() {
    return new Gson().toJson(ImmutableMap.of("pong",
        System.currentTimeMillis()));
  }

}
```

JDBC To keep the code free of property files, it was decided to set up database connection as a JDBC resource in the application server and let the application look it up by Java Naming and Directory Interface (JNDI) name in the application context. This way the datasources can be changed without needing to reconfigure and rebuild the application itself. For the simplicity of implementation and rather simple structure of our domain it was decided not to use any Object Role Modeling (ORM), even though the application server features support for Java Persistence API (JPA).

bit.ly To shorten long generated URLs, the system is using API of bit.ly, a popular URL shortening service. This becomes especially handy when users are attempting pairing of a device within the system and need to manually write a command in Raspberry Pi's terminal. The shorter the URL they are using, the quicker and the less error prone the process is. There is a RESTful API provided by bit.ly that UbiBazaar is using through the `StoreUtil` class.

5.5.4 Web frontend

From the initial idea to write a thick web app running in user's browser, we shifted to the traditional approach where all the application logic resides on the server and the user's browser barely renders HTML pages generated by the server. [6]

[6]Source code and setup guide can be found on https://github.com/ubibazaar/web.

When choosing a Java web framework to use for implementation, experience, simplicity and and community support (tutorials, samples, etc) were considered as important factors. It was decided to pick Play Framework [Typ], which has a rather simple templating engine (based on Scala, readable for Java-acquainted developers), support for routing, RESTful client libraries, keeping user sessions server-side, and many other features.

To give the generated HTML more appeal for the general public we make use of Bootstrap [Twi], which is a web front-end framework, consisting of CSS styles and JavaScript helpers that makes it really easy to style a web application. Using this framework and a theme Flatly [Par] styling the app into a modern looking one was a matter of minutes.

5.5.5 Ahab (installation manager)

This component is using Apache Maven the same way and for the same reasons described in paragraph 5.5.3.[7] It is using entities from the core library, described in subsection 5.5.2.

Ahab is a fairly simple component with a sole responsibility: ensuring that the device is running apps that is supposed to be running. That is Ahab installs and starts new containers user installs those through the frontend, and stops and remove containers as user uninstalls apps through the frontend. Ahab runs a thread every 20 seconds, checking API for apps that are supposed to be installed and then starts and stops containers so that they are in sync with what user expects.

Docker Remote API wrapper It was intended to implement a wrapper around the Docker Remote API [Docb], but it has turned out that there exist a number of already implemented libraries for this [Docc], and that the implementation of docker-client by Spotify[8] suits our needs perfectly.

Naming installation containers As stated above, Ahab needs to resolve which apps are running on a device in order to install those that should be installed, but are missing, and to stop those that are running, but should be removed. The implementation uses container names to identify which installation are they running. To ensure that Ahab does not interfere with other container that might be running on the same device, we name all the containers in a way that a common user would never name them in their right mind: using prefix `ubibazaar_inst_` followed by an installation UUID. UUID has length of 32 hexadecimal digits, having approximately $10^3 8$ unique combinations, entropy sufficient enough for all practical purposes.

[7]Source code and setup guide can be found on https://github.com/ubibazaar/ahab.
[8]Source code and documentation to be found on https://github.com/spotify/docker-client

Installation script and service Ahab is installed by a small installation script[9], that is generated by the API and contains a unique identifier of the installation manager as well as key for the installation manager to be able to authenticate against the API. This scripts downloads a release of Ahab, copies it to `/usr/local/ubibazaar` along with properties file containing the unique credentials, creates a service[10] in `/etc/init.d` and starts it. This service is started by system after rebooting, making sure that Ahab always runs.

[9]Source code of the installation script can be found on https://github.com/ubibazaar/ahab/blob/master/installation_script.sh. Make sure to replace placeholders in mustaches by real values.

[10]Source code for the init.d script can be found on https://github.com/ubibazaar/ahab/blob/master/ahab

Chapter 6 Evaluation

In the scope of the project we carried out 3 evaluations that evaluate the proposed solution both theoretically and practically. This chapter presents the description of the evaluations and their results.

1. **Prototype evaluation with focus group** has been performed as a focus group evaluation with focus on the usability of the paper prototype and its workflow.

2. **Extending to support Arduino deployment** has been performed as theoretical feasibility evaluation to conceptually assess the fitness of the proposed architecture to requirements.

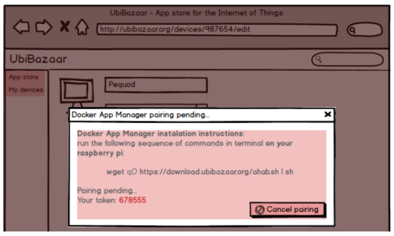
3. **Dockerizing CoSSMic** has been performed as a practical feasibility evaluation, to evaluate whether the selected deployment method is applicable in the IoT context.

6.1 Prototype evaluation with focus group

To evaluate usability of the paper prototype and validate our requirements with unbiased participants, we held a focus group meeting, where participants were presented the concept of the project and the scenarios, and shown the way through the paper prototype. Plenty of important feedback was collected during the session and many good points were made that needed to be reconsidered before we proceed to create a software prototype.

From the evaluation we found out we need to make the prototype easier to use (ease the terminology and simplify device registration mechanism) and to improve our functional requirements — stress social perspective, and address trust and security concerns.

Device pairing mechanism The original mechanism of device pairing was counting on downloading a generic installer, installing the software and then configuring it manually with unique credentials for accessing the API. The focus group found this too complicated for the user and suggested to simplify the process even more. Several methods were suggested, some of them applicable in the context (using MAC address, downloading installer with credentials built-in) and some not (using QR codes). For the software prototype, we decided to generate the installation script with built-in credentials so it can auto-configure itself. The difference can be seen in figure 6.1.

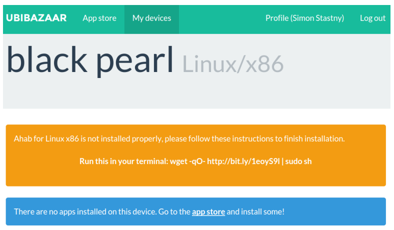

UbiBazaar - App store for the Internet of Things
http://ubibazaar.org/devices/987654/edit
UbiBazaar
App store
My devices
Pequod
Docker App Manager pairing pending...

Docker App Manager instalation instructions:
run the following sequence of commands in terminal on your raspberry pi:

```
wget qO https://download.ubibazaar.org/ahab.sh | sh
```

Pairing pending...
Your token: 678555

(a) Paper prototype.



Ahab for Linux x86 is not installed properly, please follow these instructions to finish installation.

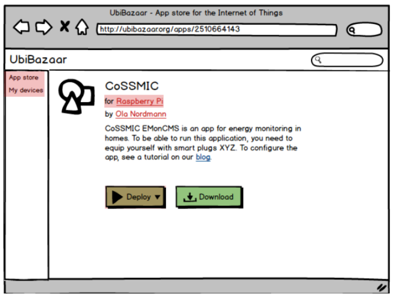
Run this in your terminal: wget -qO- http://bit.ly/1eoyS9l | sudo sh

There are no apps installed on this device. Go to the app store and install some!

(b) Software prototype.

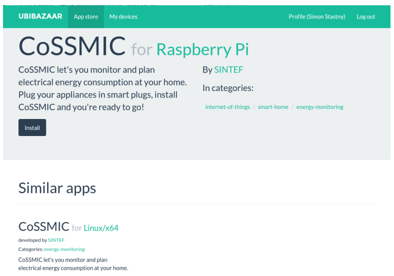
Figure 6.1: Pairing instructions in prototypes.

Language It was quickly discovered that the terminology/language used in the paper prototype is complicated and while it may (or may not) be technically correct, it does not meet the needs of common users who expect simplicity. While we have used the terms "deploy" and "deployment" up to this point, it turns out that users are more acquainted with terms "install" and "installation" which represent the same things from their point of view. Having "download" was also considered confusing and not useful, so it was removed.


UbiBazaar - App store for the Internet of Things
http://ubibazaar.org/apps/2510664143
UbiBazaar
App store
My devices
CoSSMIC
for Raspberry Pi
by Ola Nordmann

CoSSMIC EMonCMS is an app for energy monitoring in homes. To be able to run this application, you need to equip yourself with smart plugs XYZ. To configure the app, see a tutorial on our blog.

(a) Paper prototype.



CoSSMIC let's you monitor and plan electrical energy consumption at your home. Plug your appliances in smart plugs, install CoSSMIC and you're ready to go!

By SINTEF

In categories:

internet-of-things / smart-home / energy-monitoring

developed by SINTEF

Categories: energy-monitoring

CoSSMIC let's you monitor and plan electrical energy consumption at your home.

(b) Software prototype.

Figure 6.2: App detail in prototypes.

Social perspective A clear point was made that app stores are not bare catalogs of apps, but also a social platform, comprising of features such as ratings and comment sections. This social perspective should not be underestimated, as their it forms an integral part of what app stores really are - an exchange among creators and users. Creators providing software to users, users providing feedback to creators and other users. For users, it is not only the official description of the app what is important, but also how other users find the app useful and usable. This is also one of the factors that can establish trust of users against the resources in the app store.

Trust and security concerns The trustworthiness in IoT app stores have been brought up by previous research [KPX+14] and was also a big concern of the focus group. IoT has a potential to influence our daily lives in ways we may not even be able to imagine yet, and it is absolutely necessary to ensure that IoT apps, many of which will handle our personal data and control our lives to a certain extent, are to be trusted and prevent misuse by third parties.

6.2 Extending to support Arduino deployment

To evaluate how requirements 1.2 (Arduino deployment) and 1.3 (extensibility for new platforms), were handled by our solution, we decided to perform a theoretical feasibility evaluation. To do this, we would like to conceptually show how Arduino deployment could be implemented within the proposed system architecture.

As mentioned in section 4.5, NTNU students in the scope of the µC Software Store project [EST13], developed an Android application with the capability of programming an Arduino device wirelessly. We believe this approach can serve as an inspiration for designing a UbiBazaar extension for Arduino, as well as the code can be reused by a reference implementation in the future.

The figure 6.3 shows how this extension can fit into the proposed architecture. The extension in question would be a native Android app which would assume roles of both the frontend and the installation manager, as proposed in figure 5.7. This means the responsibilities of the app would be to communicate with the API as well as install the apps to the device physically, using the over-the-air deployment functionality implemented in µC Software Store.

We believe this solution fits perfectly into the proposed architecture and can be implemented easily as a future continuation of the UbiBazaar project, as it can reuse the code developed in µC Software Store [EST13].

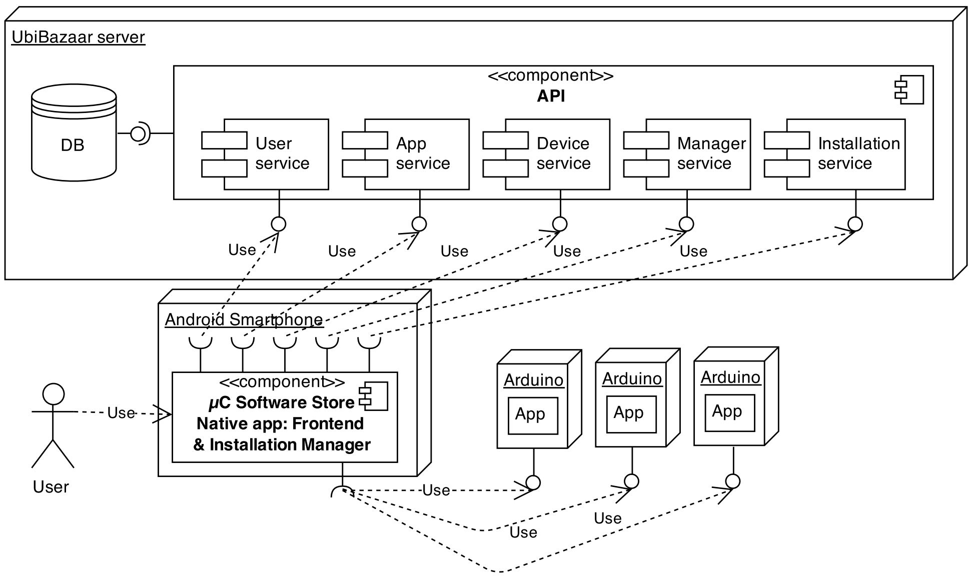


Figure 6.3: Support of Arduino support in the proposed architecture.

6.3 Dockerizing CoSSMic for Raspberry Pi

We decided to perform a practical feasibility evaluation of the selected deployment method — using Docker on Raspberry Pi — selected to address requirement 1.2. This evaluation helps us to assess whether the Docker approach is feasible for IoT apps on Raspberry Pi.

To perform the evaluation we decided to take an existing IoT app and install it on a real Raspberry Pi device using our prototype of installation manager. This section presents our findings from this evaluation and describes what challenges we faced and how have we overcome them.

We chose to do this experiment with an IoT app which is a part of the CoSSMic project. The reasons to choose this app are threeefold *a*) it is a real, existing IoT app *b*) this app targets Raspberry Pi devices *c*) the CoSSMIC project has a planned field study which could be a good test case. The CoSSMic project assumes deployment of a larger number of these devices, but the experiment was only concerned about installation of the app to an individual device and does not cope with the integration issues.

As described in 4.3 being able to do this first requires us to dockerize the app. This means writing a Dockerfile, formally describing all the necessary steps to be taken for the app to be installed. Under normal circumstances this would be created from scratch building on the knowledge about needed dependencies, etc.

In our case, we had an almost ready-made Dockerfile that dockerized the app on top of `debian:wheezy` [1] base image, which is an image of a 64-bit version of Debian, a Linux distribution.

As mentioned in 4.3, a Docker image can only run on a device whose processor architecture matches the architecture for which the image was built. Therefore we needed to change the base image to one that would work on Raspberry Pi. At the time evaluating, there were several possible base-images that could have been used:

1. `resin/rpi-raspbian` [2]
2. `cellofellow/rpi-arch` [3]
3. `mazzolino/armhf-ubuntu` [4]

To minimize the effort needed to adjust the Dockerfile to a new base image, we decided to choose `resin/rpi-raspbian` as a base image. This docker image of Raspbian which is closely based on Debian, would allow us to reuse most of the existing Dockerfile. Some changes were however still necessary:

1. Dropbox daemon was failing intermittently on Raspbian, and was removed from the Dockerfile as unnecessary for the demonstration purposes.

2. Installer for mysql tried to launch a prompt during the installation, asking for a password to set for the root user on the database. As building a docker image needs to be non-interactive, we fixed this by using `debconf-set-selections` to set the password when the installers prompts for it.

3. Configuration files for lighttpd, a web server, slightly differed between the version for Debian and for Raspbian, most notably in path used as a document root. While the Debian version has default document root in `/var/www`, the Raspbian version has it set to `/var/www/html`. This was however very easy to carry out in the lighttpd configuration file `/etc/lighttpd/lighttpd.conf`.

[1] https://registry.hub.docker.com/_/debian/
[2] https://registry.hub.docker.com/u/resin/rpi-raspbian
[3] https://registry.hub.docker.com/u/cellofellow/rpi-arch
[4] https://registry.hub.docker.com/u/mazzolino/armhf-ubuntu

Building the Docker image has proven to be a very time consuming task. There are two main factors to this:

1. Building a docker image is fail-fast. Build stops on first encountered problem with the Dockerfile, which only makes it possible to eliminate problems one by one by trial and repeat.

2. Building the docker image on Raspberry Pi turned out to be extremely inefficient and was a matter of hours (typically 3-5, occasionally 8 for no clear reason).

Building a docker image is typically not a time-consuming task, but it proved to be in our case. This was largely caused by the low performance of the Raspberry Pi device, but also due to the fact that this particular Dockerfile consisted of over 70 commands, each of them creating another layer of the container.

Building the original Dockerfile (i.e. the one basing on for `debian:wheezy`) was a matter of minutes so we have a reason to believe low performance of the Raspberry Pi is to blame.

Further optimization of the Dockerfile to reduce the number of steps is advisable, however detailed benchmarks were not performed and this is just a conjecture of our observations. This was also observed and described by other such as [Toe15], who tried to address this issue by building the image outside Raspberry Pi: “I used the emulator QEMU to emulate the Raspberry Pi on a fast Macbook. But, because of the inefficiency of the emulation, it is just as slow as building your Dockerfile on a Raspberry Pi. I tried cross-compiling. This wasn’t possible, because the commands in your Dockerfile are replayed on a running image and the running Raspberry-pi image can only be run on... a Raspberry Pi.” [Toe15]

The built docker image was uploaded to Docker Hub into repository `simonstastny/rpi-cossmic`[5] and can be used directly without needing to build it again. Once there, it was added to a running instance of UbiBazaar and deployed onto a real Raspberry Pi device. Due to the size of the Docker image, this has taken a couple of minutes, but the installation was successful at last.

[5]https://registry.hub.docker.com/u/simonstastny/rpi-cossmic/

6.4 Overview of covered requirements

This section provides an overview of how our requirements were covered by the reviewed state of the art technology and solutions. We chose to focus on the deployment method perspective and usability of the app store functionality.

Rq	Evaluation	Result
3 1.2	Usability evaluation: *Prototype evaluation with focus group*	Requirements 3.4 added. Improved prototype's language and device pairing mechanism.
1.1 1.3	Theoretical feasibility evaluation: *Extending to support Arduino deployment*	Extensibility and Arduino support feasible with little effort to be expected.
1.2	Practical feasibility evaluation: *Dockerizing CoSSMic*	Approach feasible, yet minor performance issues were encountered with the example app.

Table 6.1: Evaluation of the prototypes according to requirements.

Chapter 7 Discussion

This chapter provides an overview on known limitations of our project and possible future work that can be done as an extension of the project.

7.1 Limitations of the current solution

Survey breadth The survey we have performed has not met our expectations, as only 12 responses have been collected, and only one participant was interviewed subsequently. We feel the survey should have been performed even broader, asking more maker communities, in order to collect more responses and have more data to base our work upon.

Prototype completeness To propose a solution and create a viable prototype of the product in a timely fashion, the scope of the project was limited to only a subset of the requirements. We have focused on those that are most important to our contribution — deployment of apps to Raspberry Pi (requirement 1.2), extensibility to other platforms (requirement 1.1) and most of the app store functionality (requirement 3). Implementation of other requirements — such as deployment to Arduino and resolving device capabilities — was considered to be out of scope of this project and might be carried out as a future extension of the project. This is to be discussed in section 7.2.

While we have implemented adding new apps to demonstrate requirement 3.2, we have not gone the whole way to develop the whole life-cycle of an app. The prototype therefore does not support removing an existing app, updating it or adding new updates to it (requirement 3.2.1). Even though they are essential to the app store if it becomes a mature product, they were not essential to a prototype which sole purpose was to demonstrate deployment of an app to a device.

The search over the catalog (requirement 3.3.3) was not implemented as it was deemed unnecessary for the prototype, which only features a few of apps at maximum.

The social perspective of the app store — user feedback features from requirement 3.4 were also not implemented in the scope of the project as we focused on the technicalities of device registration and app installation instead.

Docker on Raspberry Pi performance issues The evaluation in section 6.3 showed that building a Docker image on a Raspberry Pi can be a tedious task as the performance of the device does not allow to build the image fast. In practice this means building a Docker image can easily take hours, which is not ideal for the app developer who wants to add a new app to UbiBazaar and might not want to wait this long.

Evaluation completeness A usability evaluation with real users (makers) has not been performed withing the scope of this project. Such one should be carried out to truly evaluate the usability of the prototypes for the users.

7.2 Future work

section 3.2 drawn many requirements posed on the system, however many of them were not implemented in the scope of the project. These requirements can server as a basis for future extensions of the UbiBazaar project.

Arduino deployment One of the possible extensions, support for Arduino deployment (requirement 1.3), was conceptually followed and evaluated in section 6.2. It was assessed that this extension can built on top of the existing architecture and should be rather easy to implement.

Social aspect Another requirement that was not followed further was the social aspect of the app store: discussions, rating, etc. This could be a good extension of the project but was out of our focus, which was more on the technical perspective of the deployment mechanisms.

Device capabilities The last set of requirements that we have not carried out was requirement 2 and its sub-requirements: exploring devices' capabilities and matching them against app requirements to resolve compatibility. This was again out of the narrow focus of our project, but would be an essential requirement for the final product and a good extension of our project.

Other possible extensions Extending the app store is of course possible in many other ways, starting with finishing the core functionality to support full life-cycle of apps and devices, to adding features such as bulk-deployment, automatic app update mechanism in installation managers, or adding support for new platforms.

Open source All the source code is released under Apache Software License 2.0 and publicly available at https://github.com/ubibazaar, where everyone is welcome to join the project, contribute to it and discuss ideas.

7.3 Conclusion

Our project identified that the world of makers and IoT was missing a distribution channel for software, collected relevant information through surveys using questionnaires and interviews, derived requirements from these findings, proposed a solution and implemented a working prototype addressing those.

The prototype supports many of the essential requirements that we set, but needs to be extended, completed and polished before it can become a final product that could be used in daily lives by makers and users.

As such, we consider the project to be successful, and hope to start this as an open source project at https://github.com/ubibazaar and get the maker community involved in contributing to it.

References

[And00] Jesper Andersson. A deployment system for pervasive computing. In *Proceedings International Conference on Software Maintenance ICSM-94*, pages 262–270. IEEE Comput. Soc. Press, 2000. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=883058http://www.scopus.com/inward/record.url?eid=2-s2.0-0034505884&partnerID=tZOtx3y1.

[Apa] Apache Software Foundation. Apache Maven. URL: https://maven.apache.org/.

[Bal] Balsamiq Studios, LLC. Balsamiq Mockups. URL: https://balsamiq.com/products/mockups/.

[BCFP14] Paolo Bellavista, Antonio Corradi, Luca Foschini, and Alessandro Pernafini. Towards an Automated BPEL-based SaaS Provisioning Support for OpenStack IaaS. *Scalable Computing: Practice and Experience*, 14(4):235–247, January 2014. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-84893475267&partnerID=tZOtx3y1, doi:10.12694/scpe.v14i4.930.

[Can] Canonical Ltd. Snappy Ubuntu. URL: http://www.ubuntu.com/cloud/tools/snappy.

[Can15] Canonical Ltd. ARM/RaspberryPi - Ubuntu Wiki, 2015. URL: https://wiki.ubuntu.com/ARM/RaspberryPi/.

[Cha] Sitaram Chamarty. git as a deployment tool. URL: http://gitolite.com/deploy.html/.

[Dav11] Nigel Davies. Beyond Prototypes, Again. *IEEE Pervasive Computing*, 10(1):2–3, January 2011. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-78650879022&partnerID=tZOtx3y1, doi:10.1109/MPRV.2011.2.

[Doca] Docker, Inc. Docker - Build, Ship and Run Any App, Anywhere. URL: https://www.docker.com/.

[Docb] Docker, Inc. Docker Remote API - Docker Documentation. URL: https://docs.docker.com/reference/api/docker_remote_api/.

[Docc] Docker, Inc. Docker Remote API Client Libraries. URL: https://docs.docker.com/reference/api/remote_api_client_libraries/.

[Docd] Docker, Inc. What is Docker? An open platforms for distributed apps. URL: https://www.docker.com/whatisdocker/.

[Doc15] Docker, Inc. Dockerfile - Docker Documentation, 2015. URL: https://docs.docker.com/reference/builder/.

[EST13] Jeppe Eriksen, Wilhelm Walberg Schive, and Robin Tordly. µ C Software Store Project Report. 2013.

[GVAGM10] Charles Gouin-Vallerand, Bessam Abdulrazak, Sylvain Giroux, and Mounir Mokhtari. A Software Self-Organizing Middleware for Smart Spaces Based on Fuzzy Logic. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 138–145. IEEE, September 2010. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-78149320999&partnerID=tZOtx3y1, doi:10.1109/HPCC.2010.107.

[Her] Heroku. Heroku Dev Center - Deploying with Git. URL: https://devcenter.heroku.com/articles/git.

[HM06] Didier Hoareau and Yves Mahéo. Middleware support for the deployment of ubiquitous software components. *Personal and Ubiquitous Computing*, 12(2):167–178, November 2006. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-38349165556&partnerID=tZOtx3y1, doi:10.1007/s00779-006-0110-7.

[Hyp15] Hypriot. Getting started with Docker on your Raspberry Pi, 2015. URL: http://blog.hypriot.com/getting-started-with-docker-on-your-arm-device/.

[Ind15] IndieCity. Games & Apps - Pi Store, 2015. URL: http://store.raspberrypi.com/projects.

[KC03] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-0037253062&partnerID=tZOtx3y1, doi:10.1109/MC.2003.1160055.

[KK10] Gerd Kortuem and Fahim Kawsar. Market-based user innovation in the Internet of Things. *2010 Internet of Things (IOT)*, pages 1–8, November 2010. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5678434, doi:10.1109/IOT.2010.5678434.

[KPX$^+$14] Kai Kang, Zhibo Pang, Li Da Xu, Liya Ma, and Cong Wang. An Interactive Trust Model for Application Market of the Internet of Things. 10(2):1516–1526, 2014.

[Mar] MariaDB Foundation. MariaDB An enhanced, drop-in replacement for MySQL. URL: https://mariadb.org/.

[Mel51] Herman Melville. *Moby-Dick; or, The Whale.* Harper & Brothers, 1851.

[MM07] Nenad Medvidovic and Sam Malek. Software deployment architecture and quality-of-service in pervasive environments. In *International workshop on Engineering of software services for pervasive environments in conjunction with the 6th ES-EC/FSE joint meeting - ESSPE '07*, pages 47–51, New York, New York, USA, 2007. ACM Press. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-41149153300&partnerID=tZOtx3y1, doi:10.1145/1294904.1294911.

[MM12] Dejan Munjin and Jean-Henry Morin. Toward Internet of Things Application Markets. *2012 IEEE International Conference on Green Computing and Communications*, pages 156–162, November 2012. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6468308, doi:10.1109/GreenCom.2012.33.

[New14] Joshua Newell. Make: DIY Arduino Bluetooth Programming Shield, 2014. URL: http://makezine.com/projects/diy-arduino-bluetooth-programming-shield/.

[Oat06] Briony J Oates. *Researching Information Systems and Computing.* Sage Publications Ltd., 2006.

[Par] Park, Thomas. Flatly - Flat and modern. URL: https://bootswatch.com/flatly/.

[Ras] Raspberry Pi Foundation. Ubuntu Snappy Core - Raspberry Pi. URL: https://www.raspberrypi.org/downloads/snappy/.

[Ras14] Raspberry Pi Foundation. What is a Raspberry Pi, 2014. URL: http://www.raspberrypi.org/help/what-is-a-raspberry-pi/.

[Red] Red Hat. OpenShift Developers - Deployments and Rollbacks. URL: https://developers.openshift.com/en/managing-deployments.html.

[Res13a] Resin.io. Docker on Raspberry Pi, 2013. URL: https://resin.io/blog/docker-on-raspberry-pi/.

[Res13b] Resin.io. How does resin.io work?, 2013. URL: https://resin.io/how-it-works/.

[Res15] Resin.io. Resin.io Documentation - Supported Devices, 2015. URL: http://docs.resin.io/#/pages/hardware/devices.md.

[SCBZ11] Reidar M. Svendsen, Humberto N. Castejon, Erik Berg, and Josip Zoric. Towards an integrated solution to Internet of Things - a technical and economical proposal. In *2011 15th International Conference on Intelligence in Next Generation Networks*, pages 46–51. IEEE, October 2011. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-83455224733&partnerID=tZOtx3y1, doi:10.1109/ICIN.2011.6081101.

[SKS05] Byung Y. Sung, Mohan Kumar, and Behrooz Shirazi. Flexible and Adaptive Services in Pervasive Computing, 2005. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-33645956199&partnerID=tZOtx3y1, doi:10.1016/S0065-2458(04)63005-1.

[Sta14] Simon Stastny. Specialization Project Report: UbiBazaar - App Store for the Internet of Things. 2014.

[Toe15] Jan Toebes. Fool-Proof Recipe: Docker on the Raspberry Pi, 2015. URL: https://www.voxxed.com/blog/2015/04/fool-proof-recipe-docker-on-the-raspberry-pi/.

[Twi] Twitter, Inc. Bootstrap · The world's most popular mobile-first and responsive front-end framework. URL: http://getbootstrap.com/2.3.2/.

[Typ] Typesafe Inc. Play Framework - Build Modern & Scalable Web Apps with Java and Scala. URL: https://www.playframework.com/.

[UH88] GL Urban and E Von Hippel. Lead user analyses for the development of new industrial products. *Management science*, 1988. URL: http://pubsonline.informs.org/doi/abs/10.1287/mnsc.34.5.569.

[ZWH$^+$06] Di Zheng, Jun Wang, Weihong Han, Yan Jia, and Peng Zou. Towards A Context-Aware Middleware for Deploying Component-Based Applications in Pervasive Computing. In *2006 Fifth International Conference on Grid and Cooperative Computing (GCC'06)*, pages 454–457. IEEE, 2006. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-38649143208&partnerID=tZOtx3y1, doi:10.1109/GCC.2006.92.

Appendix A

API Documentation

This appendix gives an overview of the RESTful methods the API provides.

resource	path	query params	method	result
apps	/	—	GET	returns all apps
apps	/{id}	—	GET	returns app with the specified id
apps	/query	category, platform, author	GET	all apps satisfying the query criterion (or the conjunction of criteria, if more than one is specified
apps	/	—	POST	creates a new app and returns its URI
apps	/{id}	—	PUT	updates the app with the specified id
apps	/{id}	—	DELETE	deletes the app with the specified id
categories	/	—	GET	returns tree of all categories
categories	/{id}	—	GET	returns subtree of the specific category
platform	/	—	GET	returns all platforms
platform	/{id}	—	GET	returns the platform with the specified id

Table A.1: API: Methods provided by App service

[1] full resource name: installation_methods
[2] full resource name: manager_types

resource	path	query params	method	result
installations	/	—	GET	returns all installations of the logged-in user
installations	/{id}	—	GET	returns the installation of the logged-in user with the specified installation id
installations	/query	app, device	GET	all installations of the logged-in user satisfying the query criterion
installations	/	—	POST	creates a new installation and returns its URI
installations	/{id}	—	PUT	updates the installation with the specified id
installations	/{id}	—	DELETE	deletes the installation with the specified id

Table A.2: API: Methods provided by Installation service

resource	path	query params	method	result
devices	/	—	GET	returns all devices of the logged-in user
devices	/{id}	—	GET	returns the device of the logged-in user with the specified device id
devices	/query	platform	GET	all devices of the logged-in user satisfying the query criterion
devices	/	—	POST	creates a new device and returns its URI
devices	/{id}	—	PUT	updates the device with the specified id
devices	/{id}	—	DELETE	deletes the device with the specified id

Table A.3: API: Methods provided by Device service

resource	path	query params	method	result
managers	/	—	GET	returns all managers of the logged-in user
managers	/{id}	—	GET	returns the manager of the logged-in user with the specified manager id
managers	/query	type, device	GET	all managers of the logged-in user satisfying the query criteria
managers	/	—	POST	creates a new manager and returns its URI
managers	/{id}	—	PUT	updates the manager with the specified id
managers	/{id}	—	DELETE	deletes the manager with the specified id
i.m. [1]	/	—	GET	returns all installation methods
i.m.	/{id}	—	GET	returns subtree of the specific category
m.t. [2]	/	—	GET	returns all manager types
m.t.	/{id}	—	GET	returns the manager type with the specified id
pairings	/{id}	—	POST	links the device in the body with the manager by id

Table A.4: API: Methods provided by Manager service

resource	path	query params	method	result
users	/{id}	—	GET	returns the user with the specified id
users	/query	username	GET	returns the user with the specified username
users	/	—	POST	creates a new user and returns its URI

Table A.5: API: Methods provided by User service

Appendix B
Scenarios

This appendix presents scenarios as they were designed for the paper prototype. The prototype and the scenarios were then subject to focus group evaluation described in section 6.1. It is worth noting these scenarios represent the workflow and terminology used in the paper prototype only. As there were important findings regarding the usability of the paper prototype, after which we revised the workflow and terminology before proceeding to the software prototype, these scenarios do not represent the workflow of the final software prototype.

B.1 Scenario 1

1. Registers as a user
2. Logs in
3. Goes to `Apps`
4. Clicks on `Add new app`
5. Fills in `CoSSMic` as a name
6. Selects `Raspberry Pi` as a platform
7. Writes app description in the textbox
8. Fills in `cossmic/emoncms` as Docker Hub repository
9. Clicks on `Save`
10. Finds himself on app overview page
11. Clicking on `Edit` button gets him back to editing app details

B.2 Scenario 2

1. Registers as a user
2. Logs in
3. Goes to `Apps`
4. Filters on `Raspberry Pi`
5. Clicks on `CoSSMic`
6. Clicks on `Deploy`
7. Clicks on `Add new device`
8. Fills in `Pequod` as a name
9. Selects `Raspberry Pi` as a platform
10. Selects `Docker App Manager`
11. Follows installation instructions
12. Goes to `Apps`
13. Filters on `Raspberry Pi`
14. Clicks on `CoSSMic`
15. Clicks on `Deploy`
16. Selects `Pequod` device
17. Finds himself on device overview, seeing the app deployed on the device