GUI infused IoT Architecture design and implementation

Master's Thesis
University of Turku
Faculty of Technology
Communication Systems
2022
M.Sc. Tijam Moradi
Reviewed by:
Prof. Isoaho Jouni
Dr. Antti Hakkala

UNIVERSITY OF TURKU
Faculty of Technology
MORADI, TIJAM Thesis
Master's Thesis, 64 pages
DP in Information and Communication Technology
February 2022
The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service

---

In this thesis a system architecture that can facilitate graphical user interfaces (GUIs) in in IoT framework is designed. The desired result is the above with remote management, remote controllability, flexibility and extendability. The thesis aims to answer whether building a system like this is possible, feasible, efficient and relatively easy to achieve, and also takes and approaches the productisation of the system via a case study.

Thesis case provides purpose and an area of research. The case involves building a tailorable apartment staircase digital signage, which has remote administration and controllability options. This case is a startup by nature, and the whole product is designed by the writer. Thesis case targets apartment environments. Real life issues, such as notification board management, has inspired this project.

Thesis research has a heavy influence of empiric methods in comparison to theoretical ones. Since the case solves a very practical problem and the scope is the development and productisation phase, there were not many traditional measuring methods. However, positive results were established by the conclusion of the thesis. Feasibility and sustainability of the system were the main measures, which compared the architecture design to execution results.

Upon these positive results, the thesis concludes that the described system is feasible and sustainable to build. However, alternate methods might have been more successful. The conclusion furthermore suggests that the case is achievable and fruitful with the methods used.

Keywords:

GUI, IoT, Edge, cloud, Microsoft, Azure, apartment, digital signage, remote management, productisation

Contents

1 Introduction

1.1 Background

The thesis gets a base foundation from personal experiences with apartment management. Apartment management currently in Finland is rather traditional, manual and essentially not digitalised. Society is relying more on digital services than ever before. [1] Apartment environments generally could benefit more from digitalisation advantages in comparison to what is currently offered. For example, notification deliverance lacks integrity. Most apartment staircases use physical boards, which's information is often outdated.

According to N. Moore, a modern society lives heavily in digital domain [2], thus physical notification boards and traditional apartment service offerings are becoming a bottleneck in terms of quality of life. Other shortcomings that I have noticed include local interconnectivity. In comparison to a few decades back, the role of apartment management has changed. The role change is affected by post industrialist information explosion. [3] For example, urban living depends a lot more on public transportation than before, which reflects to services that an apartment could offer for its residents. In my point of view, apartments should be able to provide services of this kind to all audiences; busy young professionals, elderly with no information technology skills, people with disabilities and all those that reside somewhere in the middle. By offering a digital modern platform for apartment managers, apartments could potentially bring more value for the residents via improved communication, accessibility and integration to local services. Evidently these features ease the operational quality for both, the residents and the managers.

I run a server at my home with a public address. I have several applications running on it that are accessible through internet, such as web sites of some companies I have an agreement with. Running a server home has connectivity risks that I was aware of. However, some time ago my apartment's board of directors failed to notify properly of a construction that would temporarily cut cable connection. As the construction took place, my server obviously went down without reconnecting, since the router had to be rebooted. Inconveniently, I was not prepared nor physically near the server at the time of the temporary network outage.

This above experience led to a thought, whether the problem could have been avoided with minimal effort. At the time of writing, great majority of apartments in Finland still use physical notification boards and resident lists. Often times it is found difficult and definitely far from instant to put up these notifications. What if the system was to be digitalised, like a couple of researched apartments have already done at some level. These systems can give valuable information for residents about multitude of things, such as upcoming noise from maintenances. The systems are also able to facilitate accessibility options, such as usability for people with disabilities like blindness. Digital accessibility in apartments is not yet publicly noted much in Finland, however I personally anticipate this to happen in near future. [4] How far can system extensions be taken? Similar to a smart home approach, could the platform be extended to have multiple elements, like elevator integrations and live bus stop schedules?

1.2 Thesis structure and objectives

The thesis consists of eight chapters, during which it takes a research orientated approach alongside an empiric experiment in regards to the topic. Thesis will go through designing an architecture for a desirable infotainment system that takes advantage of IoT technologies. Thesis covers the design, architecture, technology choices, development, production with quality assurance, extendability and conclusions for such solution. Project included in the thesis is named ApartaView. With the Apartaview case, thesis aims to answer how feasible and sustainable it is to build such a system. It also aims to evaluate how digitalisation can bring more value to modern apartment management and its residents. Thesis takes a productisation approach with this system. Assuming the developed system can be turned into a plausible product, thesis concludes with success. Questions are mostly tackled via empiric research.

Writing of the thesis was supported with a guide from Michael Jay Katz, which shed light on transforming a research to a manuscript [5]. Nature of the thesis is case orientated, thus using the guide required adaptation and conformation. Thesis was written and produced with the help of LaTeX, a macro package for typesetting system TeX. LaTeX usage was supported by a guide from Tobias Oetiker [6].

After introduction to the theme and shedding some light on the background the second chapter covers the ingredients that are required for the described solution. It goes through some services that are available, since building a maintainable solution from the ground up is too complex and unnecessary for the scope. The project aims to take use of cutting edge technologies in order to streamline development and production. Subsequently, thesis will analyse what parts of such infrastructure are already available as services or products, and, which require custom implementation. The technology possibilities are discussed at software and hardware level.

Before selecting technologies that were mapped in the second chapter, the scope of the project must be defined. The third chapter will give a bridge between available technologies to implementation by defining the case for this study. Case discussion also sheds some light on where else could a system similar to the case be implemented in – i.e. environment adaptation.

At this point, it should be more or less clear which technologies are available for utilisation and what does the target scope and environment propose. With this information, it time to proceed to designing the architecture in chapter 4. Architecture design starts with a general model, from which it is adapted to a specific model utilising selected technologies from those ones that were reviewed in the second chapter.

Naturally, after a proper design, it is finally the time for implementation. Implementation execution is discussed and described in the fifth chapter. The chapter covers the implementation on all aspects, but focuses on major obstacles along the execution. This is due to the fact that the project's development work is considerably too large in order to be described comprehensively at detailed feature level. Excluding libraries and ready made tools, the project base already exceeds 10 000 lines of code at proof of concept (PoC) stage. Since the thesis overall aims to cover common mistakes and technology specific limitations, this execution chapter will align to it on best effort basis. This is done by prioritising engineering challenges of the project in regards to the scope.

Since the project is startup by nature, it alludes that design and development is conveyed by a need. In context translation, this implicates that the product being built solves a refined problem. Henceforth, it is necessary to keep productisation in mind at all stages. For the sake of clarity and integrity, most productisation related topics are discussed in chapter six. The chapter goes over what productisation-by-design stands for, what actions are taken to verify production success and how modularity affects future improvements. The chapter also gives the reader insight on similar solutions and approaches. It is generally found relevant to have knowledge and understanding of other products in the market that may have taken a comparable approach. This chapter unveils top three competitors with their advantages and drawbacks.

Before the final chapter of the thesis, the Discussion chapter gives room for the author to reflect. Discussion chapter will have personal opinions regarding the thesis and the progress towards the completed product. In addition, it will have personal opinions and thoughts about the discussed topics.

Last chapter is dedicated to drawing conclusions along the project. In this, each chapters' topics are systematically reflected upon, whether it was ought to have done anything differently – and, what improvements could be contributed to the overall process. The final chapter will conclude the thesis.

2 Cloud technologies and services

This section covers softwares and platforms that are offered as services. These define the ingredient types that are necessary or highly valued for the project. Cloud vendors, such as Microsoft or Amazon offer most tools suitable for a multitude of different projects, including the one in this thesis. Current industry standard in cloud enabled projects has an established role for architects due to the near endless implementation options that the cloud vendor provides.

The cloud vendors offer a lot of different certificates and training, even for free. This encourages students and upcoming professionals to get familiar with the services they offer. The project in this thesis takes advantage of cloud vendor offerings, such as Microsoft Azure Student Subscription. With this subscription, the project's cloud services were free at this stage. When scaling up, the subscription type would change.

2.1 Infrastructure

Target environment infrastructure can be thought of as a specific infrastructure for certain target or a more generic philosophy of a smart building. The scope of the thesis resides more on the specific side. Building IoT (BIoT) technologies at this time are rather scattered. Most often BIoT solutions are reflected from the concept of Industry 4.0, which embraces automation of multitude of aspects, such as accessibility needs detection and automatic door opening. These sets of layers are categorised in figure 1. From a certain aspect, any infrastructure utilises one or more of the depicted layers. Other researched solutions take slightly altered infrastructures into account. For example, Long Range Digital Wireless Area Network (LoRaWAN) is often used to control slave-like smart components. In the thesis case, LoRa usage is rather a part of the infotainment display extensions (local connectivity). The infotainment display extensions are illustrated as the physical layer in figure 2. The installable staircase display is more of a node that connects the Cloud services to the building (remote connectivity), which acts as the communication layer in figure 2. Furthermore, other researched solutions focus more on local connectivity, whereas the thesis case scope is more oriented towards remote connectivity. [7] Other similar solutions also were more focused on current operation digitalisation, such as energy consumption management and automation. [8]

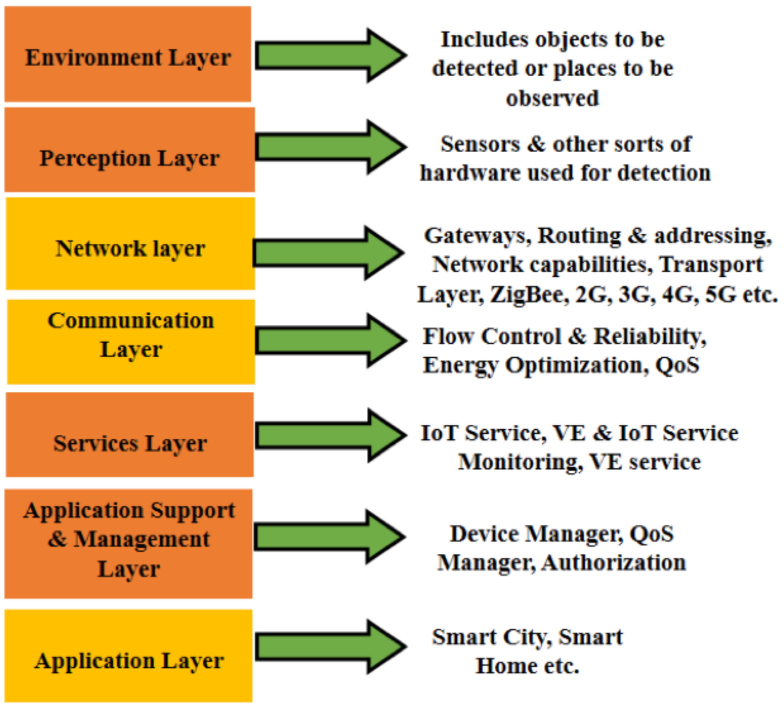

Environment Layer
Includes objects to be detected or places to be observed
Perception Layer
Sensors & other sorts of hardware used for detection
Network layer
Gateways, Routing & addressing, Network capabilities, Transport Layer, ZigBee, 2G, 3G, 4G, 5G etc.
Communication Layer
Flow Control & Reliability, Energy Optimization, QoS
Services Layer
IoT Service, VE & IoT Service Monitoring, VE service
Application Support & Management Layer
Device Manager, QoS Manager, Authorization
Application Layer
Smart City, Smart Home etc.

Figure 1. Infrastructure architectural layers and levels. [7]

The thesis takes an IoT solution approach to the problem, since it suits the purpose and the target environment. Each display has a computing module, so let's think of each as a separate IoT device. In business practicalities, this translates to each apartment staircase having one single IoT device. It is notable that most apartments have multiple staircases, thus the same manager. In figure 3 of a desired infrastructure the devices are marked as part A.

The system must support multiple managers for multiple devices. Due to this, the system needs an identity management (IDM) system and a web user interface

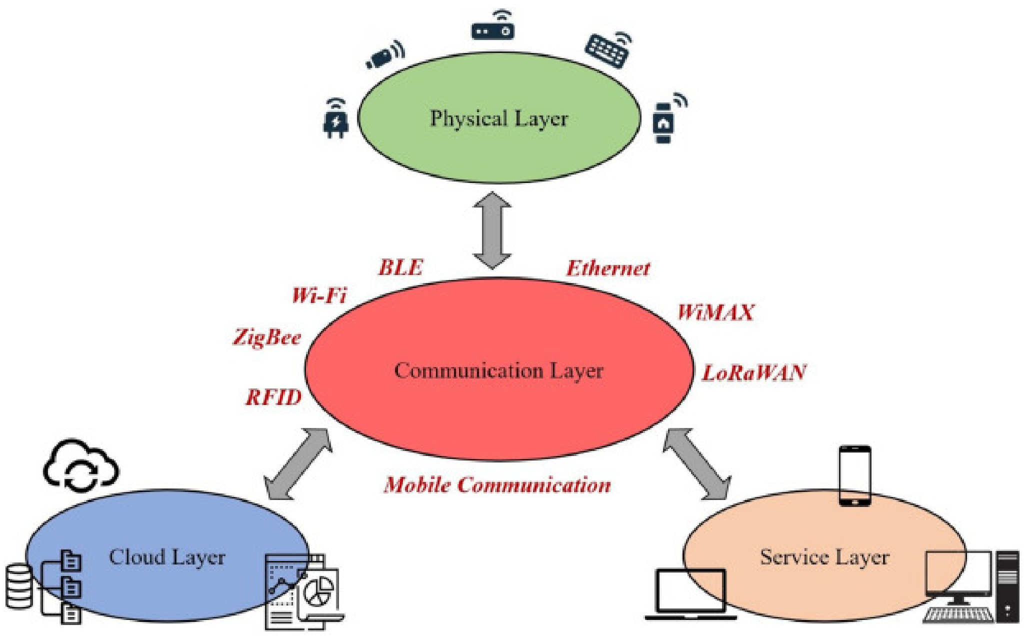


Figure 2. Alternate visualisation of infrastructure layers, where Communication Layer represents the staircase display device and physical layer provides extendability. [8]

(UI) to manage the devices as per ownership defined in the IDM. The part C in the infrastructure illustration takes these into account alongside an API to simplify and specify whole platform management.

In order to access the devices over the internet, a cloud service provider (CSP) is required. The CSP incorporates, *inter alia*, a realtime socket to the devices, resource management and data storage for multiple purposes as described in the B part of Infrastructure illustration.

2.2 On-premises IoT devices

The part A of the infrastructure – the most core component of the ensemble – represents the devices. Devices can range even on different architectures, such as ARM or x86. Most of these small devices facilitate the properties of a System on a Chip (SoC), which in this case is a basic desktop capability. For this purpose, the device must have a powerful enough central computing and graphics unit, ram for

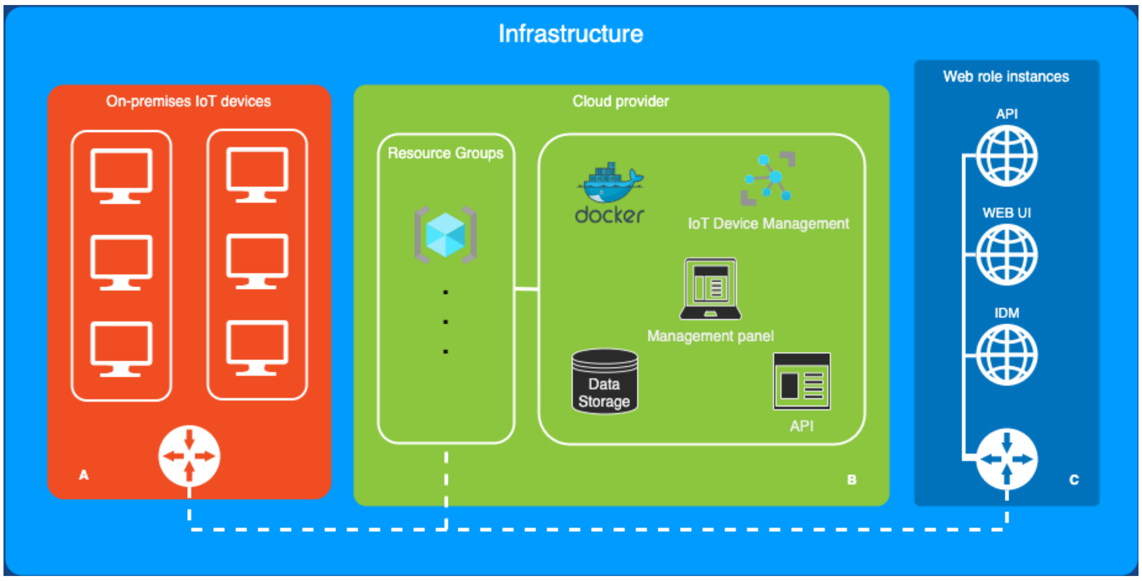


Figure 3. Illustration of Cloud infrastructure in thesis case

OS with software and connectivity for a display, internet and peripherals.

Internet connectivity can be a WiFi hardware module within the device, but for a stable connection, a cabled interface is generally recommended. In the thesis we assume one of the two mentioned connectivity types are available. It is good to note that these might not be available in all cases, and a 4G/5G modem can be used instead, as a peripheral.

Each device installed in a staircase has to have means for thing-to-human communication, which most often is a display of some sort. Other researched solutions propose, that a LED board is a feasible component and can be extended to provide a multitude of information to the user [9]. However, the thesis case productisation approach indicates a need for more tailored, user friendly and graphically aesthetic display. Thus, a large flat screen computer monitor or TV is more suited for a modern design. Further research also suggests that increasing amounts of similar IoT devices provide great opportunities for context enrichment. An example of this is deliverance of notifications to staircase infotainment screen, which is in scope of the thesis. [10]

2.3 Cloud Service Provider

Cloud service provider, or CSP for short, gives access to a Platform as a Service (PaaS) that offers a wide variety of different level infrastructure components via web. Often multiple technology stacks are supported for multiple resources in the cloud, ranging from Node.js web server to virtual network adapters. At the time of writing, the three major CSPs are Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform (GCP).

For the CSP to serve our goal, we are most interested in following components; a node for fleet management (IoT devices), web apps, container registry, database, version control, continuous integration and development (CICD), documentation and code based infrastructure templating for scaling (e.g. ARM templates). Often the CSPs also offer identity management, but there are many other good alternatives and could even be custom implemented.

The CSP should also accommodate a management API, so that we can poll information about different statuses in the end user's interface (web UI). At bare minimum, these must be in harmony with IDM and offer device query, health check and environment variable management. Optionally, it would be desirable that the CSP API would also offer device management, i.e. creation, deletion and alteration of resources that represent the physical IoT devices.

CSP will also need to accommodate device software shipment and management. A key challenge in an IoT solution is controlling and managing various applications and their alterations while the amount of devices increase rapidly. For example, Azure IoT Edge offers docker container shipment. It can be paired with a container registry. Usage of such design allows automated remote deployment of software packages and simplifies the production workload. [11]

The cloud layer provides possibility for data processing that the IoT devices provide, and, often is a central part of an IoT solution [8]. However, in this thesis case, data gatheration from the devices is minimal and more oriented towards coherence, which counts for self-healing properties, such as recovering from a local power outage. For such purposes the cloud should provide storage and means to minimize data loss. These abilities need to be flexible, since the target apartment environments will change and alter from time to time. [8] For instance, commercial environments may alter office locations and make architectural changes to floors, which need to be reflected on the infotainment display.

2.4 Web Role Instances

Third group C illustrates instances that are globally available. These provide means to interact with the whole platform we are creating here. In my bachelors thesis I created a platform with my startup for restaurant management. The tool was first built as a native standalone desktop application. Later on, it was found that the restaurants had difficulties running the software because it was not compatible with devices that are in line with target restaurant companies. As per this reason, it was necessary to re-program the whole UI software to be a web application. This way, also tablets and other similar non-desktop devices were compatible.

In order to avoid the same mistake over again, a web application has gotten the seal of approval here in terms of the UI. There are countless frameworks and different web app implementation methods nowadays, so choosing a suitable one is definitely up for question. The three major frameworks are Angular, Vue and React, which are based on javascript.

In order for the ui to interact with other components of the platform a dedicated API is more or less a necessity to increase coherence and reduce coupling. The dedicated API will provide both, a secure layer of abstraction to third party services like IDM and CSP, and, means for internal management, such as resident list editing. The API will be used by devices and applications. Modern APIs obey industry standard representational state transfer (RESTful) constraints, which were defined by Roy Fielding [12]. For the API to provide a long lifetime, RESTful architecture is ought to be followed.

The environment subject to this research in the thesis has a property that is common to most modern platforms – identity management. Identity management should provide means to create and manage users and groups, and to define their scope of access. For example, an apartment manager should only see the devices that are in the staircases of their apartment, but not other devices. The permissions should reflect in the API to prevent malicious access outside the UI, assuming UI security integrity. Security is described and debated more in chapter 4.3.

3 Case and specs

Digitalising a notification board is not a new technology, but the environment of this problem proposes a unique playground waiting to be fiddled with. IoT is shaping the future of residential and commercial smart buildings [8]. Apartment environment promotes multiple nodes/action points/properties to be digitalised as services, including, but not limited to, residents, local news, local public transportation, weather and stock market data. Hence apartments were chosen as the first target group, contrary to shopping malls, hotels or public places. Apartments are also generally more behind in information technology than the other targets mentioned.

These services can be incorporated into a computing unit that has a display and an application to control it. The unit in this thesis is called an infotainment display or alternatively digital signage. It can be complemented with intuitive hardware for user interaction. This approach steers the solution have three types of communication; thing-to-thing, thing-to-human and human-to-thing [13]. All the services and extensions can be put into four categories by their nature of feature – juridical, nice-to-have, accessibility and possibilities as future improvements. These will be discussed more in chapter 6.

3.1 Project ApartaView

This thesis goes through the major steps of developing a platform that implements above description. Under development the project got the name ApartaView. The project is a startup by its nature. This platform serves as the case of this thesis. Thesis will look in to some development possibilities and analyse alternative choices e.g. in selected technologies. Since the target environment may require service and extension personalisation of many sorts, it is important to promote scalability in technologies selection.

In order to provide more value to such infotainment displays, they are integrated into an IoT (Internet of Things) cloud service. In this thesis case, it enables tailored remote management. Core idea here is to be able to visit the target install site as seldom as possible – and in turn, have as much remote control as possible, both, for administrators, and other users. The whole ApartaView ensemble was chosen to consist of infotainment display IoT devices, a cloud to connect them to, a tailored backend, a management UI (User Interface) and an IDM (Identity Managment Platform) to scope users and services in between the systems. These parts were seen as the base necessity for such idea to be production capable as a platform, and possible by the current technologies.

3.2 Platform genericity and use case exploitability

The platform that is being built here can suit more than a single purpose. With minor or no modifications, this platform described in the figure 3 is by no means limited to apartment staircases. To be more specific, we are more or less tailoring the components' contents to bring better value in the target purpose. A different pivot means that the product's or service's core is different. For example, a customer segment pivot here means, that the product core is changed in a substantial enough way, that it serves a different environment or a target group. In this case, a customer segment pivot could be to alter the software in such way, that it is fruitful to construction sites instead of apartment staircases.

During market research of available similar products, most common solutions, that can be considered as pivots to ours, were business displays, culture sector info screens, academics and teaching assistive displays [14], dedicated advertisement displays and social and healthcare sector info screens.

4 Architecture design

After the required infrastructure and the case specs for our target environment have been defined in general terms, it is time to specify and define how the implementation will take place. In this section technologies that serve the target goal as well as possible are selected. This is done in such fashion, that it yields a system, which is easy to alter, extendable, secure, cost effective and most importantly provide convenient components.

4.1 Selected technologies

Starting with part A, which is the core of the platform, the IoT devices, there are two main options, ARM and x86. Since we are building a user interface to be ran on the devices, other lower level architectures would be bottlenecks. The devices purpose is to run a graphical user interface, which scopes out lower level cpu architectures, such as Reduced Instruction Set Computer (RISC). Cheap price, small form factor and good availability gives an upper hand for ARM, thus it is selected it at this point. In comparison though, most software doesn't incorporate support for ARM as well as they do for x86. In fact, software support was shown to be a bottle neck for other technologies that were selected in the process. This is discussed more in detail in the execution section of the thesis. Let it be known at this point that ARM was later discarded during development and x86 was used instead. Thankfully this was possible due to the proper infrastructure design, which provided flexibility.

Next step is to choose a CSP that facilitates IoT connectivity with fleet management. Three majors are AWS [15], Azure [16] and GCP [17]. They offer more or less the same capabilities, even near identical implementation flows with similar cloud component names. Due to the similarities, Azure was chosen as CSP, since it is the most familiar of the options. As the IoT component, Azure offers IoT Edge, AWS offers GreenGrass and GCP offers Google Cloud IoT.

4.2 Remote access and troubleshooting

Azure IoT Edge has a multitude of tool implementations that help remote management of the IoT devices. The relevant additional IoT tools for this project that Azure offers, are remote container deployment for updates, Digital Twin for real time metadata, Remote Method Invocation (RMI) to start certain procedures on the containerised IoT software, Message Queue (MQ) compatible socket (MQ Telemetry Transport or MQTT) and a cloud service bus. MQ is not needed at this stage of the project, since unlike most iot projects, no real time iot data is necessary to be transmitted from the device to cloud. Chapter 6.3 notes future improvements, such as activity collection and advertisement reach.

However, the iot tools that Azure has are limited to the runtime software that is run on the target device. This means, the operating system is not controllable without custom implementation, since the runtime sandbox has somewhat limited access outside of itself. This same restriction applies to the containers that it runs, which means that the custom IoT software has the same bounds.

The nature of this IoT project has three factors that define remote access requirements; install sites can be really remote, troubleshooting UI requires access to what is displayed on the screen and desktop/window manager remote configurability. Due to the IoT runtime limitations, it is appropriate to find a parallel solution. In this project, the remote management software was chosen to be remote.it [18]. Remote.it has an installable runtime and a portal to control all registered targets. For each target device the service offers availability reports and reverse TCP with custom configuration. With this, it is configurable to embody SSH, VNC and other proxies, which are necessary for this project's remote management requirements.

4.3 Security

All communication in between the different parts of the ensemble is encrypted with applicable technologies. For application layer security the project API and UI utilise HTTP along TLS/SSL (HTTPS). At transport layer, the reverse proxies with Remote.it use TCP/UDP along TSL/SSL and generated per use. Most are already implemented in cloud components, but Web App Service certificates are configured by hand using Certbot tool [19]. Web applications, the API and UI, were ran on home server at the time of writing. Normally they would follow the architecture and be ran in the CSP. Furthermore, Azure Student subscription did not support custom domain names, which defeated the productisation purpose. Due to this, API and UI were subject to server configuration that is covered in the following chapter.

At this point no physical security is designed for the systems. It was estimated to be rare to suffer a condition, where device stays online but the display is physically hijacked. If the device was to go offline, a configured alert would notify administrators via cloud monitoring service. More-on, physical security beyond the current state is not considered a priority, but is subject to re-evaluation at later point of the product lifetime. After the first targets have been successfully deployed, a secure frame can help with malicious physical attacks, such as disconnecting or damaging cables.

5 Execution

A lot of documentation is included in this section of the thesis, which dives deep into technology at certain points. The project ApartaView was executed up until beta stage at the time of writing. This means, that the core of the project is complete and is implementable to a target location with minimal bottlenecks for required features, e.g. occasional resets may happen. However, the product core is not going to have major changes after this point. Beta staging brings value to the product and reduces total project costs. [20]

Execution consists of six major elements that should play in harmony; management frontend, backend, miscellaneous configuration, cloud services deployment, iot devices development and IDM integration. These are somewhat categorised in the subsections below, but have a good amount of cross dependencies during their execution lifetime. In context, this means that one can not be developed to operation capability without the other elements.

5.1 Web UI

The end user interface will be regarded from this point on as the web UI, and acts as the service layer depicted in figure 2. However, unlike in the figure 2, the service layer communicates with the communication layer via the cloud layer, since the IoT module resides there. The chosen technology here is Angular framework, particularly the newest version at the time of writing (11). Angular framework is built on typescript, which is built more or less on javascript, a common web programming language. [21] In this project, it is required that the web UI implements a login system for identity managed access. The project also requires that the web UI has to have an interface to list devices that belong to the scope of the logged user, as seen in screenshot of the UI in figure 4. At the first release stage, the user should also be able to alter device metadata. For later phases, the roadmap allows the end user to alter what is displayed on the infotainment screen in terms of components and layout. For example, instead of local bus schedule the user could choose stocks and commodities to be displayed.

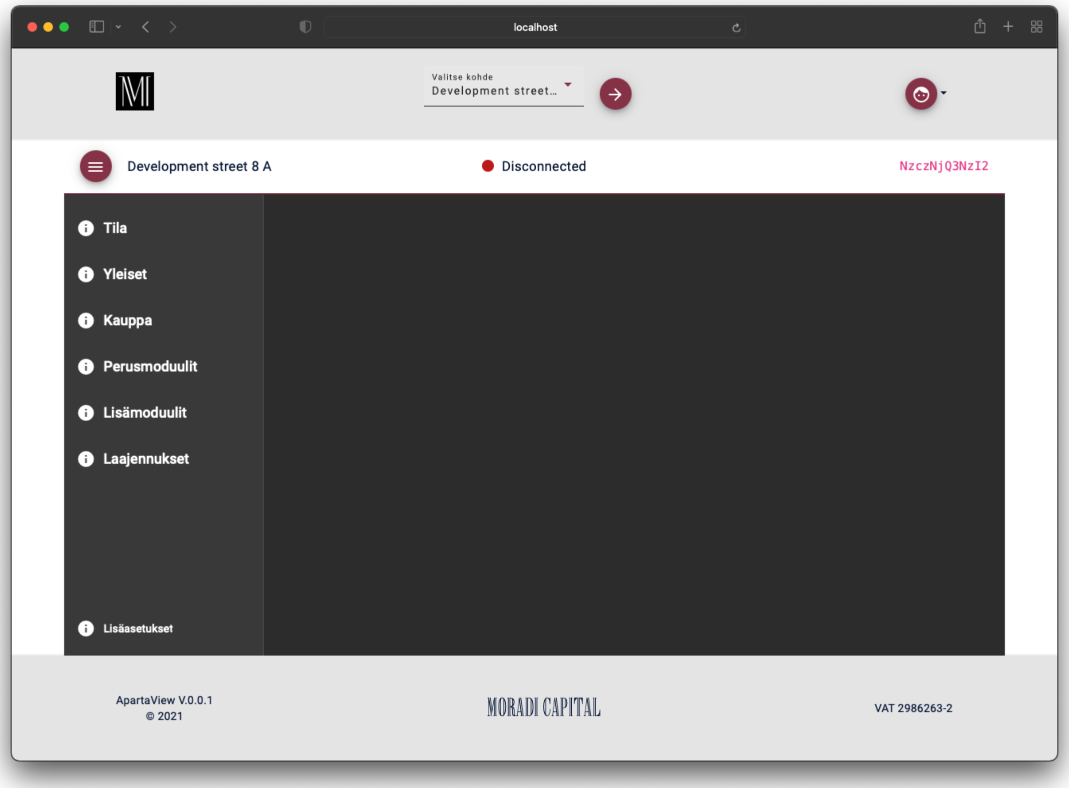


Figure 4. Screenshot of ApartaView admin panel without content during development. User can select target from navigation bar drop down menu.

The UI application with Angular framework gives certain options for configuration, like API addresses. Most of these are stored in an environment file. There are two environment files, from which a correct one is automatically chosen depending on target environment, i.e. production versus development. In this useful fashion, UI and API can both be running on the localhost at development stage, and can be developed simultaneously.

Angular framework structure, logic and formation is out of scope for this thesis, but a short coverage may help the reader and is as follows. Application starts from a main.ts, which bootstraps the App Module. App Module contains references to dependencies, routes, authentication interceptor, etc. App Module in turn bootstraps App Component, which is the core component. Angular application is mainly formed of components and services. Each component represents a displayed section. Component has a controller file in typescript to offer programmatic access, a HTML file to define content and a style file to make content appealing and arranged. What Angular here does, is simplifies harmony of these three files. Each component can then on be injected to another components view. Then again, in order for these components to have common internal and external logic, a services are introduced. For example, Authentication will be offered as a service. This way, all components can access the same instance of the user and retrieve necessary information.

5.2 API

A core feature of the project is remote management. The IoT software is ought to be configured remotely within the cloud portal to a certain degree by end users. End user in this context is the apartment management personnel. The portal, Azure in this case, however, is meant for developers and contributors alike. It is not suitable for end users, since the cloud portal interface is not tailored to a specific implementation. It offers unwanted access to resources and configurations that make it both a security risk and difficult to use for end users.

Due to this design, the CSP offers Management API's. With these API's, the software designers and developers have a possibility to create a tailored interface for their end users, which is the chosen method of delivery in case ApartaView. By using the Management API's, most key functionalities of the portal are programmatically available, such as updating digital IoT Device Twin metadata that reflects on the device itself. The end user will have to programmatically communicate with multiple backends, which poses inconsistency problems. Thus furthermore, a custom ApartaView API is developed. It serves as a centralised aggregate to other API's in addition to internal microservices, such as resident list management.

API for this project is the aggregate that provides a programmable interface for all ApartaView accessible services. This means, that it facilitates internal management and provides abstraction to other API's, kind of like a one-stop-shop. Azure does offer API Management (not to be confused with Management API) service as a resource, which provides access management, routing and quota setting for all programmable web interfaces related in a project. This service is more aimed for integration projects and projects, that have many legacy systems, like digital banking. Here, we are building a tailored API anyways to be a backend for the UI, so we might as well use it for integrating other API's to the project, thus rendering API Management an overkill. Other API's in this context are Azure Management API, where IoT devices can be manages and Auth0 API, which is the chosen IDM in ApartaView project.

ApartaView API is built with a Python framework called Flask [22]. API follows a service architecture, where connecting to third party API's and similar are abstracted at code level. The services are located in 'service' folder, which consists of Authentication, Azure and Cosmos services. They are utilised in the main program, app.py, by initiating the classes as objects.

```python
from azure.cosmos import CosmosClient
class CosmosService:
    def __init__(self, C) -> None:
        client = CosmosClient(C.COSMOS_URI, C.COSMOS_KEY)
        self.db = client.get_database_client(C.COSMOS_DB)
```

Listing 1: API `./services/cosmos_service.py`

API endpoint definitions in Flask is quite straight forward. After initiating the app with

```python
APP = Flask(__name__)
```

a route is defined with a Flask decoration

```python
from azure.cosmos import CosmosClient
class CosmosService:
    def __init__(self, C) -> None:
        client = CosmosClient(C.COSMOS_URI, C.COSMOS_KEY)
        self.db = client.get_database_client(C.COSMOS_DB)
```

Listing 2: API `./app.py`

```python
@APP.route("/api/external")
```

This route is a private test endpoint, where especially during development an API user can verify if their authentication is valid. To make it private, a function decorator is simply added after route definition with

```python
@requires_auth
```

This authentication decoration is a function defined earlier, that implements a Python wrapper from functools Python package. This can be considered advanced Python manipulation, but a standard procedure.[23]

```python
from functools import wraps
```

With this setup, the endpoint can not be accessed if the token in the headers does not pass inspection. Inspection is done using RS256 algorithm against Auth0 servers. This 'requires_auth' function presented in listing 3 determines if the Access Token is valid.

If token validation is not successful, app raises a custom AuthError that terminates the request with a HTTP response of 401: Unauthorized. With these elements, we can build a private endpoint that returns the privileged user's token's claims using a general Java Web Token (JWT) library as in listing 4.

ApartaView API should deliver a few key functionalities, which are device management, resident list management and scope control. At early stage, devices need a GET and a SET endpoint. This is needed in order to list devices in UI and to update their metadata respectively. Since devices are managed in Azure, more specifically

```python
def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        token = get_token_auth_header()
... # validate token
            return f(*args, **kwargs)
        raise AuthError({"code": "invalid_header",
                        "description": "Unable to find appropriate
                         ↪  key"},
                        401)
    return decorated
```

Listing 3: API authentication wrapper

```python
from jose import jwt

@APP.route("/api/external")
@requires_auth
def private():
    response = "You need to be authenticated to see this."
    token = get_token_auth_header()
    header_unverified = jwt.get_unverified_claims(token)
    return jsonify(token=token, message=header_unverified)
```

Listing 4: API JWT handling

Azure IoT Edge, ApartaView API needs to programmatically communicate with Azure Management API.

The resident lists can not necessarily be regarded as metadata, since they are more structured and richer in content in contrast to Device Twin metadata file. The apartment lists also logically lie under ApartaView domain, whereas the devices lie under Azure domain. For abovementioned reasons, a storage table system is required for ApartaView logical domain. In order for the API to manage this task, it will use Azure Cosmos DB as its database service where resident lists and similar info lies. This connectivity is abstracted as a service in the API.

Scope control in the API means, that the API limits access and sets borders for different users and groups. Each user is defined independently in Auth0. These definitions can be custom fields, such as the company that the user belongs to. Having the authentication system already in place, this information can be extracted from a verified JWT token and used accordingly. This functionality is required for example to return correct devices on request, since the devices are appointed to a certain company.

5.3 Server configuration

The project is aimed for production use, thus it must be ran on live servers with a proper domain. At this stage, the project will temporarily utilise the domain moradi.fi with subdomains view.moradi.fi and api.view.moradi.fi. Azure student subscription did not support custom domains at the stage of writing, rendering Azure not compatible with current project limitations on web services. It was decided that during this phase a custom linux server is going to be used to host the UI and API. This is configured by pointing an 'A' record to the server's IP address in DNS settings. This is done in the domain name registrar's service, which in this case is Netim.

After configuring DNS settings, the server must be configured to receive traffic accordingly. In context this means that the server must run a HTTP server for the UI and a reverse proxy for the API app server, which runs on localhost of the server on a certain port. Most common tools for similar tasks are Apache Http Server and Nginx. Both provide similar functionalities. Nginx is currently the most used tool with almost a 35% market share, compared to Apache having its peak market share of 71% in 2005 according to Netcraft. [24] For the sake of familiarity, Nginx was chosen for this project. Nginx has two main folders for configuration, `/etc/nginx/sites-available/` and `/etc/nginx/sites-enabled/`. Each host name (e.g. view.moradi.fi) has its own configuration file in folder `sites-available/`, from where they are activated using a symbolic link to `/etc/nginx/sites-enabled/`.

These configuration files define the type, source, port, destination and other miscellaneous settings for a host name.

```
server {
        server_name view.moradi.fi;
        root /var/www/view.moradi.fi/public;

        # index.html fallback
        location / {
                try_files $uri $uri/ /index.html;
        }
}
```

Listing 5: API Nginx Server block

Configuration for a web site is rather straight forward, but for the API it is more complex. Reason for complexity is the API's method of communication. API is ran as a linux service, which makes the role of Nginx to form a reverse proxy bridge in between the host name and the local web service. The API has other configuration specifics in addition to above. An API utilises multiple HTTP methods, such as GET and POST, which need to be defined separately. Also, most communication to the API is done from another host (view.moradi.fi), which means that cross origin policy must be configured. This is called CORS (Cross Origin Resource Sharing) and it is a common pitfall in server configuration. Enabling CORS is simply adding a Access-Control-Allow-Origin header to reverse proxy configuration in such fashion, that it applies to all requests in according manner. Since ApartaView API has to be accessible from almost any source, the configuration value for this is a wild card '*'.

The ApartaView API handles sensitive information, such as authentication tokens and other secrets that are sent over the internet. To run this type of service safely, it is ought to implement a security layer. As mentioned in the Security chapter, this project utilises de facto SSL (Secure Sockets Layer) for all HTTP connections. Security like this is achieved by having a root certificate that has a trusted author and an expiry date among other details. The root certificate can be extended and inherited for other domains in the same group. Generally these trusted certificates come with a cost, but a free way to generate such certificate is via a certbot tool. Certbot has automatic configuration for Nginx and can apply a root certificate to all domain names that point to the server that the certbot is hosted on. Relevant Nginx configuration in listing 7. Certbot is offered by Letsencrypt via Electronic Frontier Foundation.[19]

```
tijam@nexus:~$ sudo certbot --nginx
Saving debug log to /var/log/letsencrypt/letsencrypt.log
Plugins selected: Authenticator nginx, Installer nginx

Which names would you like to activate HTTPS for?
- - - - - - - - - - - - - - - - - - - -
1: moradi.fi
2: view.moradi.fi
3: api.view.moradi.fi
```

Listing 6: Certbot command line tool

```
upstream app_apiviewmoradi {
    server 127.0.0.1:3002;
    keepalive 8;
}
server {
    server_name api.view.moradi.fi;
    access_log /var/log/nginx/api.view.moradi.fi.log;
    location / {
          if ($request_method = 'OPTIONS') { ... return 204; }
          if ($request_method = 'POST') { ... }
          if ($request_method = 'GET') { ... }
          add_header 'Access-Control-Allow-Origin' '*';
          proxy_set_header X-Real-IP $remote_addr;
          proxy_set_header X-Forwarded-For
           ↪  $proxy_add_x_forwarded_for;
          proxy_set_header Host $http_host;
          proxy_set_header X-NginX-Proxy true;
          include proxy_params;
          proxy_pass
           ↪  http://unix:/var/www/api.view.moradi.fi/application.sock;
          proxy_redirect off;
     }
    listen 443 ssl; # managed by Certbot
    ssl_certificate ... # managed by Certbot
    ssl_certificate_key ... # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by
     ↪  Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by
     ↪  Certbot
}

server {
    if ($host = api.view.moradi.fi) {
        return 301 https://$host$request_uri;
    } # managed by Certbot
    server_name api.view.moradi.fi;
    listen 80;
    return 404; # managed by Certbot
}
```

Listing 7: API Nginx configuration /etc/nginx/sites-available/api.view.moradi.fi.conf

5.4 Cloud

ApartaView's CSP was chosen to be Azure in the architecture design phase. The subscription facilitates a logical directory, in which lies resource groups. A resource group is the parent for any set of resources, from which the resources can be managed in common settings. There are two resource groups, one for development and one for production. It is advised to keep these groups separated due to possible breaking changes in between resource dependencies. A resource group in ApartaView at this stage was chosen to consist of a Container Registry, IoT Hub, Cosmos DB, Storage Account, two Web Apps and an App Service Plan, as the figure 5 shows.

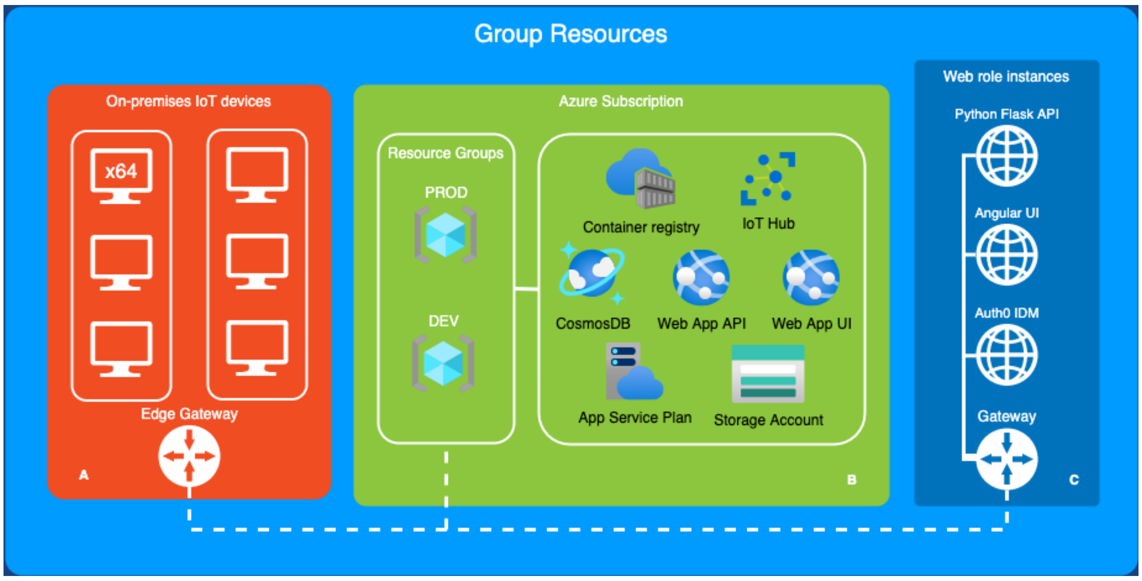
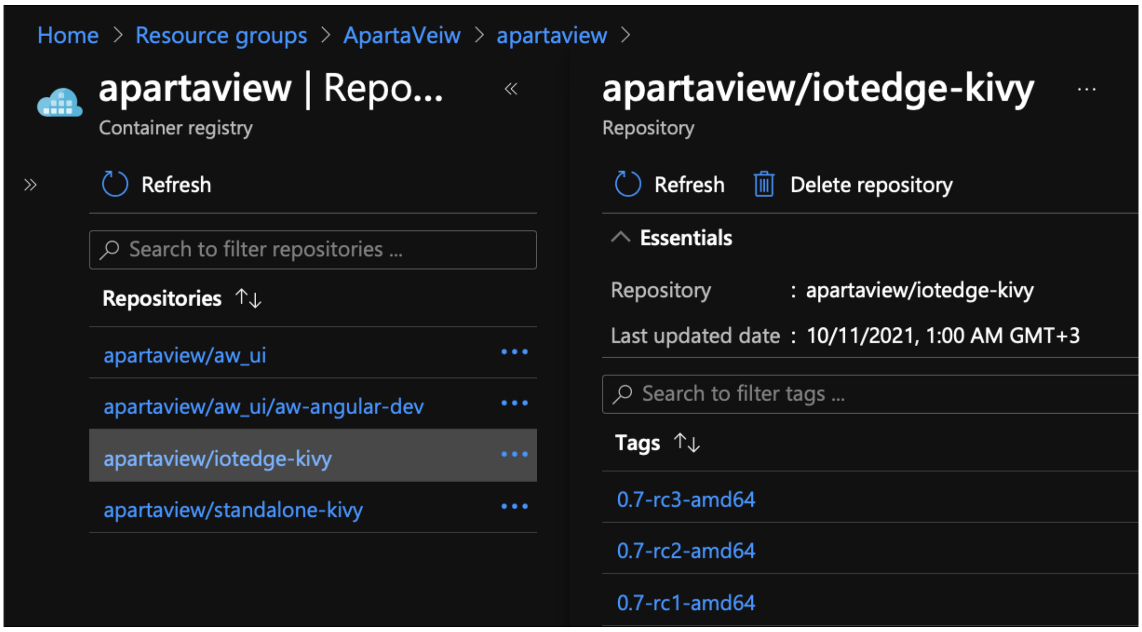


Figure 5. Selected ApartaView Architecture resources based on infrastructure.

Container registry will hold all the Docker compatible containers that are involved in the project. Containers consist of software that has all the dependencies bundled in. They depend on a container engine, which orchestrates the containers, which are made available in the Container Registry shown in figure 6. The container engine is more on configured to pull a specific image from the registry. An alternative method exists where containers are held locally at development machines and deployed from there, but in a production setting this is considered evidently bad practice. Docker containers in ApartaView project are different versions of the IoT software, which run within IoT Edge runtime on each device. The method for IoT software containerisation is discussed more in detail in the next subsection. Later on, ApartaView API will be also a containerised application.



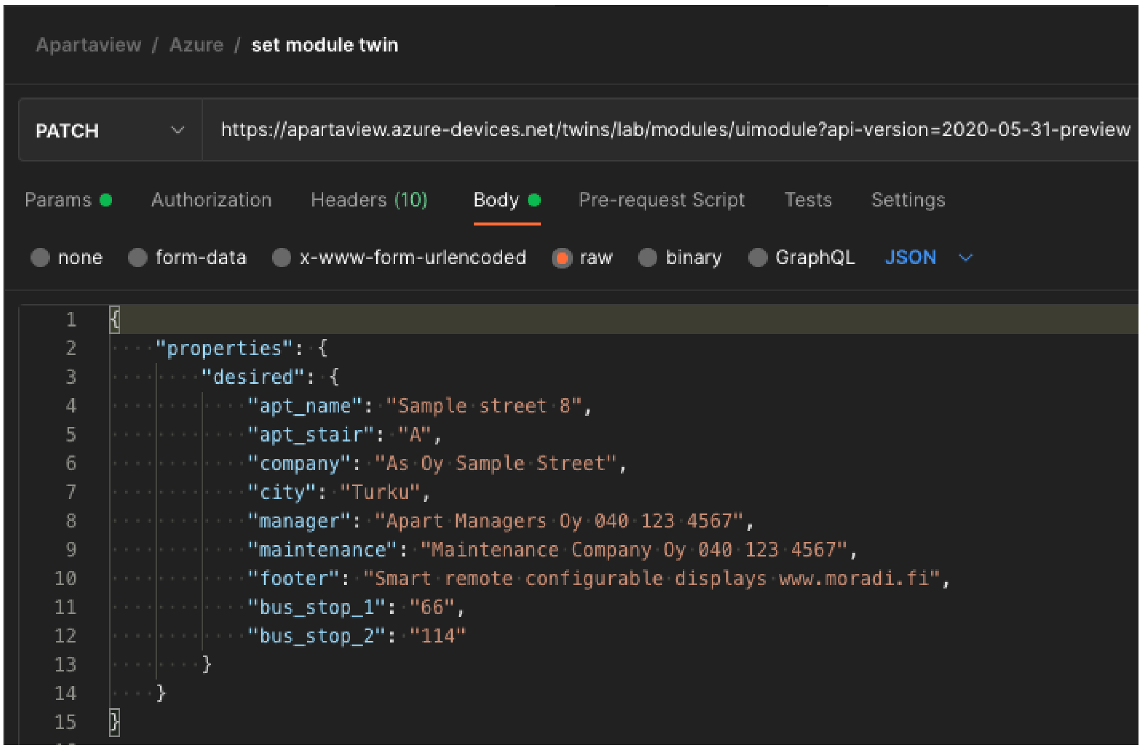
apartaview/aw_ui
apartaview/aw_ui/aw-angular-dev
apartaview/iotedge-kivy
apartaview/standalone-kivy

Repository : apartaview/iotedge-kivy
Last updated date : 10/11/2021, 1:00 AM GMT+3

0.7-rc3-amd64
0.7-rc2-amd64
0.7-rc1-amd64

Figure 6. Azure Container Registry

IoT Hub resource provides IoT Edge functionality, on which ApartaView project is focused. IoT Edge consists of two counterparts, a runtime on the device and the cloud service. These communicate over MQTT and provide extended functionalities. These functionalities include a Digital Twin, which facilitates metadata of the device and each module in a digital instance that reflects on the target. A module in this context is an enriched Docker Container that integrates with Azure IoT Edge runtime. IoT Edge functionality also gives necessary tools for remote module deployment by manifest definition. To complement communication of the counterparts, the IoT Edge module can subscribe MQTT events for Digital Twin changes and custom method invocations. Azure Management API has coverage on IoT Edge functionality. For instance, following the process in figure 7, module Digital Twin can be set by executing a HTTP PATCH on Azure Management API.



https://apartaview.azure-devices.net/twins/lab/modules/uimodule?api-version=2020-05-31-preview

```
{
    "properties": {
        "desired": {
            "apt_name": "Sample street 8",
            "apt_stair": "A",
            "company": "As Oy Sample Street",
            "city": "Turku",
            "manager": "Apart Managers Oy 040 123 4567",
            "maintenance": "Maintenance Company Oy 040 123 4567",
            "footer": "Smart remote configurable displays www.moradi.fi",
            "bus_stop_1": "66",
            "bus_stop_2": "114"
        }
    }
}
```

Figure 7. Postman PATCH query to update device's module's Digital Twin.

Only most vital data that is minor in size is injected in Digital Twin properties. Not all data is suited here, such as resident lists that were brought up earlier. For other data purposes the Cosmos DB was chosen. Azure offers a Python library, which simplifies the use of the database in a *pythonic* fashion.

```python
def get_apartment_residents(self, apt_id, apt_stair):
    container = self.db.get_container_client('residents')
    residents = container.query_items(
        query='SELECT * FROM residents r WHERE (r.apt_id = @apt_id
        ↪  AND r.apt_stair = @apt_stair)',
        parameters=[
            dict(name='@apt_name', value=apt_id),
            dict(name='@apt_stair', value=apt_stair)
        ],
        enable_cross_partition_query=True
    )
    return residents
```

Listing 8: CosmosDB query to retrieve apartment residents.

Due to Student subscription limitations explained earlier, ApartaView web applications are not hosted in the cloud, but on a custom server. However, if this was not the case, App Service Plan resource is present to define the resources available to the apps. This involves pricing, and is certainly in the scope of a cloud architect when designing a cloud ensemble. Depending on the virtual hardware allocations, the price in Azure can range approximately from 8€ to almost 200€ a month. On the Student subscription model, 10 simultaneous applications are supported with 1 gigabyte of disk space. Be that as it may, custom domains nor auto scaling is supported in the free student tier. [25] In addition to the App Service Plan, web applications require storage to serve the content from, which in turn has its own pricing plans.

5.5 Thing

Internet of Things naturally consists of Things that are connected to the internet. This is not necessarily a new field in communication technology, but definitely trending and evolves rapidly. The devices that count as Things can vary a lot in sort, but common to them is processing ability and remote communication capability in different formats. Most common occurrence might be a payment terminal. It has independent logic to a certain degree, but reports and verifies certain information against a cloud service. The cloud service orchestrates all payment terminals linked to it.

In ApartaView project, a Thing is a device that operates the display and is orchestrated by Azure, which is managed by ApartaView API. At this stage, ApartaView IoT devices don't report data back to server, such as sensor data. Later on whilst the product evolves, some sensors are designed to be added, including screen time monitoring with a camera module capable of human detection. A camera can be used to adjust the contents of the display. [26]

Device hardware was at first a Raspberry Pi 4 model B. Figure 8 demonstrates it's installation to the rack that also mounts the display. The device has 4GB ram and a quad core ARMv8 64-bit SoC with ethernet, WiFi and a HDMI connector. [27] This suits the purpose, since it has capability to run a desktop window manager with some software that is accessible over the internet. The Raspberry was installed with DietPi OS. The chosen OS provides a fair amount of customisation options out of the box, is relatively reliable and is based on such linux that can facilitate the goals of the project.

The IoT User Interface was chosen to be built with Kivy framework over Python. An alternative would have been Qt, but it turned out to be significantly more complicated to get to work within a container. Kivy requires multiple graphic libraries, from which SDLv2 is the graphics layer that Kivy utilises to draw graphics on screen. It was at this point found out, that the SDL libraries did in fact work from inside of a Docker container, but not from within Azure IoT Edge Runtime. Azure IoT Edge Runtime was even configured to use Docker container engine instead of the default Microsoft variant called Moby, but without success. The problem appeared to be dependent of the processor architecture, which for Raspberry Pi 4 is ARM.

After validating the problem with virtual machines, Raspberry Pi was retired. AMD64 took over, as the problem did not occur with x86 systems. Hardware that replaced the Raspberry Pi 4 in the project was Rock Pi X SoC from Raxda, as documented in figure 9. It has a Raspberry Pi 4 form factor, but is a completely different product. It features an Intel Cherry Tail quad core x86 processor and a 4GB ram option alongside necessary connectors. [28] Later on when animations were added to the IoT UI software, Rock Pi X graphics hardware fell short and became a bottleneck. However, the performance remained sufficient at around 10 frames per second during heavy animations after adjusting graphics driver configuration in the container. When proceeding with production, the hardware will have to be replaced

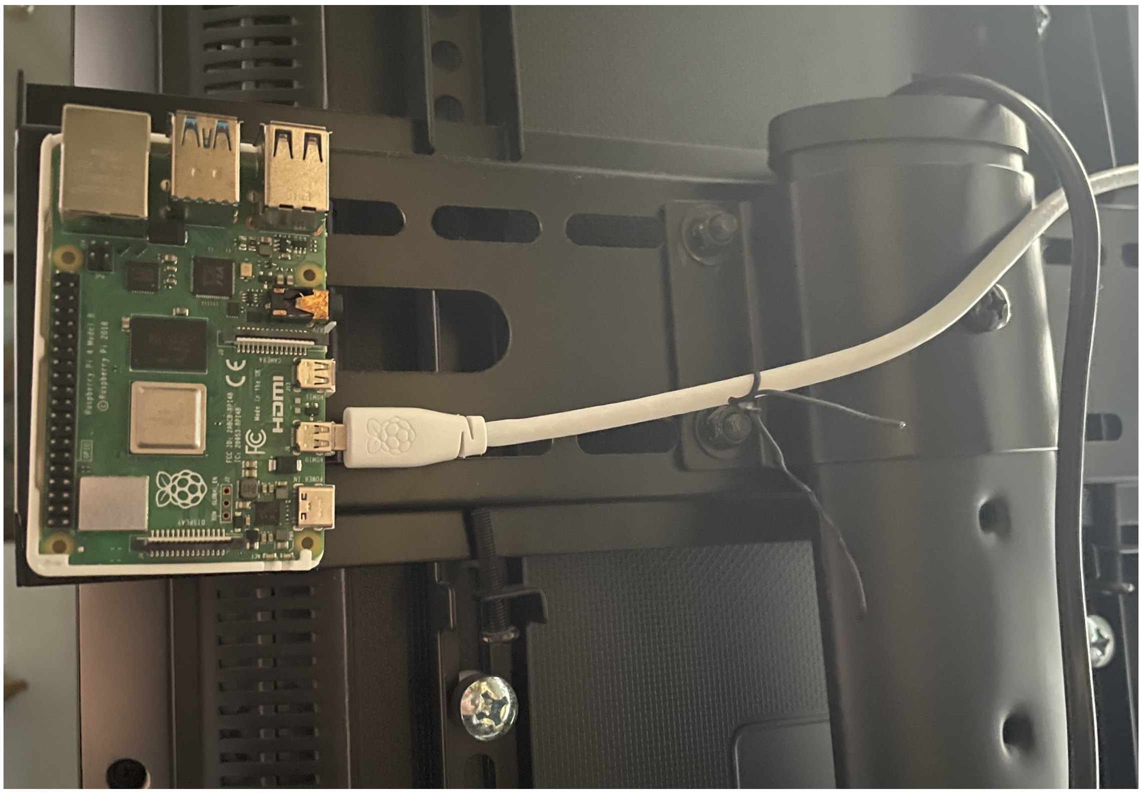


Figure 8. Raspberry Pi mounted behind a vertical display rack.

again. This time with a more graphics orientated SoC or a small form factor desktop computer.

The hardware must host a linux system that has a running window manager, in this case an X server. The IoT UI software will use the X socket to draw on to the screen from within the container. This tunneling is defined in dockerfile of the app. The system also needs the Azure IoT Edge Runtime alongside Docker to host containers. Edge runtime will take care of orchestrating, managing and launching the containers that are subject to the device.

Updating a new version to devices is straight forward, the software is first compiled and containerised on local development computer. The container is then shipped to Azure Container Registry (CR), from which containers can be pulled by authenticated requestors. After the container image has been pushed to CR a new deployment manifest is generated. The deployment manifest states which con-

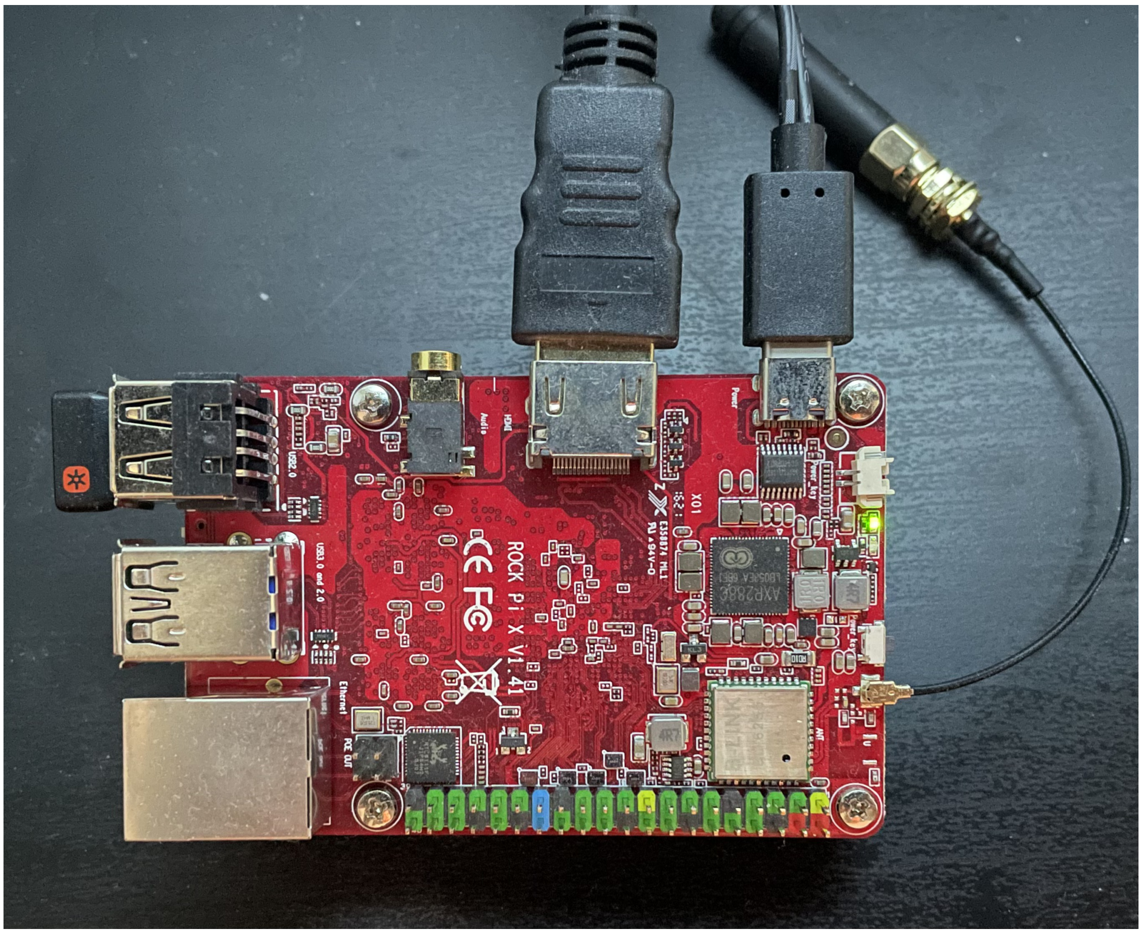


Figure 9. Rock Pi X with a WiFi antenna.

tainer images the runtime is ought to pull and where they are. The manifest is shipped to the device via Azure IoT Edge internals. Shipping the modules to the devices has a predefined workflow described in figure 10. Device then shuts down old container, pulls new and runs it automatically. This process happens near real time, excluding the time it takes to pull the new image. By empiric research, it was found to take up to 1-2 minutes to have the new container up and running starting count from manifest deployment.

5.6 IoT software

IoT Edge application root folder has a deployment.template.json for generating deployment manifests. `config/` and a `modules/` folder contain deployment manifests

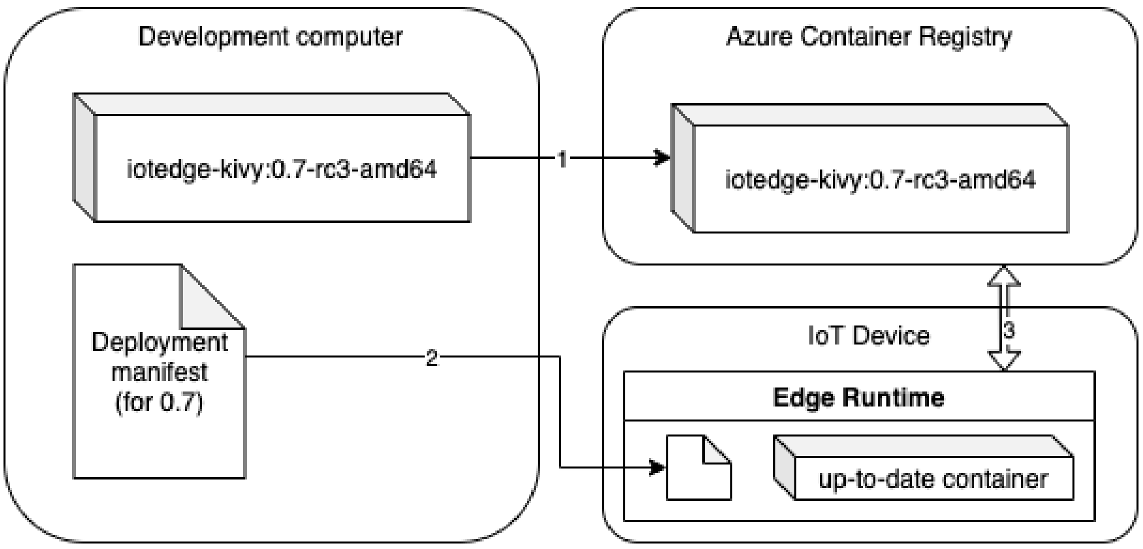
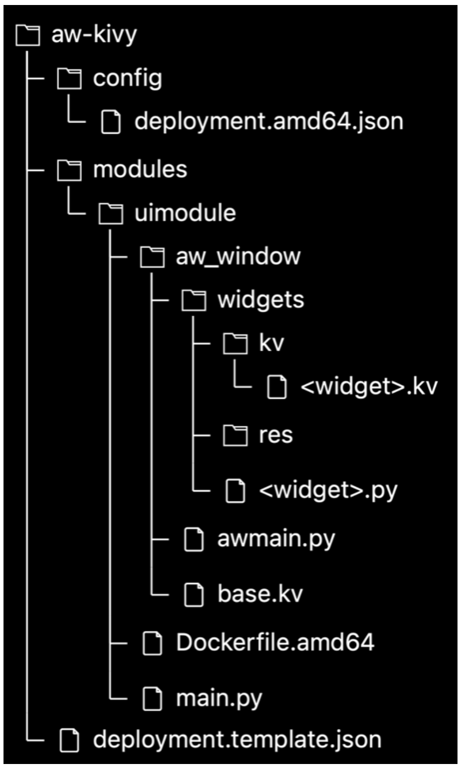


Figure 10. Container movement flow in between sources and targets.

and modules respectively in the project. In ApartaView case at this stage there is only one module named uimodule. In future, extensions like speech recognition or face AI can be just added as a module. Modules have a built-in messaging channel orchestrated by the Edge runtime. A module's root includes an application start point `main.py`, its library requirements in `requirements.txt`, a `module.json` to complement the module for deployment templating and a dockerfile to define containerising process. Figure 11 states the major folder structure that reflects the case project.

Dockerfile of this module underwent the most changes during development and was the main obstacle to orchestrate graphical objects from within Edge Runtime container. There was immense researching and testing involved to get this combination work smoothly. The dockerfile is derived from `amd64/python3.9.6-slim-buster` image. For Kivy framework to be able to draw graphics on display, it requires SDL (Simple DirectMedia Layer) library to be configured correctly. SDL, which provides an interface to graphics hardware, has to be built from source. This requires some build tools listed in dockerfile snippet listing 9. SDL utilises OpenGL (Open Graphics Library) and Direct3D to deliver a high performance graphics con-



```
aw-kivy
├ config
│ └ deployment.amd64.json
├ modules
│ └ uimodule
│   ├ aw_window
│   │ ├ widgets
│   │ │ ├ kv
│   │ │ │ └ <widget>.kv
│   │ │ ├ res
│   │ │ └ <widget>.py
│   │ ├ awmain.py
│   │ └ base.kv
│   ├ Dockerfile.amd64
│   └ main.py
└ deployment.template.json
```

Figure 11. IoT software folder structure.

trol layer. There is a Python API available. [29]

OpenGL is the default library injection in SDL. However, containerisation procedure persisted issues when running the application. This might be related to the software having access only to the Linux X window server socket, not also the hardware. Software ran, but framerate appeared less than acceptable (less than 5 fps). Thankfully, this can be replaced with an open source equivalent Mesa. Mesa has an upper hand in device virtualisation via emulation that allowed the IoT UI module to have access to accelerated graphics. [30] This change allowed the graphics to be calculated more efficiently, thus boosting the user interface software performance significantly. Animations became rather smooth and project reached the next bottleneck, hardware. At this stage, hardware performance was deemed sufficient.

At this point, all software configurations that target graphics and acceleration, had reach production quality goals. SDL will look for unix environment variable 'DISPLAY', which has to be set manually in the docker container. This is because

```
RUN apt-get install -y libmtdev-dev libfreetype6-dev libgl1-mesa-dev libgles2-mesa-dev libdrm-dev libgbm-dev libudev-dev libasound2-dev liblzma-dev libjpeg-dev libtiff-dev libwebp-dev git build-essential
RUN apt-get install -y gir1.2-ibus-1.0 libdbus-1-dev libegl1-mesa-dev libibus-1.0-5 libibus-1.0-dev libice-dev libsm-dev libsndio-dev libwayland-bin libwayland-dev libxi-dev libxinerama-dev libxkbcommon-dev libxrandr-dev libxss-dev libxt-dev libxv-dev x11proto-randr-dev x11proto-scrnsaver-dev x11proto-video-dev x11proto-xinerama-dev

RUN /bin/bash -c 'wget https://libsdl.org/release/SDL2-2.0.10.tar.gz \
   && tar -zxvf SDL2-2.0.10.tar.gz \
   && pushd SDL2-2.0.10 \
   && ./configure --enable-video-kmsdrm --disable-video-opengl --disable-video-x11 --disable-video-rpi \
   && make -j$(nproc) \
   && make install \
   && popd'
```

Listing 9: IoT software Dockerfile SDL configuration.

```
RUN apt-get -y install libgl1-mesa-glx libgl1-mesa-dri
ENV DISPLAY=:0
```

Listing 10: IoT software Dockerfile Mesa configuration.

X window server socket is symbolically linked from the host system to the container internals with container run options. Container run options are traditionally configured with a docker-compose YAML file, but in Edge Runtime, these are defined in deployment template (see listing 11). For this project, container create options must configure Binds (Docker-compose MOUNT equivalent) to attach X window server socket (`/tmp/X11-unix`), sound device (`/dev/snd` - Linux-Sound Subsystem) and rendering interface (`/dev/dri` - Direct Rendering Manager).

```
"settings": {
  "image": "${MODULES.uimodule}",
  "createOptions": {
    "HostConfig": {
      "NetworkMode": "host",
      "IpcMode":"host",
      "Binds": [
        "/tmp/X11-unix:/tmp/.X11-unix",
        "/dev/snd:/dev/snd",
        "/dev/dri:/dev/dri"
      ],
      "Env": [
        "DISPLAY=:0"
      ]
    },
    "NetworkingConfig": {
      "EndpointsConfig": {
        "host": {}
      }
    }
  }
}
```

Listing 11: IoT Edge module deployment template.

After setting up graphics dependencies on the container, app requirements can be installed, following with a base Kivy installation. For Kivy to understand fonts, they must be separately copied to a certain OS path under `/usr/share/fonts/truetype/`. Before finally running the UI software with docker *CMD* command, locales must be set like in listing 12. This is to deliver special abbreviations, such as date formats and certain characters common to nordics, like ‘Ä’.

UI software `awmain.py` is located in subfolder `aw_window` and is initialised in an asynchronous loop from `main.py`. Function getApp() in `awmain.py` defines the window size, positioning and returns a

`BaseApp()`

class, which is inherited from

`MDApp()`

```
COPY requirements.txt ./
RUN pip install -r requirements.txt
RUN pip install kivy[base]
COPY . .
COPY aw_window/*.ttf /usr/share/fonts/truetype/
# Set the locale
RUN apt-get -y install locales
RUN sed -i '/fi_FI.UTF-8/s/^# //g' /etc/locale.gen && locale-gen
ENV LANG fi_FI.UTF-8
ENV LANGUAGE fi_FI:fi
ENV LC_ALL fi_FI.UTF-8
ENV LC_TIME fi_FI.UTF-8
CMD [ "python", "-u", "./main.py" ]
```

Listing 12: IoT software Dockerfile locale and run config.

, which is inherited from

`App()`

, a Kivy application class. MD is an abbrevation for Material Design. It is an extension library for Kivy that has ready made basic widgets that follow material design philosophy.

```python
import sys
import asyncio
sys.path.insert(0, 'aw_window')
import awmain
if __name__ == "__main__":
    ui_instance = awmain.getApp()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(ui_instance.app_func())
    loop.close()
```

Listing 13: Edge module asynchronous entry point in `awmain.py`

The BaseApp class has an

`app_func()`

method (called in listing 13), which starts two asynchronous threads, shares a thread safe Queue between them and returns the thread pool. Multithreading here uses 'asyncio' library. The first thread's task is running the UI, and the second is to run Edge using Microsoft Azure IoT Edge Python module. It initialises the module and connects it to cloud. This way the Edge thread can be interacted with from cloud. More on, the Edge thread can use the Queue to invoke an action in the UI thread, with thread safety.

```python
def app_func(self):
    self.kq = KivyQueue()
    self.kq.set_callback(notify_func=self.on_queue)
    self.edge_task = asyncio.ensure_future(self.edge_agent(self.kq))
    async def run_wrapper():
        await asyncio.sleep(5) # So edge gets time to start
        await self.async_run(async_lib='asyncio')
        self.edge_task.cancel()
    return asyncio.gather(run_wrapper(), self.edge_task)
```

Listing 14: Custom class KivyQueue is inherited from Python internal Queue.

Edge thread invokes

`edge_agent()`

function, which takes the shared Queue reference as a parameter, so it can be fired on events. Using IoTHubModuleClient from Azure IoT Python library within the runtime, it is sufficient to call

`.create_from_edge_environment()`

to instantiate the module. The client instance is augmented with an event hook when the module's Digital Twin is updated (called patching). This event data is appended to a class variable, and added to the Queue as shown in listing 15. This way UI thread can react to the changes in real time without polling, since IoTHubModuleClient uses MQTT protocol under the hood. Same principal can be applied in order to register other events, such as remote method invocation.

UI thread in turn invokes an

```python
from azure.iot.device.aio import IoTHubModuleClient
...
    async def edge_agent(self, kq):
        module_client =
        ↪  IoTHubModuleClient.create_from_edge_environment()
        await module_client.connect()
        async def twin_dp_handler(patch):
            self.twin |= patch
            kq.put('twin', patch)
        module_client.on_twin_desired_properties_patch_received =
        ↪  twin_dp_handler
```

Listing 15: Twin patch handler attachment hook.

```
async_run()
```

method, which is inherited from Kivy's App() class. In the 'self' where this is called, the run method invokes a build() and an on_start() methods respectively. Usage of on_start() is shown in listing 16. In Kivy, the build() is ought to return a Kivy Widget() class. Widget() class has been inherited as

```
AwMainGrid(Widget)
```

. As a side note, code level ApartaView notations are labelled AW instead of AV. In the inherited build() method theme, locale and similar configurations are set. Inherited on_start() method starts all custom components that appear on the UI. Each method starts a custom component within Python code in order to provide custom functionality.

Each Kivy representation of a component is inherited from Widget(). Widget has a corresponding .kv file, which can be compared to a HTML document containing a stylesheet. Widget's .kv reference can be accessed in Widget's code via

```python
self.root
```

. Accessing the root, the markup can be browsed via identification reference, such as

```python
class AwMainGrid(Widget):
...
class BaseApp(MDApp):
    ...
    def build(self):
        ...
        return AwMainGrid()
    def on_start(self):
        ...
```

Listing 16: Instantiating Kivy app.

```python
self.root.ids.<my_widget_id>
```

. From here on, the reference returns a Python object that can be interacted with. Custom components are stored in subfolder `./widgets/<widget>.py` and their .kv file respectively from `./widgets/kv/<widget>.kv`. Yle (Yleisradio) news widget is a custom widget, and shown as an example initialisation in listing 17. Some components may include multiple .kv files, such as the Föli Widget for bus schedules. It has its own .kv file for each bus row. This simplifies iterating through the data, since the `bus_row.kv` can be loaded as a Widget instance, altered, and finally injected to the parent.

```python
from widgets.yle_widget import YleWidget
...
    def _init_news_app(self):
        self.root.ids.newsappcontainer.clear_widgets()
        self.newsapp = YleWidget()
        self.newsapp.init_news(CAROUSEL_SPEED_NEWS, NEWS_AMOUNT)
        self.root.ids.newsappcontainer.add_widget(self.newsapp)
```

Listing 17: Custom Kivy widget instance initialisation.

Kivy classes content definition can be extended in .kv files. These definitions apply for all instances that are the instance itself or of its children (inheritance). Definitions can be theming related like font adjustment, or, full-on Kivy widget instances.

```
<Widget>:
    font_name: 'Montserrat-Regular.ttf'
    font_size: 26

<AwMainGrid>:
    MDGridLayout:
        cols: 1
        size: root.width, root.height
```

Listing 18: Kivy widget class inheritable styles.

5.7 Distributed Systems and Authentication

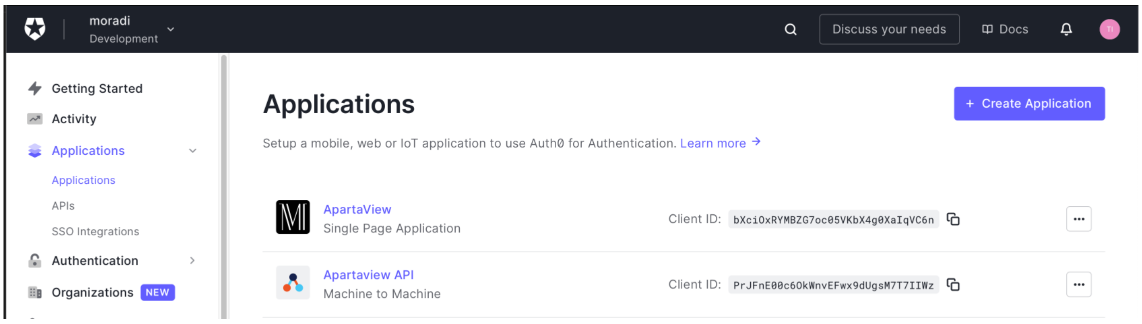
ApartaView's programs are distributed on multiple different technological layers, which all are required to follow scoped information sharing. As described earlier, Auth0 was chosen as the IDM for ApartaView. Furthermore, Auth0 is common to all technological layers including the IoT software (apartment ownership relation), ApartaView API (access management and scoping) and web UI (control panel login). Auth0 has a nice control panel and an API for our programs to interact with. Auth0 has a similar subscription philosophy as Azure, where putting up an Auth0 IDM results in having a dedicated Auth0 tenant. In this case, the tenant is moradi.eu.auth0.com, also known as Auth0 Domain.

Prior to using Auth0 API, the tenant must be configured via the Auth0 control panel. Auth0 tenant is ought to have two applications, one for ApartaView web UI and one for ApartaView API as shown in figure 12. Each need application URIs (Uniform Resource Identifier, listing 19) for various configurations; callback URLs, logout URLs, Web Origins and Origins (CORS, Cross Origin Resource Sharing). Configuration of these include app URLs in different environments, e.g. development machine and cloud hosted. This way other sources and/or destinations can not use the service, thus the IDM service access is scoped.

Implementing the IDM requires to correlate the configuration with practical use case. Here the situation is of such, that the person who logs in has to have autho-

```
http://localhost:4200,
http://localhost:3002,
https://view.moradi.fi,
https://api.view.moradi.fi
```

Listing 19: ApartaView uniform resource identifiers.



Applications

Setup a mobile, web or IoT application to use Auth0 for Authentication. Learn more →

ApartaView
Single Page Application
Client ID: bXciOxRYMBZG7oc05VKbX4g0XaIqVC6n

Apartaview API
Machine to Machine
Client ID: PrJFnE00c6OkWnvEFwx9dUgsM7T7IIWz

Figure 12. Auth0 Applications in ApartaView project.

risation to the devices, which lie in such staircases, that belong in those apartment companies that the person is ought to have access to. This assignment is called a role. Users can simply be attached to certain roles via non-code configuration. Those roles can be added to the token with a rule. This rule is ran on every login, during which a JWT token is generated. Namespace for the roles in this case is `http://moradi.fi/roles`. Since roles represent the apartment companies, this can be utilised to filter the devices that the user is ought to have access to. Devices have ownership information in the Module Digital Twin, which is utilised here. Henceforth, technically the users don't need management access to the Device Twin, but instead only to the 'uimodule'.

```
function (user, context, callback) {
    const namespace = 'http://moradi.fi';
    const assignedRoles = (context.authorization || {}).roles;

    let idTokenClaims = context.idToken || {};
    let accessTokenClaims = context.accessToken || {};

    idTokenClaims[`${namespace}/roles`] = assignedRoles;
    accessTokenClaims[`${namespace}/roles`] = assignedRoles;

    context.idToken = idTokenClaims;
    context.accessToken = accessTokenClaims;

    callback(null, user, context);
}
```

Listing 20: Auth0 Rule for role injection.

```
{
    "token": "eyJhbGciOi ...",
    "message": {
      "aud": [
        "https://api.view.moradi.fi/",
        "https://moradi.eu.auth0.com/userinfo"
      ],
      "azp": "bXciOxRYMBZG7ocO ...",
      "exp": 1638615371,
      "http://moradi.fi/roles": [
        "tkk8"
      ],
      "iat": 1638528971,
      "iss": "https://moradi.eu.auth0.com/",
      "scope": "openid profile email",
      "sub": "auth0|60a8c323802b880068698055"
    }
}
```

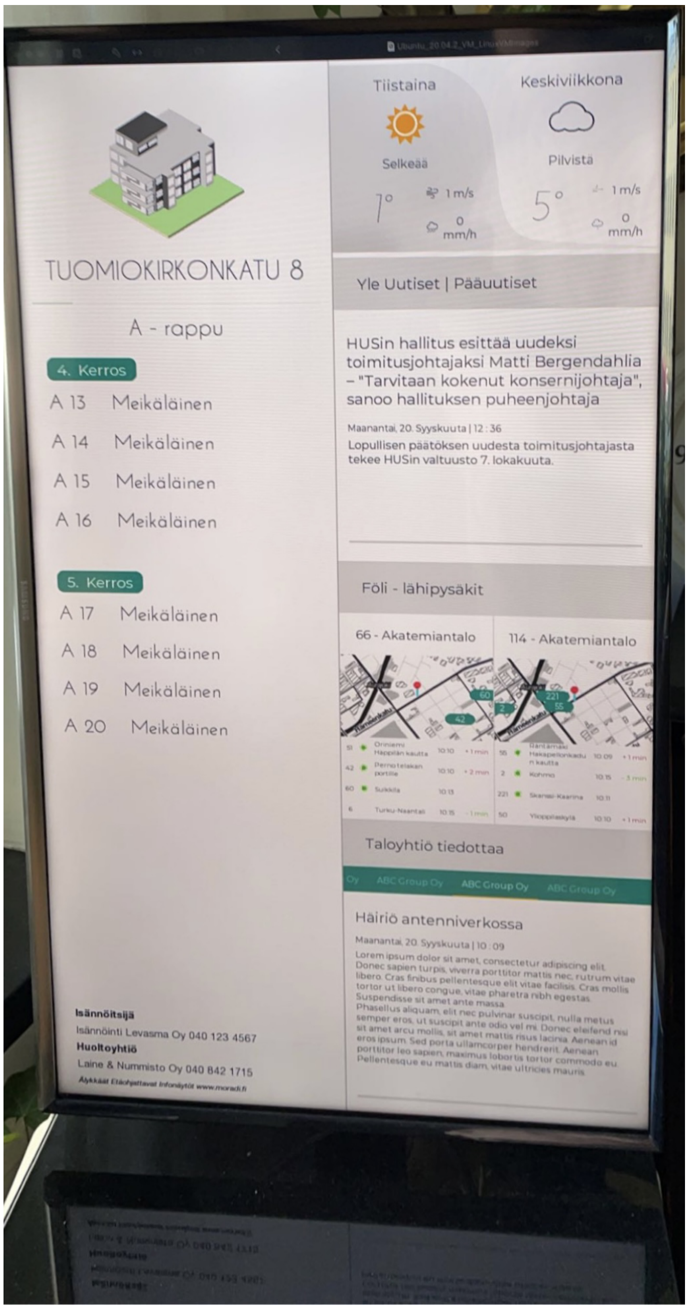
Listing 21: Identity token structure for company `tkk8`.

6 Productisation

A core section of computer engineering is moulding a project into a product. Modern IT productisation often means that the moulded product is a service of some sort. ApartaView productisation involves offering the ensemble as a service. In this case, ApartaView is PaaS (Platform as a Service).

On an abstract level, productisation towards a service is similar to food chain development. A restaurant is not interested to buy livestock, but a refined version of the livestock, such as a packaged tenderloin. The tenderloin may be the product, but the restaurant might be more interested to order a service from the farmer. In this analogy, the livestock's refined products can be offered as a service, such as a subscription. While subscribed, the restaurant gets all the tenderloin it needs for a fixed price, defined in the contract. Hence, the responsibility of tenderloin deliverance is shifted for the farmer. With correct subscription pricing, both benefit. The restaurant pays a margin for the good service and the farmer gets a better price for the livestock. Because the farmer has more expertise in transforming a livestock to an amount of tenderloin, the farmer has an upper hand in demand forecasting. There are more risks for the farmer, but with good service contract design, damages can be mitigated.

A similar fashion is followed in ApartaView productisation. Apartment managers have no expertise in maintaining the displays in their staircases. Due to this, ApartaView software is bundled to hardware. The software presented in this thesis case is delivered to apartments with all parts of the ensemble infused together, as shown in figure 13. This way, the software can be tailored to certain hardware environment in order to provide a better customer experience. Henceforth, with a service model ApartaView supplier can charge a margin within computer engineering domain expertise and the apartment manager does not need to worry about maintaining the system. However, the device package can cost up to 2000€ at this stage, while a suitable subscription price is around 50-100€ a month based on empiric research. In cases like this, it is suitable to charge a sign-up margin, i.e. a small upfront payment for the device to compensate return of investment. More on, ApartaView supplier can offer multiple service and hardware configurations that have different service levels. This gives supplier the ability to offer optional extensions for a different price (service tailoring), such as a touch screen and a more interactive version of the software bundled together.



A 13	Meikäläinen
A 14	Meikäläinen
A 15	Meikäläinen
A 16	Meikäläinen
A 17	Meikäläinen
A 18	Meikäläinen
A 19	Meikäläinen
A 20	Meikäläinen

Isännöitsijä
Isännöinti Levasma Oy 040 123 4567
Huoltoyhtiö
Laine & Nummisto Oy 040 842 1715
Älykkäät Etäohjattavat Infonäytöt www.moradi.fi

Yle Uutiset | Pääuutiset

HUSin hallitus esittää uudeksi toimitusjohtajaksi Matti Bergendahlia – "Tarvitaan kokenut konsernijohtaja", sanoo hallituksen puheenjohtaja

Maanantai, 20. Syyskuuta | 12 : 36
Lopullisen päätöksen uudesta toimitusjohtajasta tekee HUSin valtuusto 7. lokakuuta.

66 - Akatemiantalo

51	Oriniemi Häppilän kautta	10:10	+ 1 min
42	Pernotelakan portille	10:10	+ 2 min
60	Suikkila	10:13	
6	Turku-Naantali	10:15	- 1 min

114 - Akatemiantalo

55	Rantamäki Hakapellonkadun kautta	10:09	+ 1 min
2	Kohmo	10:15	- 3 min
221	Skanssi-Kaarina	10:11	
50	Ylioppilaskylä	10:10	+ 1 min

Taloyhtiö tiedottaa

Oy ABC Group Oy ABC Group Oy ABC Group Oy

Häiriö antenniverkossa

Maanantai, 20. Syyskuuta | 10 : 09
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec sapien turpis, viverra porttitor mattis nec, rutrum vitae libero. Cras finibus pellentesque elit vitae facilisis. Cras mollis tortor ut libero congue, vitae pharetra nibh egestas. Suspendisse sit amet ante massa. Phasellus aliquam, elit nec pulvinar suscipit, nulla metus semper eros, ut suscipit ante odio vel mi. Donec eleifend nisi sit amet arcu mollis, sit amet mattis risus lacinia. Aenean id eros ipsum. Sed porta ullamcorper hendrerit. Aenean porttitor leo sapien, maximus lobortis tortor commodo eu. Pellentesque eu mattis diam, vitae ultricies mauris.

Figure 13. Base model laboratory IoT device running ApartaView.

6.1 Verification, testing and QA

Delivering ApartaView PaaS for apartments means that the platform must be tested and developed in an isolated environment. Having separate environments gives developer team flexibility to test and roll out new features and changes. Isolated environments are reflected in Azure as resource groups, DEV for development and PROD for production. PROD resources facilitate all actual customer deployments. Each interface of each instance in each group has its own namespace. Namespaces enable environment targeting, which can be configured in environment variables of each software of the ensemble. For example, ApartaView API has three types of instances; localhost when running on development machine, `dev.api.view.moradi.fi` for cloud hosted development environment and `api.view.moradi.fi` for cloud hosted production environment. Cloud hosted environment in this context means accessible over the internet and hosted by a CSP.

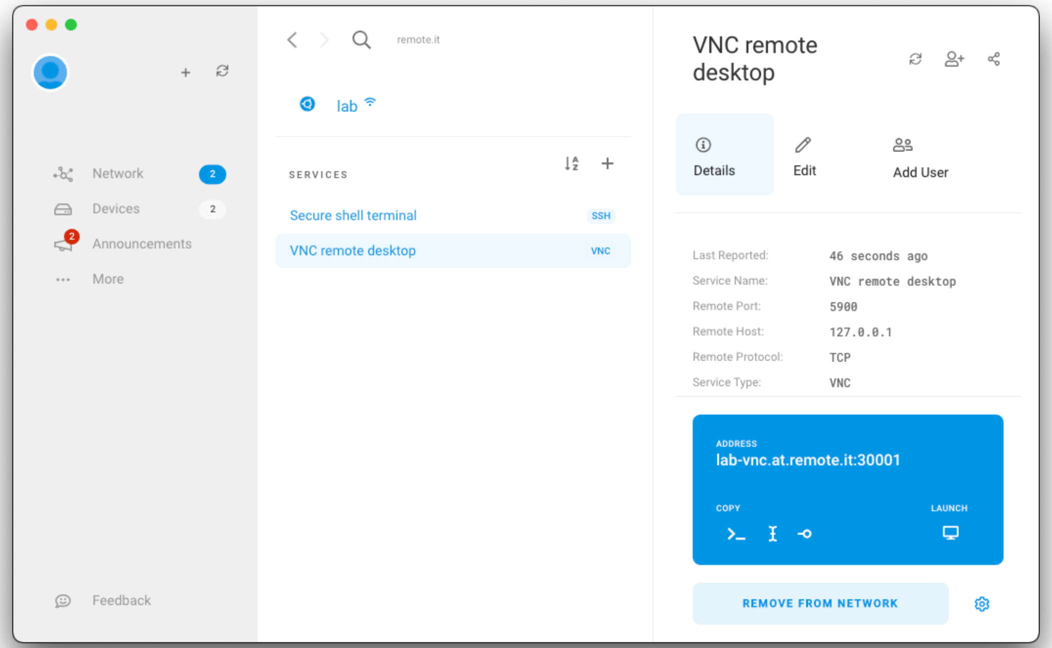
Workflow for rolling out a new version of a web service (API or UI) involves developing the changes initially on the developer's machine. API and UI can be ran simultaneously on development machine, which allows developing changes that are dependent on one or the other. However, testing changes that have been made is somewhat limited. Changes that require the IoT software are verified and tested on DEV namespace. This is done by packaging and shipping the API or UI software to live DEV servers in the same manner, as it would be published to PROD. After verifying that the changes work as intended, the same publishing principal is applied for PROD to roll out new changes to customers (apartments).

IoT software changes are rolled out with similar philosophy as API and UI but with different practical steps. IoT software is dependent on the hosting operating system, thus local development is ran on a virtual machine. A virtual machine in this case is an Ubuntu 20.04 with Azure IoT Edge Runtime installed. This virtual machine is registered in Azure as a device, thus available for deployment. The software package is containerised in the standard fashion and deployed like it would be deployed to any device. After verifying the software changes within the virtual machine, the same manifest is deployed to LAB (laboratory) resource. The LAB resource is simply a near identical set of hardware in comparison to what is installed in apartment staircases. This way, the software can be verified and tested with actual equipment prior to live PROD deployment. Some updates may require external optional hardware, which naturally are not available within a virtual machine.

Upon rolling out changes to production, they must be verified. In a more traditional software project, production changes can usually be verified physically from the development machine. For API and UI projects, this is done with regular web tools, like Postman and a web browser. However, having IoT software with a graphical user interface, the changes need to be verified by visiting the install site. This is fairly impractical, since the install sites can be really remote. In order to overcome this major problem, the IoT device's display must be accessible remotely from the developer's machine. Hence, a VNC (Virtual Network Computing) server is required on all IoT devices. The VNC server is piped to X window server socket. A reverse proxy must be installed alongside to access the VNC remotely, since the IoT devices are ought not to have a static public IP address and VPN (Virtual Private Networking) was deemed too traditional for this project. In ApartaView case, *Remote.it* was chosen to deliver reverse proxying to devices for quality assurance in production. Upon installation, remote.it runtime is configured to facilitate SSH and VNC as seen in figure 14 screenshot. If either needs to be accessed, reverse proxy can be generated from remote.it control panel.

6.2 Ensuring service continuity

A high priority open task in ApartaView project is to install and configure CI/CD Pipelines, which stand for Continuous Integration, Continuous Delivery. CICD pipe removes the compilation and deployment from the workflow. It takes the source code as input and a running software as a result. Even though codebase does not need compilation (Python), the deployments in ApartaView project are complex, hence there is a need for this kind of functionality. This functionality will be provided



Last Reported: 46 seconds ago
Service Name: VNC remote desktop
Remote Port: 5900
Remote Host: 127.0.0.1
Remote Protocol: TCP
Service Type: VNC

ADDRESS lab-vnc.at.remote.it:30001

Figure 14. VNC proxy access from Remote.it management panel.

by Azure DevOps services. CICD pipes for API and UI are quite straight forward, Python source only needs to be packaged and shipped to an Azure App Service resource, according to environment (DEV or PROD).

CICD pipe for IoT software is a bit more complex, but Azure DevOps offers an Edge Module CICD template. In order for the CICD not to get too difficult to maintain, each device has the same software shipped to it. However, some customers may have different configurations due to extensions and layouts. Extensions are defined in subscription levels, which are stored in the database. Thus, IoT software extension and layout configurations are fetched from ApartaView API upon app start up and initialised accordingly.

6.3 Extension support and modularity

ApartaView project is designed to be easily alterable by the developers and the customers to a certain degree. Future desires and legislations may set requirements, for example for accessibility, to which ApartaView philosophy aims to answer. Extensions, that are on the roadmap are vision impairment support, anonymous resident detection, touch controls, speech control and an external mini touch display. Digital signage adaptation to individual preferences is found fruitful. [31] Developing abovementioned is not possible in a virtual machine. Extensions are designed in LAB environment, where extension hardware can be installed on an IoT device during software development.

Some of the extensions have features involving sensors, such as eye tracking. These can be subject to monetisation, which is dependent on data gathering. Gathering sensitive data has a wide variety of limiting regulations that need to be followed. [32] For example, EU's *GDPR* (General Data Protection Regulation) defines that data gathering of persons needs consent if a data subject is identifiable. The regulation also defines what can be counted as identifiable, and is ought to be followed when designing extensions, especially monetised ones. [33]

6.4 Existing digital signage product market

There were a couple of other similar digital signage solutions compared to this thesis' subject case. They were observed during market research online alongside visiting new and old apartments. Most of the researched implementations lack in advanced technologies, such as remote controllability and extendability. They provide basic functions, but some do integrate with the local infrastructure, including bus schedules.

Since the scope of the thesis is apartment environments, only on those will be in focus. Other pivots can for example be advertising, restaurants, culture and teaching environments. Nonetheless, it's important to keep in mind, that most of the other solutions are designed for multiple targets and it reflects in the extensions and integrations they offer for the product. Due to the scope of the thesis, a production quality infotainment screen for apartments has the following minimum requirements: computing hardware, display hardware, infotainment software and means of control or management.

6.5 Products with major similarities

First implementation in examination is a product from iDiD Digital Signage company (`idid.fi/`). The solution offered by this company seems to be the most advanced available in Finland. They offer multiple basic modules and provide integration services. Their solution seems to have a base software that is engineered separately for each customer using existing integrations. The solution is applicable for many different environments, not only apartments. Since access to their technology of choice is not available, there is no certainty of how their technological implementation suits all environments' requirements. [34]

However, from my startup background knowledge, I expect a lack of quality in some aspects of iDiD when offering a solution that is designed to fit all needs. All environments have a few common aspects that are subject to solution quality. Major aspects include remote controllability, fleet management, concurrence, uptime and software plasticity to customers' needs. In terms of features for the display, they offer a remote management panel, an app and a content creation tool.

On their website they state, that the content in produced purely in HTML, which clearly defines the limits of their solution's flexibility. Therefore, customer can not create interactive components and extendability is limited. On the contrary, they already offer standard connectors for a dozen of services, such as PowerBi, Wilma and Taloyhtio.info. Also, their website states that it runs on multiple operating systems, implicating that they are rather software orientated than having extended hardware support other than a touch screen.

6.6 Products with minor similarities

Second solution in examination is provided by Fisplay (`fisplay.com/`). They merely produce an elementary and less advanced software. They don't have much information on their website, neither did the website work correctly during writing. Their solution is very static and does not offer much room for flexibility, let alone extension support, such as pointer devices or controllers. [35]

Third one in peer inspection is FirstView MediaCloud (`firstview.fi/`). According to their website, they offer a remote controllable media software for devices they order from another company. Their media software has multiple integrations and has the necessary capabilities for apartment environment implementation. However, they do not mention apartment infotainment display solutions on their website. FirstView seems to be more focused on business environments. [36]

7 Discussion

At first, I was attempting to make my thesis on another project from the current company I worked in during the writing of thesis. However, the customer stalled the start of the project. This situation was unwanted, since I wanted to graduate. After approximately a year, I founded the ApartaView project. At that time I realised that not only did I want to graduate, I also wanted to do it for a driven purpose. This combination seemed near impossible, but I managed to succeed on both matters. Setting up a company is only one of many examples of what was making ApartaView challenging, which were out of scope for the thesis. However, I plan to pursue the thesis case further, since the case's long term viability sparks my interest. Also, my bachelors thesis involved a technology startup, which I founded at the time of it's writing. [37] With the product Plate Analytics [38] of the startup, I also graduated from Founders Institute, a Silicon Valley accelerator institute. [38] Regardless of ApartaView's journey being difficult, it is safe to conclude that past projects and an entrepreneurial mindset had a positive impact.

The ApartaView development had many challenging cornerstones. It was rather difficult to lead the whole ensemble on my own, in contrary to having a mentor and a ready project from a company. Core of the thesis evolves around IoT. However, since an IoT product usually needs other elements around it, the project grew significantly. The web services and server configurations could have been a thesis project of their own. Because of this, thesis scoping required a couple pivots during the writing. Another engineering challenge that presented itself in the middle of the project was getting the IoT UI through a container and a runtime with sufficient performance. Developing the API and UI also took effort worth mentioning.

Significant amount of time was used on engineering certain features, which later on proved themselves unnecessary. Amongst these is the usage of Raspberry Pi ARM platform. First task for an IoT device is the OS and environment setup. In the early stages of the project, the OS and environment was dependent on ARM architecture. Thus, it needed to be redesigned entirely. Dietpi was chosen as the OS for the ARM platform. [39] It would have been very nice of Azure Edge Runtime to support display sockets on ARM machines, since Dietpi OS had a lot of configuration options suitable for ApartaView. The Azure IoT Edge + ARM display socket incompatibility is a quite specific bug, and appeared to not even being reported before.

Azure IoT Edge could have been replaced by another Message Queue (MQ) and Kubernetes container delivery service. A reliable MQ example is Apache AcitveMQ, which has industry approval for production targets. ActiveMQ has very good performance and reliability ratings, whereas Azure IoT Edge is yet not as established. [40] Kubernetes in turn would have provided remote deployment for IoT devices. Henceforth, ActiveMQ with Kubernetes could have been a better option.

A part of sustainability and feasibility hypothesis was to determine if CSP's could offer the necessary functionality. Even though Azure IoT Edge had a couple of quirks and bugs along the way, it doesn't implicate that other CSP's would suffer the same. For example, AWS GreenGrass could have been tested after the selected architecture components encountered issues – with a technology pivot. It is open for discussion, whether another CSP would have suited the thesis case better. Results have shown AWS GreenGrass to be more resource efficient in comparison to Azure IoT Edge, but has a different operating philosophy that affects the results. [41] Nonetheless, thesis case showed successful results running UI software through ready-made IoT components from Microsoft Azure CSP.

Overall, the system built here has unfair advantages to competitors in terms of technology selection. The technology that was used to deliver this service heavily supports modularity and extendability by design. Overviewing the whole context, there is only one major tech competitor, but most competition is still physical, i.e.

physical notification boards and physical resident lists. For the latter, the system gives services that haven't been available before, such as live bus schedules and daily notifications that are immediately available.

As discussed in the productisation chapter, the tech competitor iDiD does not offer extendability in the fashion that ApartaView offers. There is a significant difference in product design, where iDiD developers tailor the software via programmable or configurative changes. In turn, ApartaView offers tailorability for the customer, i.e. apartment managers. They can themselves change the layouts and choose desired extensions. In turn, ApartaView product developers can consult the apartment managers on what the residents could find useful. Technology burden is shifted from the customer whilst the customer still gets multiple configuration options that they can tailor to suit their needs better. Even in the case where the customer insists, they can also make their own templates too in Kivy markup language. Digital signage context adaptation is found useful and attractive. [42]

8 Conclusions

All technology specifics aside, major focus of thesis was to find out whether a generic IoT solution workflow was feasible considering graphical software. Thesis work was ought to cover development, testing, production, administration and tailored management in terms of acceptable final state of outcome. All of the ensemble modules' development showed that the thesis case was not a common solution, despite being modern and straight forward. Lack of previous research is probably affected by rarity and case specificity. Even though the case development followed best practices, there were many engineering challenges along the way, such as containerisation in execution chapter.

Working towards the hypothesis, whether it is feasible to use IoT with graphical software, was conveyed by the case process more than the theory related to it. Based on case research and methodologies used, having the correct elements make a solution like this possible. This doesn't yet prove the hypothesis correct. More on, the elements provide decent reusability to facilitate extendability requirements and mouldability capabilities. For example, Kivy framework allows templating. Since the templates can be dynamically loaded without exiting the application, the requirement constraints are fully met. Thus, a general conclusion is that hypothesis was proven correct and the system was feasible to build. However, nature of computer engineering shows that there are usually many different methods to solve a single communications technology problem. Those are debated and the best one is chosen.

The *many different methods* principle applies also in this thesis. Azure IoT Edge was proven to work as the IoT platform, because it is stable and gives room for dynamic moulding via Digital Twin functionality. However, it proposed some problems, such as the ARM incompatibility specific to UI containers. Most Single Board Computers (SBCs) are on ARM platform, thus supporting ARM would be beneficial to the overall solution. In conclusion, Azure IoT Edge might have not been the best platform choice for IoT in the nature of the thesis case. Regardless of this perspective, the solution on x86 platform is sustainable, therefore sustainability hypothesis is valid, because ARM compatibility was not a requirement.

It was also up for question whether the feasible IoT solution with surrounding elements had the ingredients to be a full on sellable product. Productisation chapter pointed out, that similar solutions existed on the market as products. However, their flexibility and degree of conservativity was up for question. Hypothesis was that a modern IoT system could replace current market products, such as physical notification boards and other similar digital signage products. Note, that replacing requirement is in a future proof fashion, in such way, that the ensemble could be extended without additional systems. By concatenating the foundation of the previous hypothesis with replicability and user management, thesis case solution as a whole can be affirmed feasible and sustainable. This is due to the characteristics required for productisation being fulfilled based on the thesis productisation research.

List of source codes

List of Figures

References

[1] I. Rašan, “Digitalization in the role of humanization or dehumanization of modern society,” in *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, pp. 1460–1465, 2021.

[2] N. Moore, “The information society,” tech. rep., World Information Report, 1997.

[3] P. F. Webster and F. Webster, *Theories of the Information Society*. Routledge, 2002.

[4] Helsingin Kaupunki and Kaupunkisuunnitteluvirasto, *Kerrostalojen kehittäminen - Talotyyppiselvitys*. Edita Prima, 2007.

[5] M. J. Katz, *From research to manuscript : a guide to scientific writing*. Dordrecht: Springer, 2nd ed. ed., 2009.

[6] T. Oetiker, “The Not So Short Introduction to LaTex2e: Or LaTex2e in 131 Minutes,” *At http://www.tex.ac.uk/*, 2004.

[7] A. Verma, S. Prakash, V. Srivastava, A. Kumar, and S. C. Mukhopadhyay, “Sensing, controlling, and iot infrastructure in smart building: A review,” *IEEE Sensors Journal*, vol. 19, no. 20, pp. 9036–9046, 2019.

[8] K. Lawal and H. N. Rafsanjani, “Trends, benefits, risks, and challenges of iot implementation in residential and commercial buildings,” *Energy and Built Environment*, 2021.

[9] T. K. Sree, V. Swetha, M. Sugadev, and T. Ravi, “Iot based rgb led information display system,” in *Congress on Intelligent Systems*, pp. 431–442, Springer, 2020.

[10] T. Kubitza, A. Voit, D. Weber, and A. Schmidt, “An iot infrastructure for ubiquitous notifications in intelligent living environments,” in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, UbiComp ’16, (New York, NY, USA), p. 1536–1541, Association for Computing Machinery, 2016.

[11] P. P. Gaikwad, J. P. Gabhane, and S. S. Golait, “A survey based on Smart Homes system using Internet-of-Things,” in *4th IEEE Sponsored International Conference on Computation of Power, Energy, Information and Communication, ICCPEIC 2015*, 2015.

[12] R. T. Fielding and R. N. Taylor, “Principled Design of the Modern Web Architecture,” *ACM Transactions on Internet Technology*, 2002.

[13] S. Agrawal and M. L. Das, “Internet of things - A paradigm shift of future internet applications,” in *2011 Nirma University International Conference on*

Engineering: Current Trends in Technology, NUiCONE 2011 - Conference Proceedings, 2011.

[14] I. Kapetanovic, S. Mujacic, S. Kasapovic, and S. Kulaglic, “Integration of streaming and digital signage technology for higher education,” in *2012 20th Telecommunications Forum (TELFOR)*, pp. 1300–1302, 2012.

[15] Amazon Corporation, “Cloud Services - Amazon Web Services (AWS).” https://aws.amazon.com. Visited 20.1.2022.

[16] Microsoft, “Microsoft Azure: Cloud Computing Services.” https://azure.microsoft.com. Visited 20.1.2022.

[17] Google, “Google Cloud: Cloud Computing Services.” https://cloud.google.com. Visited 20.1.2022.

[18] Remot3.it, “Remote.it.” https://remote.it, 2021. Visited 15.12.2021.

[19] Electronic Frontier Foundation, “Certbot.” https://certbot.eff.org, 2021. Visited 15.12.2021.

[20] M. Fine, *Beta Testing for Better Software.* Wiley Computer Publishing, 2002.

[21] Google, “Angular - The modern web developer’s platform.” https://angular.io, 2021. Visited 15.12.2021.

[22] The Pallets Projects, “Flask.” https://flask.palletsprojects.com/en/2.0.x/, 2021. Visited 15.12.2021.

[23] Python Software Foundation, “functools - Higher-order functions and operations on callable objects.” https://docs.python.org/3/library/functools.html. Visited 15.12.2021.

[24] Netcraft, “November 2021 Web Server Survey.” https://news.netcraft.com/archives/2021/11/23/november-2021-web-server-survey.html, 2021. Visited 15.12.2021.

[25] Microsoft, “App Service Pricing.” https://azure.microsoft.com/en-gb/pricing/details/app-service/windows/?cdn=disable#pricing, 2021. Visited 15.12.2021.

[26] H. Kurono and Y. Shibuya, “Non-contact digital signage allowing users to change contents with their face orientation,” in *2020 IEEE 9th Global Conference on Consumer Electronics (GCCE)*, pp. 747–751, 2020.

[27] Raspberry Pi Foundation, “Raspberry Pi 4.” https://www.raspberrypi.com/products/raspberry-pi-4-model-b/, 2021. Visited 15.12.2021.

[28] Raxda Limited, “Rock Pi X.” https://wiki.radxa.com/RockpiX, 2021. Visited 15.12.2021.

[29] SDL Project and S. Lantinga, "SDL Simple Direct Media Layer." https://www.libsdl.org, 2021. Visited 15.12.2021.

[30] B. Paul, "Mesa 3-D Graphics Library." https://www.mesa3d.org. Visited 15.12.2021.

[31] T. Ogi, Y. Tateyama, and Y. Matsuda, "Push type digital signage system that displays personalized information," in *2014 17th International Conference on Network-Based Information Systems*, pp. 411–415, 2014.

[32] N. Shilov, O. Smirnova, P. Morozova, N. Turov, M. Shchekotov, and N. Teslya, "Digital signage personalization for smart city: Major requirements and approach," in *2019 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, pp. 1–3, 2019.

[33] European Commission, "EU General Data Protection Regulation (GDPR): Personal Data," 2018.

[34] iDiD Digital Signage, "iDiD - helppokäyttöinen sisällönhallintajärjestelmä." https://www.idid.fi. Visited 15.12.2021.

[35] Fisplay Oy, "Fisplay - Liikkuva kuva ei sammaloidu." https://fisplay.com. Visited 15.12.2021.

[36] First Technology Oy, "FirstView Digital Signage." https://www.firstview.fi/ratkaisut/digital-signage/. Visited 15.12.2021.

[37] T. Moradi, "Teknisen startupin Lean-prosessi (Lean process of a technology startup)," 2019. Department of Future Technologies, University of Turku.

[38] Founder Institute, "Plate Analytics - A Founder Institute Portfolio Company." https://fi.co/insight/feeding-the-future-top-fi-food-tech-and-agriculture-portfolio-companies, 2019. Visited 15.12.2021.

[39] D. Knight, "Dietpi OS for Single Board Computers." https://dietpi.com, 2021. Visited 15.12.2021.

[40] V. M. Ionescu, "The analysis of the performance of rabbitmq and activemq," in *2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)*, pp. 132–137, 2015.

[41] A. Das, S. Patterson, and M. Wittie, "Edgebench: Benchmarking edge computing platforms," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 175–180, 2018.

[42] W. Lee, H.-W. Lee, Y.-T. Lee, W. Ryu, and M. Choi, "Content management system for environment adaptive digital signage," in *2014 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 528–529, 2014.