

Project Report

Bachelor Project – Electricity Recording Assistant



Supervisors:

Poul Væggemose

Students:

Gais El-AAsi - 279910

Marcel Valentijn Daniel Notenboom - 279963

*Number of characters incl. space, own figures and tables (800 char/illustration)
and footnotes: 145682 characters, 60.7 pages (2400 char/page)*

Software Engineering Bachelor Project - VIA University College Campus Horsens

17th of December 2021

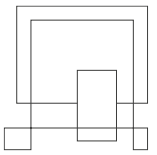
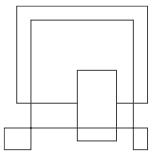


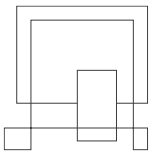
TABLE OF CONTENT

ABSTRACT

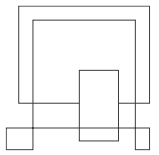
| | |
|--|----|
| INTRODUCTION | 1 |
| ANALYSIS | 4 |
| GLOSARY | 4 |
| REQUIREMENTS | 5 |
| <i>Functional Requirements</i> | 5 |
| <i>Non-functional Requirements</i> | 7 |
| <i>Delimitations</i> | 7 |
| USE CASE DIAGRAM | 8 |
| <i>Actors</i> | 8 |
| <i>Use Case Definitions and Descriptions</i> | 9 |
| ACTIVITY DIAGRAMS | 13 |
| SYSTEM SEQUENCE DIAGRAM | 14 |
| DOMAIN MODEL | 15 |
| <i>Domain Model Description</i> | 16 |
| TEST CASES | 16 |
| <i>User Acceptance Testing</i> | 17 |
| <i>System Testing</i> | 18 |
| DESIGN | 19 |
| ARCHITECTURE | 19 |
| <i>High-Level Architecture</i> | 19 |
| <i>Component Architecture</i> | 20 |
| <i>Cloud Architecture</i> | 21 |
| <i>Data Flow</i> | 22 |
| <i>DevOps Architecture</i> | 22 |
| TECHNOLOGIES | 23 |
| <i>Tech Stack</i> | 23 |
| <i>Cloud Provider and Cloud Services</i> | 25 |
| DESIGN PATTERNS | 26 |
| <i>Dependency Injection</i> | 26 |
| <i>Options Pattern in ASP.NET Core</i> | 27 |



| | |
|--|-----------|
| <i>Repository Pattern</i> | 28 |
| CLASS DIAGRAM | 28 |
| <i>Frontend Application Class Diagram</i> | 28 |
| <i>Backend Application Class Diagram</i> | 30 |
| <i>Embedded Application Class Diagram and Flow Diagram</i> | 33 |
| DEVICE SCHEMATICS | 35 |
| SEQUENCE DIAGRAM | 37 |
| DATABASE STRUCTURE | 40 |
| <i>PostgreSQL Database</i> | 40 |
| <i>Firestore Database</i> | 41 |
| USER INTERFACE AND USER EXPERIENCE | 42 |
| <i>UI Library</i> | 42 |
| <i>Charting Library</i> | 43 |
| SECURITY | 45 |
| <i>Cloud Provider Security Configurations</i> | 45 |
| <i>Security through code and development</i> | 46 |
| IMPLEMENTATION | 47 |
| EMBEDDED APPLICATION | 47 |
| BACKEND APPLICATION | 50 |
| FRONTEND APPLICATION | 54 |
| CI/CD IMPLEMENTATIONS..... | 58 |
| TESTING | 61 |
| UNIT TESTS | 61 |
| RENDERING TESTS | 63 |
| USABILITY TESTS | 65 |
| <i>Scenarios</i> | 66 |
| <i>Process</i> | 66 |
| <i>Results</i> | 66 |
| ACCEPTANCE TESTS | 68 |
| SYSTEM TESTS | 69 |
| RESULTS AND DISCUSSIONS | 72 |
| CONCLUSIONS | 75 |

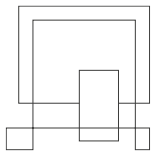


| | |
|--|-----------|
| PROJECT FUTURE | 76 |
| SOURCE OF INFORMATION | 1 |
| APPENDICES..... | A |
| APPENDIX A – PROJECT DESCRIPTION | A |
| APPENDIX B – USER GUIDE | A |
| APPENDIX C – ADMIN GUIDE | A |
| APPENDIX D – MARKET RESEARCH | A |
| APPENDIX E – USE CASE DIAGRAM..... | A |
| APPENDIX F – USE CASE DESCRIPTIONS..... | A |
| APPENDIX G – ACTIVITY DIAGRAMS..... | A |
| APPENDIX H – SYSTEM SEQUENCE DIAGRAM..... | A |
| APPENDIX I – DOMAIN MODEL..... | A |
| APPENDIX J – ACCEPTANCE TESTS AND RESULTS..... | A |
| APPENDIX K – SYSTEM TESTS AND RESULTS..... | A |
| APPENDIX L – ARCHITECTURE DIAGRAMS..... | A |
| APPENDIX M – COMPONENT DIAGRAM..... | A |
| APPENDIX N – DATA FLOW DIAGRAM..... | A |
| APPENDIX O – CI-CD DIAGRAMS..... | A |
| APPENDIX P – CLASS DIAGRAMS | A |
| APPENDIX Q – FLOW DIAGRAM | A |
| APPENDIX R – DEVICE SCHEMATICS | A |
| APPENDIX S – SEQUENCE DIAGRAMS | A |
| APPENDIX T – DATABASE DIAGRAMS | A |
| APPENDIX U – SOURCE CODE | A |
| APPENDIX V – USABILITY TESTS AND RESULTS..... | A |
| APPENDIX W – DRAW.IO DIAGRAMS FILE..... | A |

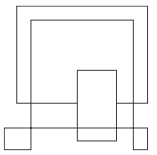


LIST OF FIGURES

| | |
|---|----|
| FIGURE 1 - MECHANICAL PLUG-IN TIME SOCKET | 2 |
| FIGURE 2 - USER STORY GENERAL FORM..... | 5 |
| FIGURE 3 - USE CASE DIAGRAM | 8 |
| FIGURE 4 - ACTIVITY DIAGRAM - MANAGE DEVICE..... | 13 |
| FIGURE 5 - SYSTEM SEQUENCE DIAGRAM - MANAGE DEVICE – SUNNY SCENARIO..... | 14 |
| FIGURE 6 - SYSTEM SEQUENCE DIAGRAM - MANAGE DEVICE - FAILURE SCENARIO | 15 |
| FIGURE 7 - DOMAIN MODEL..... | 16 |
| FIGURE 8 - ARCHITECTURE DIAGRAM | 19 |
| FIGURE 9 - COMPONENT ARCHITECTURE DIAGRAM..... | 20 |
| FIGURE 10 - CLOUD ARCHITECTURE DIAGRAM | 21 |
| FIGURE 11 - DATA FLOW DIAGRAM..... | 22 |
| FIGURE 12 - CI/CD PIPELINE | 23 |
| FIGURE 13 - REPODB REPOSITORY PATTERN | 28 |
| FIGURE 14 - FRONTEND APPLICATION CLASS DIAGRAM PART I..... | 29 |
| FIGURE 15 - FRONTEND APPLICATION CLASS DIAGRAM PART II..... | 29 |
| FIGURE 16 - FRONTEND APPLICATION CLASS DIAGRAM PART III..... | 30 |
| FIGURE 17 - BACKEND APPLICATION CLASS DIAGRAM PART I..... | 30 |
| FIGURE 18 - BACKEND APPLICATION CLASS DIAGRAM PART II..... | 31 |
| FIGURE 19 - BACKEND APPLICATION CLASS DIAGRAM PART III..... | 31 |
| FIGURE 20 - BACKEND APPLICATION CLASS DIAGRAM PART IV | 32 |
| FIGURE 21 - BACKEND APPLICATION CLASS DIAGRAM PART V | 33 |
| FIGURE 22 - EMBEDDED APPLICATION CLASS DIAGRAM..... | 33 |
| FIGURE 23 - FLOW DIAGRAM PART I..... | 34 |
| FIGURE 24 - FLOW DIAGRAM PART II..... | 35 |
| FIGURE 25 - DEVICE SCHEMATIC | 36 |
| FIGURE 26 - SEQUENCE DIAGRAM PART I..... | 37 |
| FIGURE 27 - SEQUENCE DIAGRAM PART II..... | 38 |
| FIGURE 28 - EMBEDDED SEQUENCE DIAGRAM PART I..... | 39 |
| FIGURE 29 - EMBEDDED SEQUENCE DIAGRAM PART II..... | 39 |
| FIGURE 30 - EMBEDDED SEQUENCE DIAGRAM PART III..... | 40 |
| FIGURE 31 - CONSUMPTIONRECORD TABLE..... | 41 |
| FIGURE 32 - FIRESTORE DATABASE | 41 |
| FIGURE 33 - ANT DESIGN PROBLEM SOLVING ROAD | 42 |



| | |
|--|----|
| FIGURE 34 - ANT DESIGN SELECTORS EXAMPLES | 43 |
| FIGURE 35 - VICTORY CHARTS EXAMPLE | 44 |
| FIGURE 36 - VICTORY CHARTS EXAMPLES IN THE PROJECT | 44 |
| FIGURE 37 - APP ENGINE SERVICE PRINCIPAL..... | 45 |
| FIGURE 38 - SECRET MANAGER..... | 46 |
| FIGURE 39 - SETTING UP THE MQTT CLIENT | 47 |
| FIGURE 40 - MAIN LOOP | 48 |
| FIGURE 41 - ELECTRICITY CONSUMPTION CALCULATIONS | 48 |
| FIGURE 42 - NETWORK PROFILE SETUP..... | 49 |
| FIGURE 43 - CONNECT TO THE WIFI NETWORK | 49 |
| FIGURE 44 - STARTUP CLASS | 50 |
| FIGURE 45 - DEVICE MODEL CLASS..... | 51 |
| FIGURE 46 - GOOGLE UTILITIES FUNCTION | 51 |
| FIGURE 47 - FIRESTORE GENERIC FETCHER | 52 |
| FIGURE 48 - POSTGRESQL FETCHER..... | 52 |
| FIGURE 49 - DEPENDENCY INJECTION EXAMPLE | 53 |
| FIGURE 50 - SCOPEDSERVICEPROVIDER EXAMPLE | 53 |
| FIGURE 51 - DATA AGGREGATION USING LINQ | 54 |
| FIGURE 52 - ROUTING IN APP COMPONENT | 54 |
| FIGURE 53 - COMPONENT DECLARATION AND REACT HOOKS | 55 |
| FIGURE 54 - FUNCTIONAL OBJECT FIELD DECLARATION | 55 |
| FIGURE 55 - USEQUERY HOOK EXAMPLE | 56 |
| FIGURE 56 - SIMPLE STATE SETTER EXAMPLE..... | 57 |
| FIGURE 57 - FETCHINTERVALSELECTOR COMPONENT USAGE..... | 58 |
| FIGURE 58 - CALLBACK FUNCTION CALL..... | 58 |
| FIGURE 59 - CLOUD BUILD FRONTEND..... | 58 |
| FIGURE 60 - DOKERFILE FOR THE FRONTEND APPLICATION | 59 |
| FIGURE 61 - NGINX CONFIGURATION FILE | 59 |
| FIGURE 62 - BACKEND APPLICATION PIPELINE..... | 60 |
| FIGURE 63 - DATACONTROLLER UNIT TESTIN - SETUP | 62 |
| FIGURE 64 - DATACONTROLLER UNIT TEST I..... | 62 |
| FIGURE 65 - DATACONTROLLER UNIT TEST II..... | 63 |
| FIGURE 66 - RENDERING TESTS - IMPORTING | 64 |
| FIGURE 67 - RENDERING TESTS - ARRANGE | 64 |



| | |
|---|----|
| FIGURE 68 - RENDERING TEST - ACT AND ASSERT | 65 |
| FIGURE 69 - RENDERING TESTS - RESULT | 65 |
| FIGURE 70 - USABILITY TESTING SCENARIO..... | 66 |
| FIGURE 71 - PERFORMANCE MEASUREMENTS..... | 74 |

LIST OF TABLES

| | |
|---|----|
| TABLE 1 - USE CASE DESCRIPTION PART I | 11 |
| TABLE 2 - USE CASE DESCRIPTION - PART II..... | 11 |
| TABLE 3 - USE CASE DESCRIPTION - PART III | 12 |
| TABLE 4 – ACCEPTANCE TEST..... | 17 |
| TABLE 5 - SYSTEM TEST | 18 |
| TABLE 6 - USABILITY TESTING RESULTS | 67 |
| TABLE 7 - USER ACCEPTANCE TEST RESULT..... | 68 |
| TABLE 8 - ACCEPTANCE TESTS GENERATE REPORTS | 69 |
| TABLE 9 - SYSTEM TEST RESULT..... | 70 |
| TABLE 10 - SYSTEM TEST - NOT SUCCESSFUL | 70 |
| TABLE 11 - SYSTEM TESTS RESULTS..... | 71 |

Abstract

The bachelor project is set to offer a comprehensive and detailed method with focus on data drill-down for monitoring, analyzing, activating, visualizing and reacting to data about the electricity consumption of a plugged-in device. The project will be using a self-assembled hardware solution, and develop a system that will govern the entire process. In addition, it will be developed (and deployed) around a cloud infrastructure to allow high-availability and little downtime.

The hardware has at its base an ESP32 microcontroller with WIFI capabilities, the ACS712 electricity sensor and an Arduino. It is meant to collect electricity consumption data from a plugged-in device, aggregate and compute the kilo watt hour consumption and send it to the Google IoT Core.

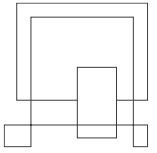
The backend of the system has the responsibility to ingest, transform, and store data in a persistent storage. In addition, it exposes an API that provides ways of extracting aggregated data based on the needs of the caller. Furthermore, it acts as the backend of the system, for managing user profiles, device registration, and other related tasks.

The user interface is provided via a web application, that connects to the backend via the exposed API. It offers the user ability to visualize raw data in near-real time. In addition, it provides means for visualizing aggregated data, making comparisons of different devices and/or time slices, management of the user profile and device registrations.

The system ensures security by following the best practices in a cloud infrastructure and through encryption of the data and the connection between the device and the cloud provider. The system itself is built around the cloud infrastructure and utilizes as much as possible its services.

To successfully achieve the purpose of this project, an agile software development process was governing the development lifetime. It ensured that the progress is logical and has a strong foundation based on the needs of the product owner. Moreover, DevOps practices were utilized to allow continuous integration and development.

The result of this project is a working system, that succeeds in fulfilling the most important requirements set for the system. It is not a production-ready solution, but a dependable proof of concept. To take it to the next level, some suggestions were added to the project in the Project Future chapter.



Introduction

Throughout history, electricity has become one of the pillars of the modern world. A series of fundamental discoveries made by Hans Ørsted, André-Marie Ampère, Michael Faraday and Georg Ohm has paved the way towards electricity becoming an essential part of society. (Institute for Energy Research, 2021)

As the electricity started being produced in larger amounts, and commercialized, the need for measuring its consumption rose. The first meter was patented in 1881 by Thomas Edison. It was an electrolytic meter that used a copper strip which was weighed at the beginning and end of the billing period; and used the gained weight as its basis for consumption calculation.

Fast forward a few decades to the 1960's when the idea of remote monitoring spark, initially using remote pulse transmission and gradually being replaced by various protocols. Nowadays, meters have complex functionality, usually based on the ever-evolving latest electronic technologies. (Smart Energy International, 2006)

In early 2000's the idea of *Smart Homes*¹ started to become more popular and affordable. Powered by the development of *IoT*² and the *need* to control, oversee and automatize elements of a house remotely, more and more parts of a home were innovated. From sound systems to home security to appliances and anything that comes in between. (Assosiation Franco-Chinoise du developpement urbain durable, 2021)

Another important trend that contributed to a higher degree of close monitoring of the electricity consumption is eco-friendliness and the idea of monitoring and controlling how electricity is used more closely. It is done so to reduce unneeded electricity consumption, to try to optimize how the electricity is used and to identify energy *vampire* devices. (Mohammed Ghazal, 2015)

Given the foundation described above, the idea of a *Smart Plug*³ was inevitable. The first attempts to better control the electricity usage per outlet, even if not yet smart per say, were *Plug-in Time Sockets*. These devices were simplistic, more often than not, mechanical devices that allowed to stop electricity consumption without having to be present or unplug the device itself – Figure 1.

¹ Smart Home – a home equipped with lighting, heating, and electronic devices that can be controlled remotely by smartphone or computer.

² Internet of Things – is a network of physical “things”, which are incorporated with sensors, software and other technologies for the purpose of connecting and exchanging data with other devices and systems over the Internet.

³ Smart Plug – a device that allows to control and/or monitor the usage of electricity per electric outlet.

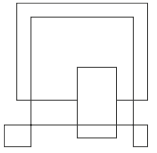


Figure 1 - Mechanical Plug-in Time Socket



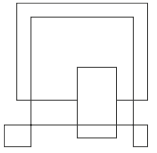
Nowadays, there are various types of truly Smart Plugs that one can choose from, with different functionality and relatively affordable prices. These devices usually have one or more of the below presented capabilities/characteristics: provides remote scheduling, control and IFTTT⁴; real time or/and non-real time remote electricity consumption monitoring, can be used with home assistants (Google Home, Alexa, etc.); suited for indoor or/and outdoor usage; requires or does not a hub/bridge for connection; use WIFI or/and LTE for connection to a network; can be controlled using a smartphone application or/and web-based access; is subscription based vs one time pay; connects directly in the socket or to the electrical panel; have other special functionalities - (ex. Protocols for connecting other proprietary sensors); usually come with certain limitations for maximum load.

A quick look at what is currently on the market - a sum up of the most recommended devices by various reviewers as *PCMag*, *TechRadar*, *CNET* and *Wired UK* - offers an overview of most common features present in the offerings. All the devices were compatible - a device was marked as compatible if it supported at least one assistant - with home assistants like *Alexa* and *Google Assistant*. Other common features were usage of WiFi, IFTTT and Controlling/Scheduling functionality.

Less common, were Power Consumption Monitoring (17%) - even so, many only offered basic functionality like: real-time *on spot*, or only simple reports given that it was accessed via mobile application; possibility of using LTE connection or a web application to monitor data (Appendix D).

Following above, a problem area can be identified, as many tools only focus on the ease of use and mobile application *trend*, making it hard to find a solution that will offer proper monitoring of the consumption and activation of the data. A gap can be identified between industrial/professional devices that are hard to use/install and require deep domain knowledge - meters used by the energy suppliers - and extremely

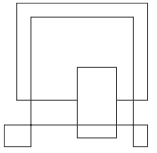
⁴ If This Then That – Is a functionality that offers a user the possibility to define a response to various internal or/and external events.



simplistic devices - previously mentioned *Smart Plugs* - that do not offer meaningful monitoring of electricity consumption for the general population.

The existing general population solutions for monitoring electricity consumption are limited in only offering basic insights about it, due to improper presentation of the collected data that does not capitalize on the use of desktop screen size and user interface capabilities.

This project aims to provide a **comprehensive** and **detailed** method with focus on data **drill-down** for **monitoring, analyzing, activating, visualizing** and **reacting** to data about the **electricity consumption** of a plugged-in device.



Analysis

In this chapter, a deeper and more analytical look at the problem domain will be taken. The purpose of this section is to translate the problem domain into a formal definition that will serve as the foundation for the entire project.

Glosary

At this stage, it is crucial to still stay close to the domain, thus using the same vocabulary, terms and concepts is a must. Next, the essential terminology with associated definitions is presented.

Subscriber - a person with no technical knowledge about electricity consumption, electricity measuring, software development or electrical/electronic domain that is going to utilize the device for personal use in monitoring electricity for home devices. The user is going to receive this service as subscription - thus Subscriber. Your plain boring John Doe;

Device - referred to the self-build electronic device for monitoring, collecting, aggregating and sending electricity consumption using the ESP32-HUZZAH32 and Arduino boards;

System - refers to the overall software system that will govern the entire process of monitoring electricity consumption, storing, transforming, transporting and presenting data as well as interacting with the user;

Consumer - any kind of electrical/electronic device (given appropriate electrical load) that will have its electricity consumption monitored while plugged in;

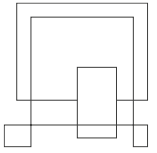
Timer - a time-based task scheduler will be present that will set off various tasks therefore it is crucial to include as an actor that governs the system;

Data Collection (*collected data, collecting data, etc.*) - the action of observing and registering information about electricity consumption;

Update Rate - rate at which the device is sending collected data about the current consumption;

Administrator (*Admin, Sysadmin, etc.*) - person responsible for administering and managing the system;

Google Cloud Platform (GCP) – cloud service provided by Google.



Requirements

After establishing the terminology used in the project, next will be defined the requirements for it. The requirements are grouped into *Functional Requirements* and *Non-functional Requirements* and are closely related with problem domain. The requirements will serve as the building blocks for the foundation of the project.

Functional Requirements

Functional requirements represent what this project will thrive to achieve, from a functional (capabilities) perspective. To ensure a formal definition, they are presented as *User Stories* (Ambler, n.d.) that have a specific format allowing all requirements to adhere to. User Story format:

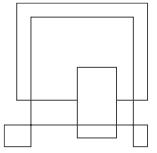
Figure 2 - User Story General Form

As a **user**, I want **feature** so that **user** can **business value/benefit**.

In addition to using the general form, the requirements are prioritized to ensure that the appropriate attention will be paid to more important ones. The prioritization is done using *MoSCoW* Framework (Agile Business Consortium, n.d.) that provides a way of grouping requirements into four defined categories. Moreover, the requirements within the same category are prioritized, by number, where a lower number means a higher priority.

Must: non-negotiable device/system needs that are mandatory

1. As a subscriber, I want to monitor electricity consumption, so that I can see how much electricity a consumer(s) is using.
2. As a subscriber, I want to be able to access monitoring information from any place with internet access, so that I can monitor electricity consumption remotely.
3. As a subscriber, I want to be able to see the current status of the devices, so that I can easily understand which devices are online and which are offline.
4. As a subscriber, I want to be able to visualize historical data, so that I can go back and read the electricity usage at a certain point in the past.
5. As a subscriber, I want to be able to make comparisons of consumptions between devices so that I can see how much a device has consumed over another device.
6. As a subscriber, I want to be able to make comparisons of consumptions between periods so that I can see the difference between two or more periods.
7. As a subscriber, I want to be able to monitor electricity consumption in real-time (near real-time), so that I can observe how much a device is consuming at the moment.



Should: important initiatives that are no vital, but add significant value

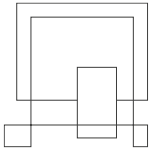
8. As a subscriber, I want to receive reports for certain defined intervals, so that I can have an easy-to-read overview of electricity consumption.
9. As a subscriber, I want to have a profile that I can login to so that I can have an overview of my devices and configurations.
10. As a subscriber, I want my subscriber data, configurations and monitoring information to be stored securely so that other un-authorized subscribers cannot access it.

Could have: nice to have initiatives that will have small impact if left out

11. As a subscriber, I want to adjust the update rate of the device so that I can have more control on how often data is updated.
12. As a subscriber, I want to receive automatically generated reports on a pre-defined timely basis, so I don't have to manually create the reports.
13. As a subscriber, I want to receive updates for the software automatically without me having to interfere with it, so that I do not have to manually update it.
14. As a subscriber, I want the device to be able to stop electricity consumption monitoring without having to physically unplug the device, so that I can remotely control when the device is monitoring - sleep mode.
15. As a subscriber, I want to be able to manually save/favorite sections of time and create a custom tag for them, so that I can easily find it back.
16. As a subscriber, I want to create custom commands (IFTTT) so that I can automate some of the tasks when controlling consumers.
17. As a subscriber, I want to be able to use the device as a remote switch so that I can turn off any consumer without having to physically unplug it.
18. As a subscriber, I want to be able to schedule when a consumer is consuming electricity and when not so that I can create schedules for different consumers.

Will not have: initiative that are not a priority for this specific time frame

19. As a subscriber, I want to access the system on a mobile device (with internet access) through a dedicated mobile/pc application so that I don't have to use the browser for that.
20. As a subscriber, I want to receive assistance/support from the developers of the system, so that they can fix the system in case of malfunction.



Non-functional Requirements

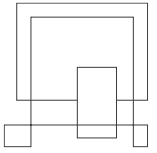
Along with the Functional Requirements, a few non-functional requirements were defined. Their role is to describe system's characteristics rather than capabilities. They do not have a particular prioritization but are rather equally important.

1. The **system** must have a **cloud infrastructure** to ensure **high availability** and **little down time**.
2. The system's embedded part must be developed using the **MicroPython** programming language as the main programming language – other languages can be used as support.
3. The system's non-embedded part development environment must be **.NET Core** using **C#** programming language.
4. The system's front-end must utilize a front-end framework - as opposed to plain HTML, CSS and JavaScript.
5. The system's UI must be easy to use and intuitive for non-technical subscribers.
6. The system must be built in a **modular** fashion to allow future expandability of the system - to other platforms, different types of measuring devices, etc.
7. The system must be **scalable** to provide the possibility of increasing numbers of subscribers without compromising quality of the service.
8. The system must not present a latency higher than five seconds – a certain task can take more than that depending on whether it is aggregating data, and the amount of data but system latency should not exceed five seconds;

Delimitations

To understand and limit the scope of this project, a set of delimitations are needed in place:

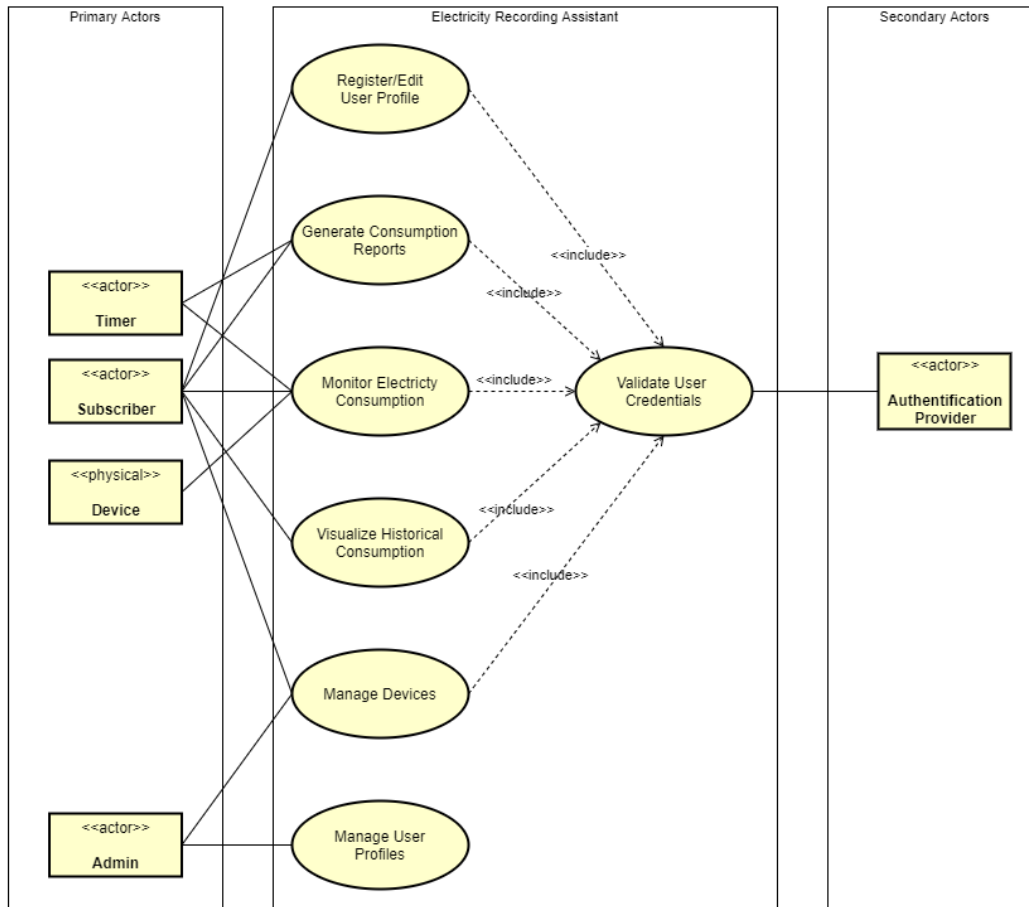
- The project will not consider mobile applications (either Android or iOS) as part of its scope;
- The project will not focus on exploring the possibility for integration with *Home Assistants*;



Use Case Diagram

Use cases present different overall functionalities of the system derived from the Functional Requirements. It creates a new level of abstraction where the requirements are categorized and grouped to form a use case that an actor can perform actions to.

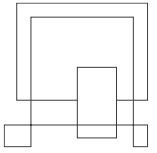
Figure 3 - Use Case Diagram



In addition, it also presents how different use cases interact or are connected with each other, providing a more formal overview of the requirements. The Use Case diagram can be found in Appendix E.

Actors

The use case includes two main categories of entities: actors and use cases. While use cases represent an overall functionality of the system, and actor is an entity outside the system that interacts with it in a certain way. Further actors can be split into two distinct groups: primary actors – that initiate a use case; thus, they can be described as independent – and secondary actors – that do not interact with a use case by themselves but are rather used by the system to achieve a certain goal.



Primary Actor

- Subscriber – as defined in the *Glosary* it is the end-user of the system. The subscriber will initiate various use cases that will provide him with certain benefits.
- Timer – a timer even if it is rather a concept than an actual entity, it plays an important role within the system. More precisely it takes care of *automation* to some degree, in collecting data about the electricity consumption and in creating periodical reports (daily, monthly, etc.). A timer will interact directly with a use case, making sure that certain tasks are performed at exact rate.
- Device – as defined in the *Glosary* a Device is the physical device that interacts with the system. The device has the primary role of collecting data and forwarding it to the system.
- Admin – as defined in the *Glosary* the Admin is the technician that operates and maintains the system, it has certain powers in managing other users' profiles as well as managing all the devices. Its primary role is to offer support to Subscriber through certain use cases as well as make sure the proper operation of the system.

Secondary Actor

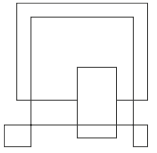
- Authentication Provider – since the system will be connected and directly interacting with the Internet, it is a priority to have in place a proper authentication mechanism. As the importance of cyber security grew over the years, it became a separate entity, that it not part of the system per say. It provides a functionality that the system is using, mainly in validating users' credentials.

Use Case Definitions and Descriptions

To further narrow down what each use case represents some definitions of the use cases are needed. Furthermore, Use Case Descriptions were created that provide a more in-depth look at each use case, and how an actor will interact with it.

Use Case Definitions

- *Register/Edit User Profile* – this use case includes all the functionalities in creating and editing a user profile. Without a user profile a Subscriber cannot interact with the system. This use case includes *Validate User Credentials* – when editing, not when creating – as only a specific Subscriber should be able to access his/her own profile;
- *Generate Consumption Report* – this use case encompasses the capabilities of creating consumption reports for specific periods of time from pre-existing data and making them available to the Subscriber. It describes the functionality of a Subscriber defining and



requesting a consumption report. In addition, the Timer interacts with this use case, when automatic and predefined reports are created;

- *Monitor Electricity Consumption* – this is the core use case of the system, and it represents the action of observing electricity consumption of a Consumer through the system. The Subscriber uses it in viewing the electricity consumption, the Timer in triggering the system into observing electricity consumption and the Device into being instructed to collect data about electricity consumption;
- *Visualize Historical Consumption* – represents the ability to observe electricity consumption at some point in the past (given existing data);
- *Manage Devices* – includes the functionalities of adding, removing, resetting, disabling or enabling a device. Subscribers have access to perform some actions only on devices registered in their user profile, while the Admin has access to perform the actions over all devices.
- *Manage User Profiles* – represents the functionality of editing, resetting or removing any user profile, and is only initiated by the Admin;
- *Validate User Credentials* – is the action of verifying and mapping user credentials to a user profile using the *Authentication Provider*;

Use Case Descriptions

Use Case description, provide a step-by-step description of how an actor interacts with a use case. It allows for a more detailed overview of each use cases as well as a foundation for future testing. To ensure comparability and offer formality to the process each use case description necessary contain a set of items:

- Name – name of the use case;
- Summary – a short description of what it does;
- Actor – the actor that performs the steps;
- Precondition – some conditions that must be met before initiating the use case;
- Base Flow – actions and system counter actions under normal conditions;
- Alternative Flow – contingency steps in cases of failures or malfunction;
- Comments – additional notes to be considered;

Next tables presented the use case diagram for the *Manage Device* use case – it is split in multiple sections out of reading convenience. The rest of the use case descriptions can be found in Appendix F.

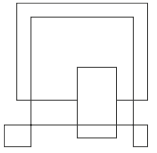


Table 1 - Use Case Description Part I

| Item | Value |
|--------------|--|
| Use Case | Manage Device |
| Summary | All the functionalities in controlling and managing the electricity consumption monitoring device. |
| Actor | Subscriber |
| Precondition | The device must be powered and connected to the internet. The subscriber must have access to a computer with a browser installed that is connected to the internet. The subscriber must have a user profile and the device must be registered with the subscriber. The subscriber must be logged in. |

Here are included the name of the use case, a short summary, the actor that will initiate the use case and the precondition. It is important to note that not having fulfilled the preconditions it is very likely that the system will not react as described.

Table 2 - Use Case Description - Part II

| | Actor | System |
|-----------|--------------------------------|---|
| Base Flow | 1. Access Main Control Panel | 2. Display Main Control Panel |
| | 3. Access Devices Panel | 4. Display Devices Panel |
| | 5. Select Device | 6. Display actions to perform |
| | 7. Select an action to perform | 8. Display configurations that can be added by Subscriber |
| | 9. Add configurations | 10. Validate and apply configurations |
| | | 11. Notify Subscriber that the action has been successfully completed |

The Base Flow consists of the steps performed by the actor and the reaction of the system to actor's actions. It is to be noted, that in some case (as step 10 and 11) the system or the actor may perform more than one step/action. The Base Flow describes the *Sunny Scenario*⁵.

⁵ Also referred as *Happy Path* or *Happy Flow* and describes the default scenario featuring no exceptional or error conditions.

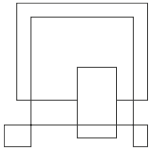
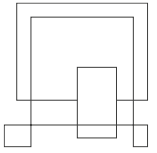


Table 3 - Use Case Description - Part III

| | |
|------------------|--|
| Alternative Flow | <ol style="list-style-type: none"> 1. Base Flow 1 - Access Main Control Panel <ol style="list-style-type: none"> a. System does not display Main Control Panel or displays an error b. Make sure that the computer has an internet connection and uses a compatible browser c. Base Flow 1 2. Base Flow 1 - Access Main Control Panel <ol style="list-style-type: none"> a. System displays Main Control Panel but there are no available devices b. Make sure that Device has been registered with the system c. Base Flow 1 3. Base Flow 5 - Select Managing Device option <ol style="list-style-type: none"> a. System notifies that Device is offline b. Make sure that Device is plugged in and has access to internet c. Base Flow 1 4. Base Flow 9 - Adds various configurations and apply them <ol style="list-style-type: none"> a. Subscriber enters invalid configurations b. Notify subscriber that the configurations are invalid and the task could not be performed c. Base Flow 9 5. Base Flow 10 - Apply the provided configurations <ol style="list-style-type: none"> a. An error occurs making it impossible to apply the configurations b. Notify subscriber that the action could not be performed c. Base Flow 9 |
| Comments | <p>The process can be cancelled at any point in Base Flow 1 - 10 - before the system has performed the action. Compatible browsers:</p> <ul style="list-style-type: none"> ● Chrome Version 49 or newer |

The last part describes the Alternative Flow. It is concerned with malfunction or error conditions in which the system does not respond in the expected way. It references the step when the malfunction occurs, as well as describes the expected malfunction and the contingency steps.

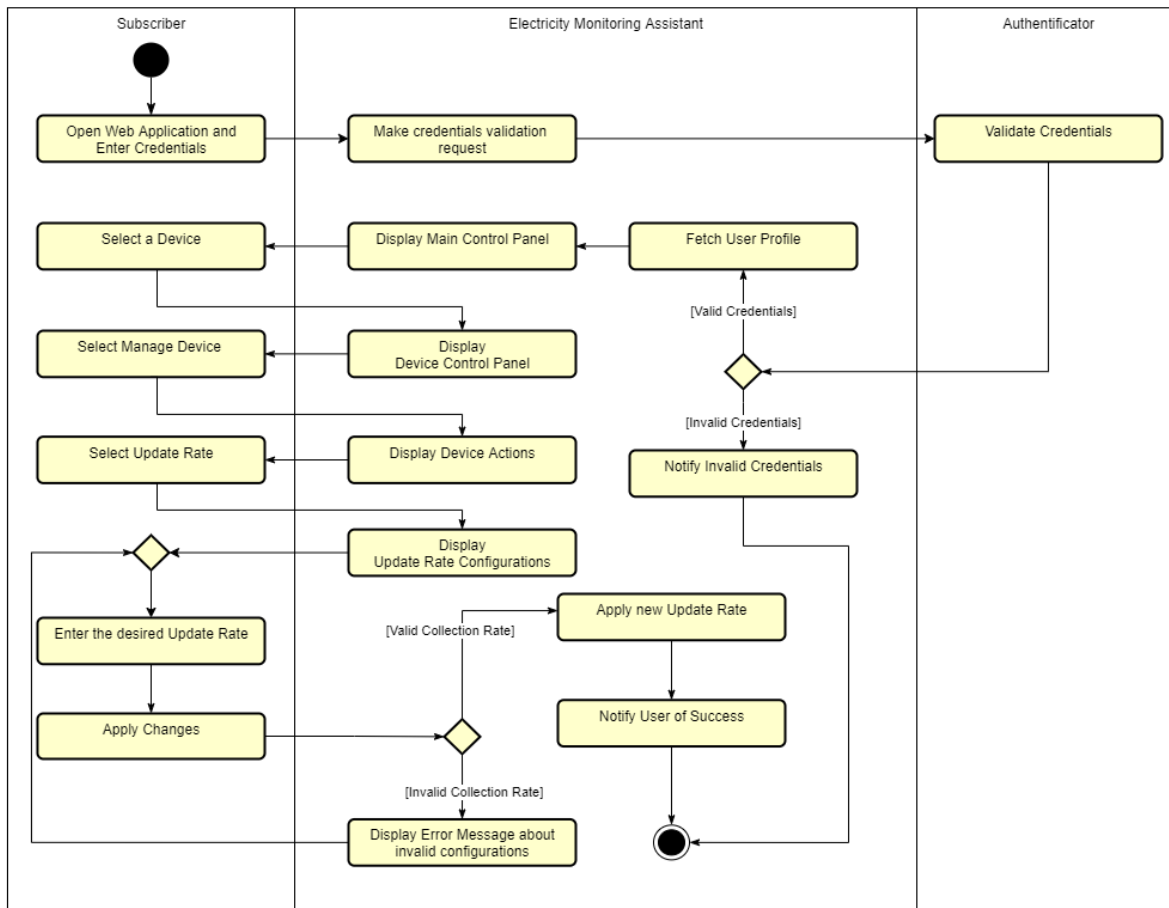
In addition, the Comments section is included that enriches the use case description with some additional notes, in regards to cancelling conditions and necessary compatible browsers.



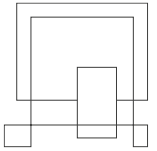
Activity Diagrams

Complementary to Use Case Descriptions, Activity Diagrams provide a visual overview of the flow for a use case. It incorporates both the basic flow and the alternative one and describes in the same manner an actor's actions and system's reactions to it. The activity diagram helps to better observe the flow of actions/re-actions and how the actor interacts with the system as well as on what conditions different deviations occur. The complete set of activity diagrams can be found in Appendix G.

Figure 4 - Activity Diagram - Manage Device



The diagram above describes the Manage Device use case. It can be seen that it includes the actor, the system and the secondary actor – Authenticator. The start and end of the diagram are denoted by the full and subsequently the underlined circle. The use case is initiated by the Subscriber, when opening the web application and entering his/her credentials. Given the credentials the system will try to validate user credentials using the Authenticator.



Valid credentials will grant user access to the next steps while invalid once will notify the user about it and finish the flow. Next the user will access the device control panel and will attempt to update the collection update rate configuration.

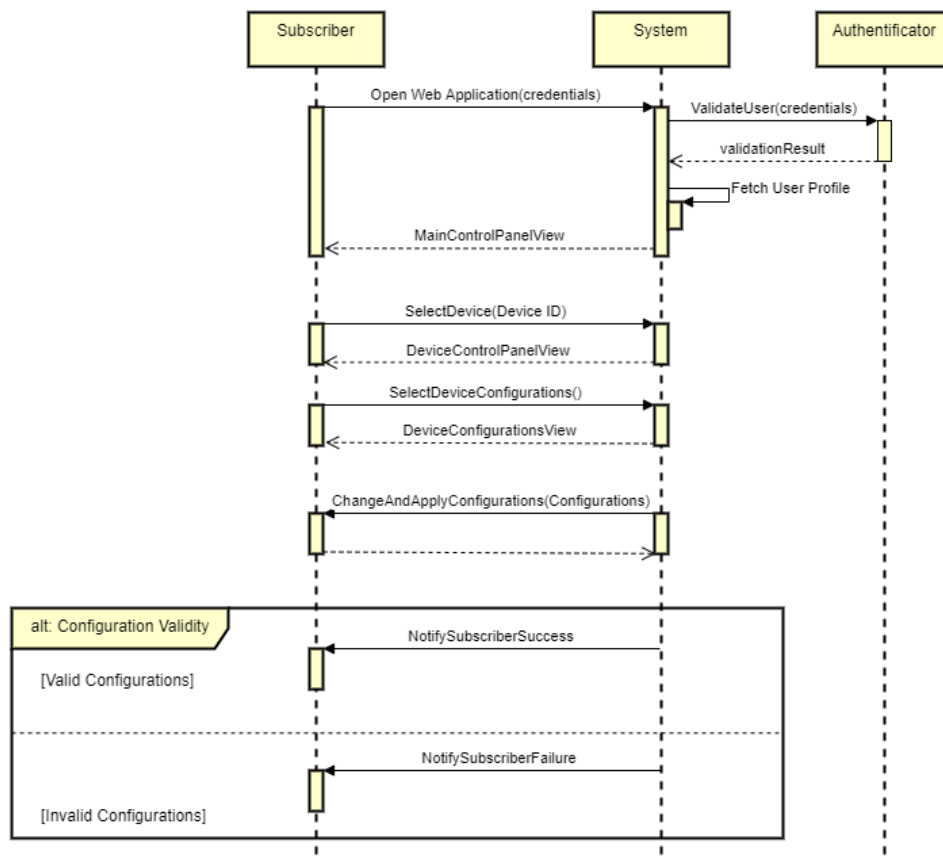
In the same manner, if the user entered invalid configurations, then the system will notify and reverse back the action, otherwise it will apply the new configurations and notify the user of success.

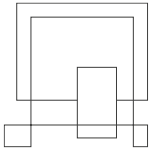
This activity diagram, offers the basis for design and further implementation as it describes, almost in pseudo code how the system should act.

System Sequence Diagram

The system sequence diagram treats the system as a black box, and describes the sequences as well as the data flow that occur between the actor and the system. Worthy to mention is the fact that even if it may seem redundant to define system sequence diagrams for the same use cases as the activity diagrams, many times it can provide a different perspective while unveiling different details missing in the activity diagram. It is crucial to mention that the process of defining the diagrams offers more value than the diagrams themselves. (Larman, 2005)

Figure 5 - System Sequence Diagram - Manage Device – Sunny Scenario

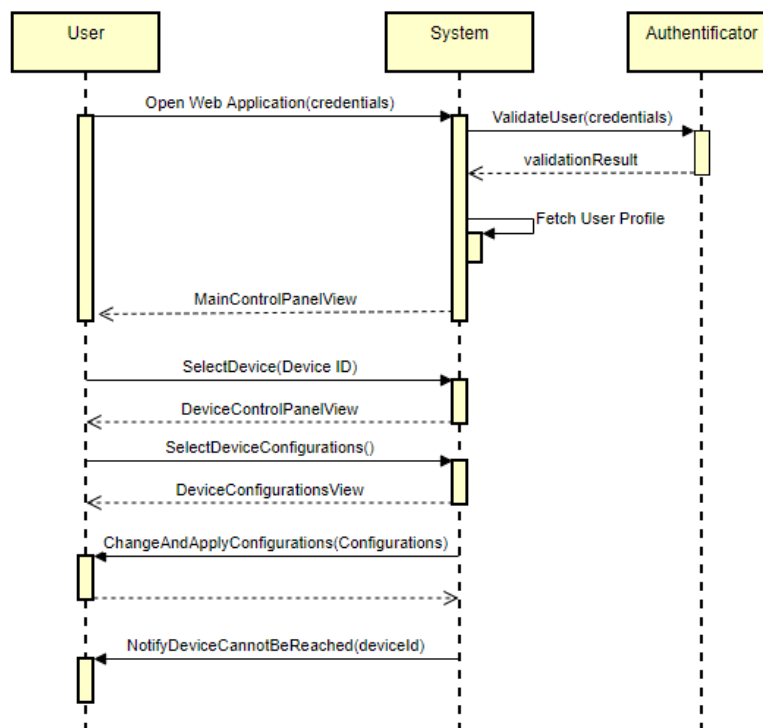




As mentioned before, the system sequence diagram, offers a good overview of the data flow between the actors and the system. In the sequence one, can be seen that the Subscriber passes the credentials to the system that itself passes the credentials to the authenticator for validation. After that the system uses the response from the authenticator to continue the flow and fetch a user profile's data.

As the last point, the sunny scenarios can be accompanied by failure scenarios, that indicates what a system should do in cases of failures or malfunction.

Figure 6 - System Sequence Diagram - Manage Device - Failure Scenario



In this case, when the system cannot perform the task because the device suddenly disconnected, the system should inform the Subscriber about inability to perform the task. Moreover, the system will attach the device id, to the message so that the Subscriber knows which device malfunctioned. The rest of the system sequence diagram can be found in Appendix H.

Domain Model

The domain model is a visual representation of the problem domain. It encompasses the entities of the problem domain as well as the relationships between them. Using the knowledge gathered through the analysis process about the problem area, a domain model for the project was created. The domain model can be found in Appendix I.

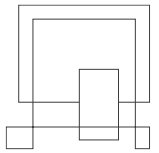
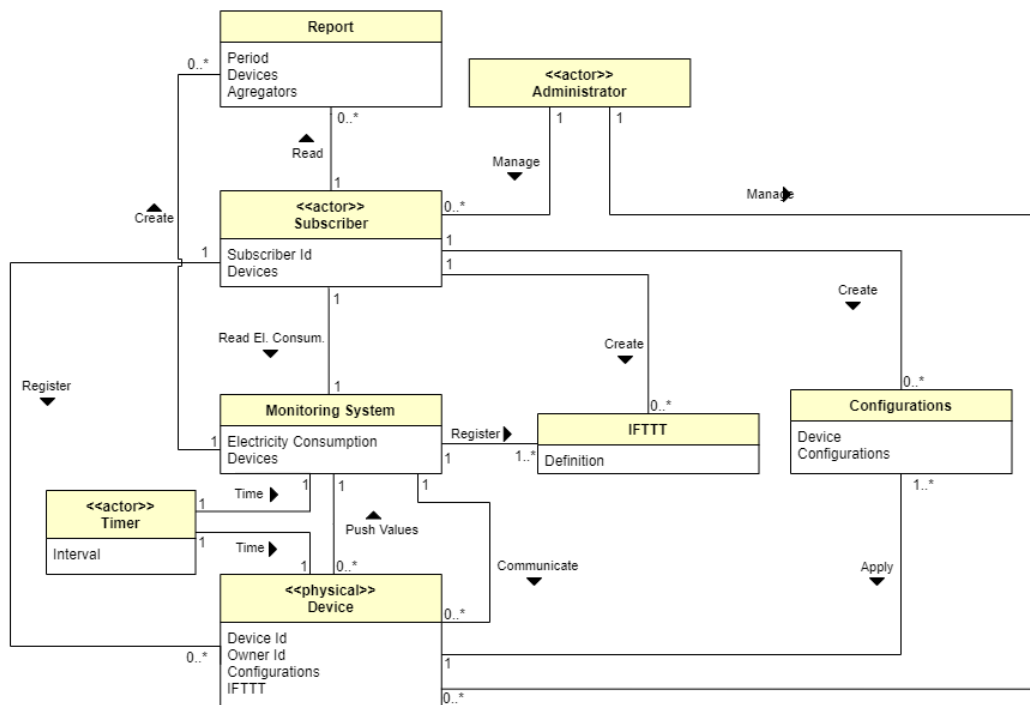


Figure 7 - Domain Model



Domain Model Description

The domain model is made of eight entities and the relationships between them. The *Timer* coordinates both the *Device* in collecting and forwarding data and the *Monitoring System* in generating timely reports. Both entities have a one-to-one relationship with the *Timer* as each entity can have only one *Timer* and vice-versa.

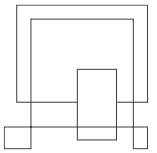
Opposingly, the relationship between the *Monitoring System* and the *Device* is one-to-many as a system can monitor multiple devices, but the device must connect to a single system. The domain model will further serve as the basis for the class and er diagrams.

Test Cases

Even if quite early in the development of the project, at this point already it is possible to define test. This will allow in the future steps to be able to test, how the system performs compared to what was defined in this chapter.

Even if this project has an Iterative Model for its SDLC⁶ (divyanshu_gupta1, 2021) when it comes to testing, some aspects of the V-Model (Dharmendra_Kumar, 2019) will be borrowed. Because of that

⁶ Software Development Life Cycle



it is already possible to define *User Acceptance Tests* (pp_pankaj, Geeksforgeeks, 2019) and the *System Tests* (pp_pankaj, Geeksforgeeks, 2019). The Acceptance Tests and System Tests are testing the same aspects but from different perspectives. Even so, the tests were formulated in slightly different ways – but maintaining the principle of testing the same aspects – to tailor the tests for the testers.

User Acceptance Testing

User Acceptance Tests are used to identify if the final system is working as expected for the end-user – Subscriber in the case of this project. To identify that, the tests were formulated using Use Cases of the system. All the User Acceptance Tests can be found in Appendix J.

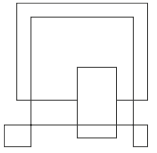
Table 4 – Acceptance Test

| ID | Details | Results | | Assessment |
|----|-------------------------------|--------------------------------------|--------|------------|
| | | Expected | Actual | |
| 1. | Access Main Control Panel | Display Main Control Panel | | |
| 2. | Access Electricity Monitoring | Display Electricity Monitoring | | |
| 3. | Select devices | Display data for the selected device | | |
| 4. | Change refresh rate | Display data for the selected device | | |
| 5. | Change interval | Data updates accordingly | | |

As can be seen, the tests consist of three main dimensions

- Details – that describes what the user is expected to do;
- Results:
 - a. Expected – that describes how the system is expected to react
 - b. Actual – the actual result of the test
- Assessment – assessment whether the test is passed or not

In the case of the *Monitor Electricity Consumption* test, the user is performing certain steps to achieve a final goal – of visualizing electricity consumption in near-real time. The acceptance tests will allow to determine whether the system is fulfilling user needs not only functionally but also in the same way that the user would expect.



System Testing

The System Testing will be made of the same test cases defined for the *User Acceptance Testing* with the difference of being performed by the developers. It will not only allow to identify where the system does not comply with the defined testing suits, but additionally, since it will be performed by the developers, will provide insights on what exactly causes the error.

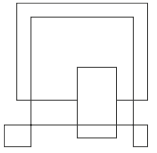
Additionally, the System Tests are usually performed on the defined requirements (functional, non-functional or both) and are focusing on testing the entire system as a whole. The tests focus on providing feedback about the quality of the system and test both the design of the system as well as the expectations of the product owner. To ensure a better coverage of the system using the tests, additional test cases were defined to be performed by the developers. All the additional System Tests can be found in Appendix K.

Table 5 - System Test

| M <i>As a subscriber, I want to monitor electricity consumption, so that I can see how much electricity a consumer(s) is using.</i> | | |
|---|--|------------|
| ID | System Test | Assessment |
| 1. | The device must be able to measure the electricity consumption | |
| 2. | The device must be able to collect and format data so it is easy to transmit | |
| 3. | The device must be able to transmit data to a cloud system at predetermined intervals (collection rate) | |
| 4. | The system must be able to observe and collect data transmitted by the device | |
| 5. | The system must be able to store, transform, aggregate and present data to the subscriber | |

The test cases, are defined for each requirement and are set as a collection of sub-requirements against which the system is tested. This allows provides two benefits:

- Allows to test how much a specific requirement is fulfilled;
- Tests the overall quality of the system, while treating the system as a black box;



Design

In this chapter the design of the solution will be established. The chapter will explore various design and technology choices, describe the overall architecture of the solution as well as architecture of small sub-parts. The chapter is the final step before starting the implementation of the application and should provide a clear blueprint.

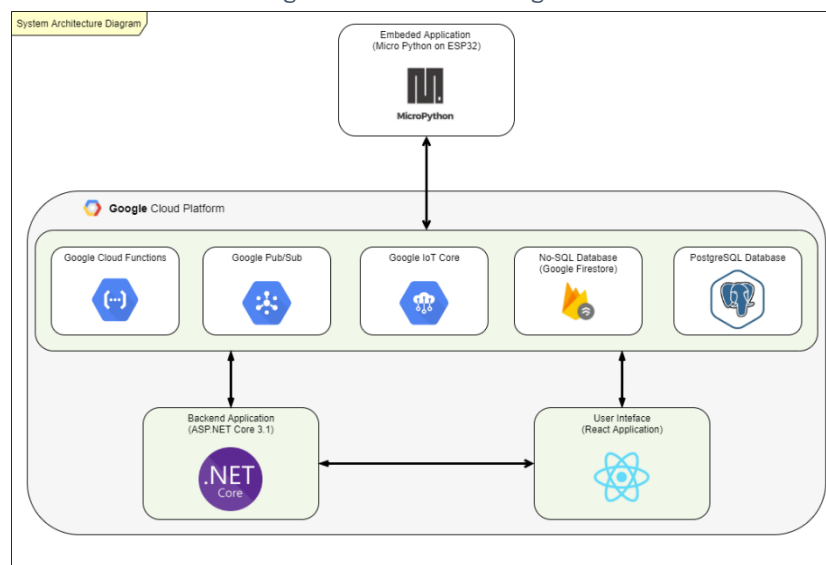
Architecture

Next, the architecture of the solution (both high-level and more detailed) will be presented. In addition, a look on the data flow will be taken, closing down with a description of the DevOps setup (Atlassian, n.d.).

High-Level Architecture

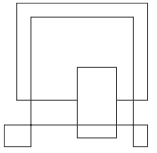
The high-level architecture presents the main parts of the solution. It is made for a quick overview of the solution's components and their relationships. The diagram can be found in Appendix L.

Figure 8 - Architecture Diagram



As can be seen the solution consists of 4 main modules:

- Embedded Application – that runs on the *ESP32* board and connects to Google Cloud Platform. The main role of the application is to collect, format and send data to the cloud;
- Google Cloud Service – various services used for the application including storage – will be described in more detail in the chapter *Cloud Provider and Cloud Services*;

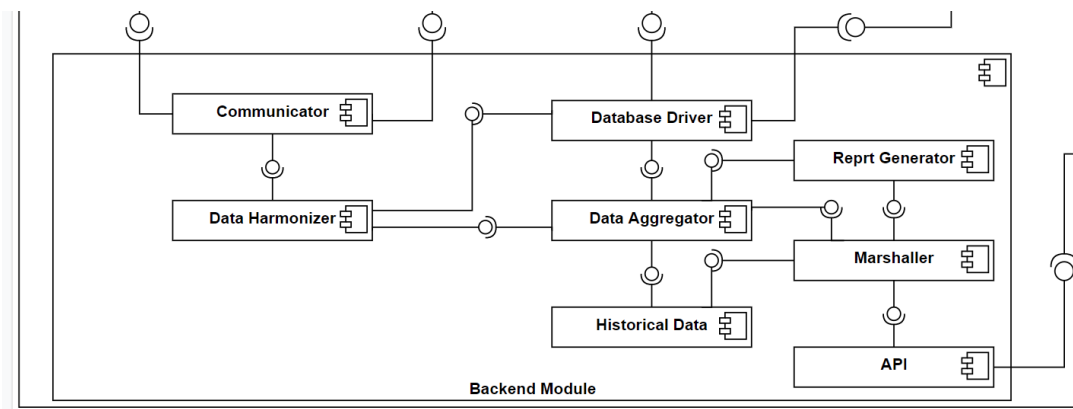


- Backend Application – acts as a bridge between the user and raw data. Its roles consist in aggregating, storing, retrieving and activating data as well as a data flow coordinator.
- Frontend Application – will be the presentation layer for the user, as well as the gateway to the solution.

Component Architecture

Component architecture presents different components of each layer/application within the solution and how they interact with each other. It is another *layer* of detail, that breaks down the applications/modules into smaller components based on their functionality. Only a part of the diagram will be presented out of visual convenience, the original diagram can be found in Appendix M.

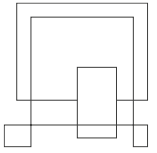
Figure 9 - Component Architecture Diagram



In the diagram is presented the *Backend Module* and the components that it is made of. As mentioned before the components are based on their functionality within the system. Some of the components will be further discussed:

- Communicator – component responsible for communicating (in/out) with GCP;
- Database Driver – component responsible for interacting with the databases (both SQL and No-SQL);
- Data Harmonizer – component responsible for normalizing data that passes through the module;

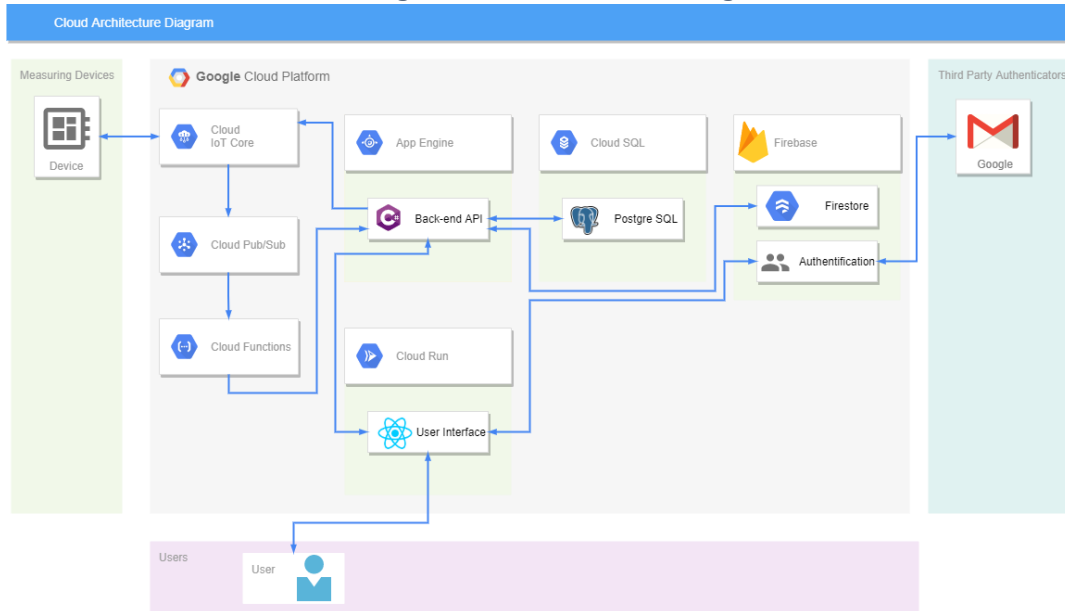
The component diagram offers a logical grouping of different tasks that a module should do. It will provide a great overview of functions that will need to be implemented.



Cloud Architecture

Cloud Architecture describes the solution's architecture in the context of GCP. It presents how different modules work together as well as what services are used to achieve that. The Cloud Architecture diagram can be found in Appendix L.

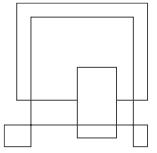
Figure 10 - Cloud Architecture Diagram



The main part of the diagram is the grey area labeled *Google Cloud Platform*. It Provides an overview of what services are used to run the applications, as well as what services are used to facilitate the data flow.

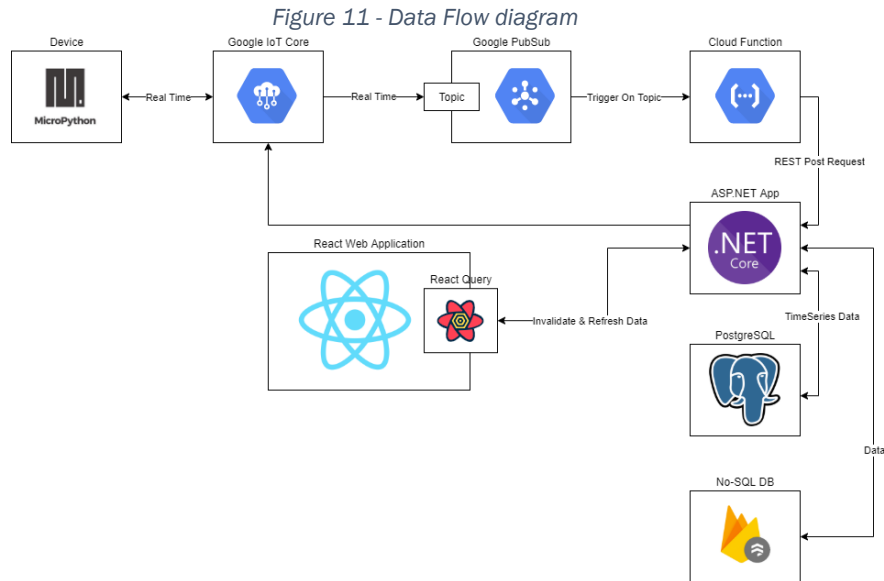
Next will be presented two main services that act as hosting for the applications. The rest of the services will be described in detail in the *Cloud Provider and Cloud Services* chapter.

- *App Engine* – is a fully managed, scalable and serverless deployment slot for running applications. It has native integration for Node, Java, C# and other runtimes as well as possibility to deploy Docker containers with custom runtimes. In the system it serves as the deployment slot for the Backend Application. (Google Cloud Platform, n.d.)
- *Cloud Run* – is a highly scalable and serverless platform for running containerized applications. In the system it serves as the deployment slot for the Frontend Application. (Google Cloud Platform, n.d.)



Data Flow

It is essential to understand the data flow within the system for a better overview of how different parts of the system work together. The Data Flow diagram, presents how the data travers main components of the system. The diagram can be found in Appendix N.



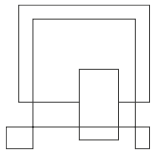
From the diagram it can be seen that data after being collected by the embedded application is transmitted to the IoT Core and exposed through the Pub/Sub service. From there the Cloud Function has a trigger on the specific Topic with the Pub/Sub and every time there is a new message it issues a post request with the message data.

The Cloud Function is used in between the backend application and Pub/Sub service to allow decoupling of services and modularization of the architecture.

The backend application receives the request, aggregates and normalizes the message data and inserts it into the correct table with the proper identification. Then the frontend application is constantly checking for new data and displaying to the user.

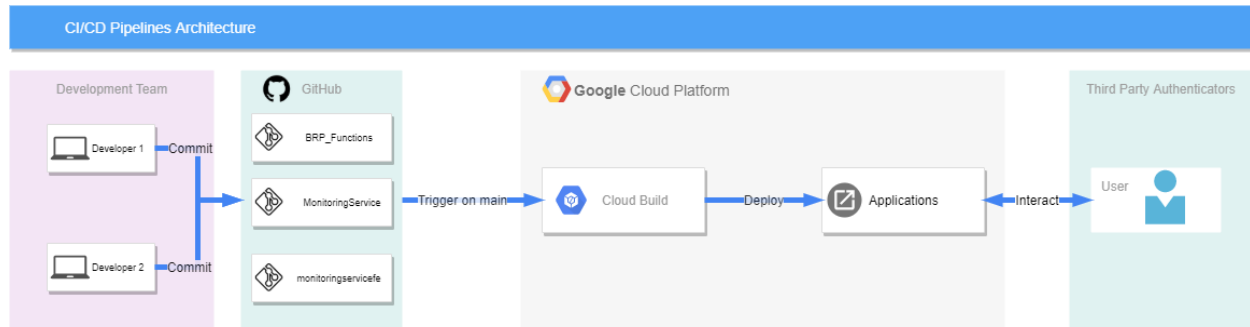
DevOps Architecture

Even if it is not a part of the system per say, continuous integration and development is a core part of the system development and nowadays, has become a close part to the process. It allows for the development teams to be agile and offers control and convenience over the deployment strategy.



To ensure a well-functioning system with little downtime following DevOps practices is a priority. Considering that, it was decided to develop and establish CI/CD Pipelines⁷ with the following architecture. The diagram can be found in Appendix O.

Figure 12 - CI/CD Pipeline



This setup allows for easy deployment of the services to the cloud. When new code is development and committed to the repositories the *Cloud Build* service has triggers and starts the build process. It utilizes various *build tools* and follows the *steps* defined in the *cloudbuild.yaml* files for each application. After the applications are build, they are deployed, started and ready for usage.

In this way, the build and deploy process is automated and requires little to no energy from the developers, at the same time allowing to be consistent and agile.

Technologies

In this chapter, a look at the different technologies that will be used throughout the project will be taken. It is by no means a full *list* of all the technologies, but rather a selection of more important one that are relevant for this project. In addition, a presentation of the cloud provider and services used in the system will be made.

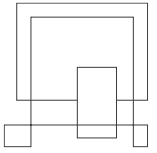
Tech Stack

The technologies used in a project can greatly influence its performance, scalability and modularity. After careful consideration a list of base technologies was computed that will provide a good foundation for the project's implementation as well as some degree of comfort for the developers.

Backend

- *ASP.NET* – starting from the non-functional requirements, it was stated that .NET Core platform must be used for the backend part of the system. *ASP.NET* is an extension of the .NET Core

⁷ Continuous Integration and Continuous Development



platform with tools and libraries tailored for building web applications. It is fast and scalable, and performed greatly in the *TechEmpower* benchmarks. (Tech Empower, n.d.)

- *BetterHostedServices* – is an open-source micro library developed by user *GeeWee* (*github*) that adds another layer over .NET's *HostedServices* that solves some of often met issues with the services. A more in detail description is presented in his blog (Wengel, n.d.).
- *RepoDB* – is an open-source project initiated by *Michael Camara Pendon* that fills the gaps of micro-ORMs⁸ and macro-ORMs by providing a simplified at the same time fast and powerful way of working with databases in the .NET world without too much overhead (RepoDB, n.d.).

Frontend

- *React* – as per non-functional requirements of using a frontend framework, *React* was the choice that this project settled on. *React* is a powerful framework that offers component-based development (modular) of interactive UIs developed using *JavaScript* (*reactjs*, n.d.).
- *Ant Design for React* – is a fronted library that branches as from *Ant Design* with specific implementation for *React*. It offers a set of design principles and pre-build components in designing enterprise-class user interfaces for web applications (Ant Design, n.d.).
- *React Query* – is a *React* library that offers performant and powerful data synchronization. It is declarative, automatic, powerful and configurable at the same time simple and familiar (Tanstack, n.d.).
- *Axios* – is a *promise-based* client for Node.js that allows creating http requests for web applications (Axios, n.d.).
- *Lodash* – is a *JavaScript* library that offers utility functions with focus on modularity and performance (*lodash*, n.d.)

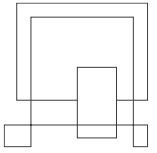
Database

- *PostgreSQL* – is an open-source database system with focus on reliability, robustness and performance (*postgresql*, n.d.).
- *Firestore* – is a cloud No-SQL database service offered by *Firebase* (part of Google Cloud Platform) with focus on scalability and flexibility (Firebase, n.d.).

Embedded

- *MicroPython* – as per non-functional requirements, the project must utilize *MicroPython* in its embedded solution. *MicroPython* is an implementation of *Python3* language with focus on efficiency that includes a subset of Python's standard libraries optimized of microcontrollers

⁸ ORM – Object-relational mapping



(MicroPython, n.d.). The language allows to abstract out the hardware as well as a fast-growing open-source community around it.

- *Arduino* – as a support language when utilizing the *Arduino* board.

Hardware

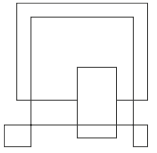
The hardware that has been built for this system is based on the **ESP32HUZZAH** microcontroller for the processing and wireless connectivity and the ACS712 for the electricity measurements. The hardware prototype has been built on a Veroboard using breakout modules for the software development. The prototype should be remade on a circuit board with more integrated modules and a proper AC safe casing once the hardware is verified as correctly working.

- **ACS712** - The ACS712 has been used in the form of a module board to speed up design time and reduce cost for the current small batch prototyping. It provides a solution for AC or DC current sensing using a linear hall sensor circuit on the surface of the die. It provides us an easy interface to read out the sensor through a 1-line variable voltage line.
- **ESP32-HUZZAH** - This is a ESP32 microcontroller module board using the **WROOM32** module. This module provides us with a microcontroller with integrated Bluetooth and Wi-Fi capabilities and a stable platform for prototyping. A reason for using this module is because the ISO emission standards are already in place and it takes out the need to have this tested and documented separately. In addition, it also has the ability to be set up to function on a battery if we find the need for this.
- **4 Amp 220V AC glass fuse** - There is a fuse implemented in the circuit to protect the hardware from damage because the project involves working with live AC current.
- **AC-DC Converter** - In order to simplify the construction of the hardware we opted to use an off the shelf AC-DC converted that we can plug in. For future development and actual production this would be integrated into the board itself with custom made hardware.
- *Arduino* – is used as the interface for collecting data from the **ACS712** and passing data to the ESP32 device.

Cloud Provider and Cloud Services

The choice of the cloud provider was made after careful deliberation and was settled on using *Google Cloud Platform* as project's cloud provider. The argument for this choice included:

- Developers past experience with the provider;
- Reliable, powerful and rich in services that fits project's needs;
- Low-costs with rich free quota offers;
- Pre-existing credits and possibility to obtain credits for paid services;



Choice of services, mainly implied identifying what services will fit the needs for the project. The services used in the project are (with exception of the two services already mentioned in *Cloud Architecture* chapter):

- *Cloud IoT Core* – is a service that provides capabilities to securely connect, manage and ingest data from embedded devices as well as two-way communication. It acts as the cloud gateway through which data comes in. The service uses MQTT⁹ standard for communication. (Google Cloud Platform, n.d.)
- *Cloud Pub/Sub* – Cloud IoT Core works together with Pub/Sub service to allow receiving messages ingested by it. Pub/Sub is a publisher – subscriber communication service that allows asynchronous communication with low latency. (Google Cloud Platform, n.d.)
- *Cloud Functions* – is a serverless and scalable service to run small applications (functions) on a pay-as-you-go plan. The functions allow for small functionalities to be run with little overhead. In the system their role is to observe new messages in the Pub/Sub service and create post requests to the Backend Application. (Google Cloud Platform, n.d.)
- *Cloud SQL* – is a cloud database service with support for most common technologies as: *MySQL*, *PostgreSQL* and *SQL Server* (Google Cloud Platform, n.d.).
- *Cloud Build* – is a serverless CI/CD platform that allows to build, test and deploy application. Among other features it has native Docker support – which allows for easy selection of different desired build technologies that are not directly available as part of Google's build library - and an extremely generous free tier (Google Cloud Platform, n.d.).
- *Container Registry* – is a fully-managed Docker container image storage. It is closely integrated with Google's CI/CD platform which allows for easy management of the build images (Google Cloud Platform, n.d.).

Design Patterns

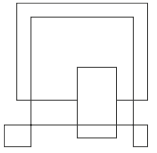
In this chapter a few design patterns that will be used in the project will be presented, along with an outline of the coding principles and practices.

Dependency Injection

Dependency injection is a technique that allows for a class to abstract from its dependencies by decoupling an object's creation from its usage. Usually, the dependency injection pattern involves 4 elements:

- *Service* – a class that will be used in other classes (i.e., the dependency);

⁹ MQTT is a machine-to-machine lightweight communication standard



- Client – an object/class that will be using the Service (i.e. the dependent);
- Interface – it is services interface that is used by the Client;
- Injector – a manager of the process, it creates and locates services;

In *.NET Core* world a lot of this is already taken care by the platform. The injector's responsibility is provided by the *IServiceProvider* interface, which is a build-in service container that allows registering service and injecting them in dependent classes.

Hand in hand with dependency injection, there's a topic that should be addressed as well, is service's types and their lifetimes when registering dependencies with the injector. *.NET Core* defines 3 types of service with their respective lifetimes:

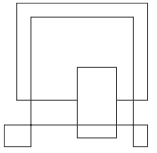
- Transient – new instances are created every time the service is requested, and disposed when the request is finished;
- Scoped – a new instance is created for each defined scope, but the same instance is returned upon requests within the same scope;
- Singleton – a single instance is created and returned every time the service is requested;

A special case of the Singleton type is the *IHostedService* which has the same lifetime as the singleton (single instance throughout the existence of the application). The difference is that the hosted services have the necessary implementation for making them long-running background services/workers (Larkin, Smith, Addie, & Dahler, n.d.).

Options Pattern in ASP.NET Core

Options Pattern allows for accessing groups of settings through a defined class type. In this way the options/settings for an application are set during the initialization of the application, and can be easily accessing through the life of the application via decency injection of the *IOption<TOptions>* type.

ASP.NET Core provides various *out of the box* solutions for setting application settings from different sources: *AppConfig file*, *Azure Key Vault*, *Azure Application Configurations*, etc. At the same time, it allows for definition of custom sources for the configurations. This allowing for the system to utilize already existing Google Cloud service for storing applications settings called *Google Secret Manager*. (Larkin & Anderson, Options pattern in ASP.NET Core, 2021)

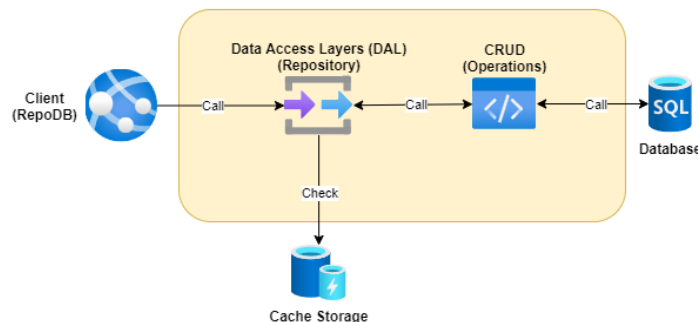


Repository Pattern

The repository patterns define a way for centralizing and/or normalizing the access to data sources in an application. It encapsulates the logic required for accessing data by creating a layer of abstraction through an interface that defines the needed functionalities for different data sources.

In the project's system, the repository pattern will be indirectly used through the *repoDB* library and directly implemented for other data source (*Firestore*). It will be done to provide a unified way of working with the two data sources and have an understanding of the system's models between the

Figure 13 - *repoDB* repository pattern



two storages. An example of how *repoDB* is defining and utilizing the repository pattern can be observed through the diagram bellow (*repoDB*, n.d.):

Class Diagram

The class diagram of the system serves as a set of exact descriptions of how the system looks and how its internals are connected. The system covers three main diagrams for the Frontend *React* Application, Backend *dotnet* Application and the Embedded *MicroPython* Application.

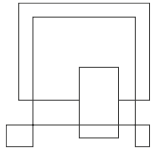
Frontend Application Class Diagram

The front-end application is build using *React* framework. When developing *React* Applications there are usually two main paths that are taken: functional or class components¹⁰ both of them having their pros and cons.

In this project, the preferred way was developing the application using functional component. The main reasons for that are:

- Easier to read and maintain – because each component is a plain JavaScript function;
- Generally, less code and boilerplate;
- Encourages better separation of concerns;

¹⁰ Even if it is completely acceptable to mix those, usually it is preferred to stick with one in order to maintain the same style across the application.



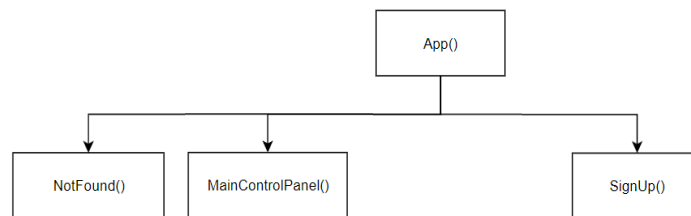
- Better performance as mentioned by the React team (Alpert, 2015);

Because of choosing the functional way of developing the application it hard to talk about a *class* diagram as everything is composed of functions (also referred as components) rather than classes. In this sense it is more appropriate to define the *component diagram*. That presents the components of the application and how they are connected.

Another point that needs to be underlined, is that React prefers downstream communication (from parent to child). Even if it is possible to achieve bi-directional communication - using data *bags* or third-party libraries like *Redux*, it is generally recommended to avoid upstream communication.

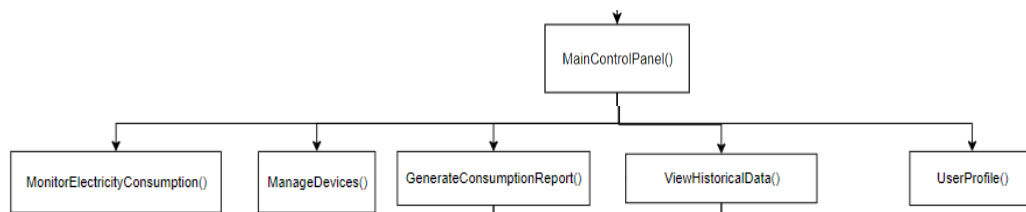
The complete diagram can be found in Appendix P.

Figure 14 - Frontend Application Class Diagram Part I



In the diagram above, it can be seen that the root component is the *App()* component with 3 children. The *App* component serves as the gateway when accessing the application then routing to the corresponding children based on the entered route and/or access rights.

Figure 15 - Frontend Application Class Diagram Part II



The component that encompasses most of the functionalities is *MainControlPanel()* as it can be seen it has five children components. The five child components map one-to-one to the *Use Cases* that the user interacts with, as they contain the functionalities belonging to those use cases. The responsibility of the *MainControlPanel* is to be a central point of distribution for different functionalities. It acts as an accessor and passes user's identifications to the child components.

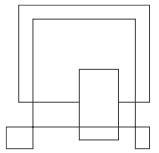
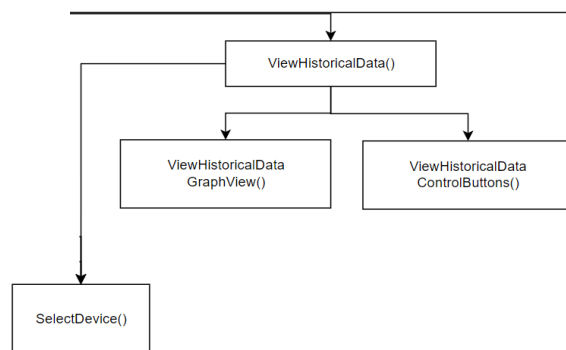


Figure 16 - Frontend Application Class Diagram Part III



Drilling one step down the *ViewHistoricalData()* component is presented. It has three children that have very specific responsibilities:

- *ViewHistoricalDataGraphView()* – presents the data in a plot;
- *ViewHistoricalDataControlButtons()* – defines the control buttons for navigating data;
- *SelectDevice()* – defines how one or more devices are selected;

As mentioned above, it can be seen that the functional way of defining the components, offers a great way of creating small components with single responsibilities that are then put together to create the application.

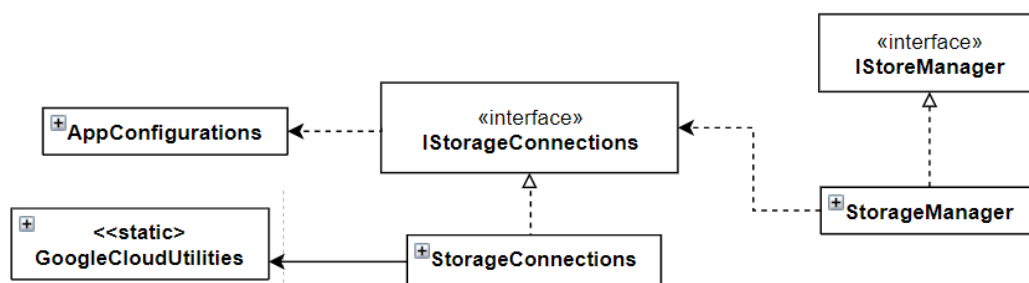
Backend Application Class Diagram

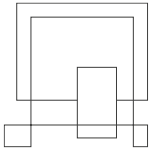
For the Backend Application, in addition to the classic class diagram, a simplified diagram will be presented that will allow to focus more on the relationships between the classes. The diagrams can be found in Appendix P.

Dependency injection play an important role in how the application is structured. It offers an inexpensive way of injecting functionalities without too much overhead. Because of that many relationships between classes are defined as *dependency*.

One main role of the backend application is to interact with storages – both PostgreSQL database and Firestore database. Therefore, two classes were made that act as proxies in abstracting from connecting and interacting with the storages: *StorageConnections* and *StorageManager*.

Figure 17 - Backend Application Class Diagram Part I

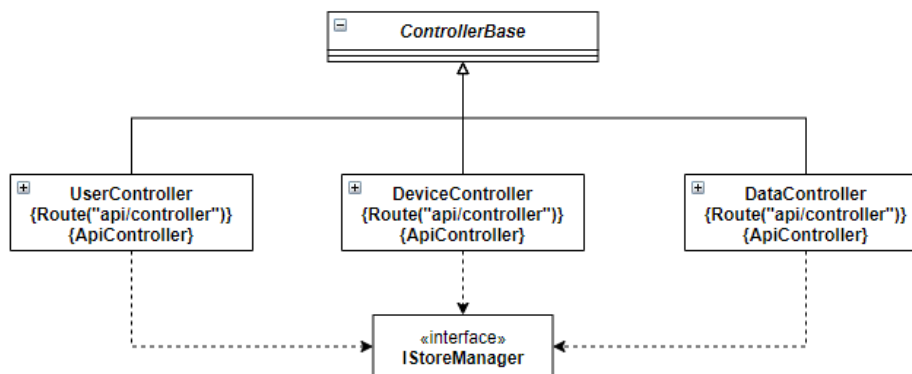




It can be seen that *StorageManager* depends on *StorageConnections* for getting the connections with the storages, in order to offer the functionalities for interacting with the databases. The *StorageConnections* relies on the *AppConfigurations* and *GoogleCloudUtilities* that containing various details for creating the connections.

And as mentioned the *StorageManager* is used by various parts of the application, especially the controller that are used by the frontend application.

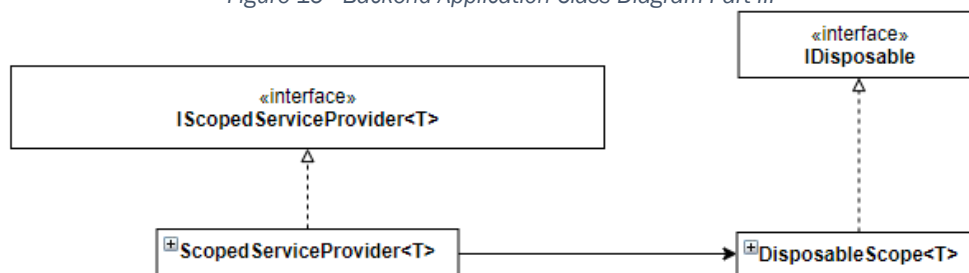
Figure 18 - Backend Application Class Diagram Part II

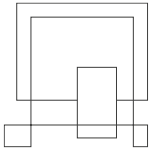


As described previously, when using dependency injection, it is very important to make sure that services with a longer lifetime are not used withing services with a short lifetime – ex. Singleton services inside Transient Services. To avoid that it is possible to create a scope, after which the service will be disposed. But creating scopes every time a lifetime overlap occurs results in having a lot of boilerplate code. In addition, to achieve that, the *IServiceProvider* interface needs to be injected, which in many cases results in being forced to pass a reference of the *ServiceProvider* down the class creation chain.

To avoid that a *pattern* like design was created, where a scope provider is defined an injected in the required class. By doing so it is always ensured that the instance is correctly disposed, and it avoids having a lot of boilerplate code.

Figure 19 - Backend Application Class Diagram Part III



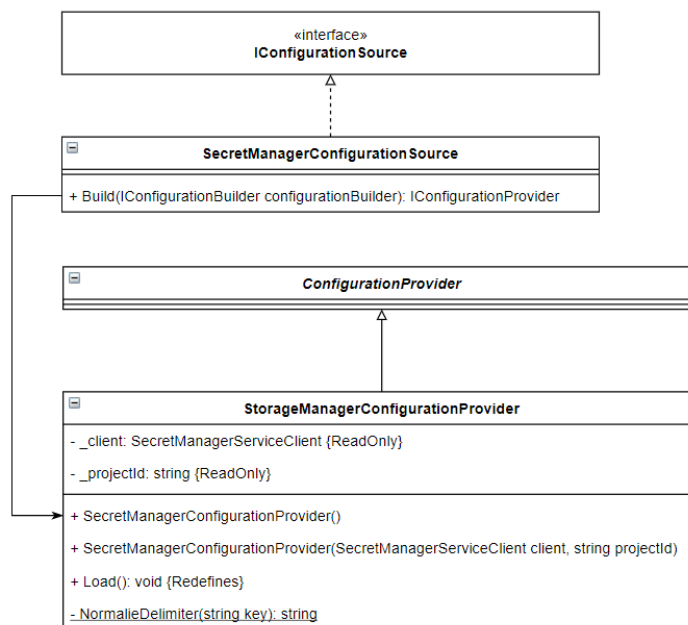


The *ScopedServiceProvider* was defined using the generics functionality of the .NET. It takes a T-class type and returns a scoped service provider in which the service can be utilized.

In order to utilize the *Options Pattern* discussed previously, to have the *AppConfigurations* class that holds various configuration information it was required to define a custom way of accessing the configurations. For this project it was decided to leverage the *Secret Manager* service provided by Google Cloud Platform as the central holder of the configurations – more on pros of using Secret Manager service in the *Security Chapter*.

Unfortunately, ASP.NET does not provide an out of the box way of connecting and utilizing the Secret Manager – it is rather focused on offering way of connecting to its Azure counterpart. To overcome this, it was needed to define a custom way of connecting to the service. It was possible by utilizing the *IConfigurationSource* interface and *ConfigurationProvider* abstract class offered by ASP.NET.

Figure 20 - Backend Application Class Diagram Part IV



By defining the above structure, it was possible to allow having a central place for storing and fetching application configurations.

In addition, to make it more convenient to add the service to the application, the method extension capabilities of the language were utilizing, by creating an extension method called **AddGoogleSecretsManager()** that extends from the *IConfigurationsBuilder*. It offers an easy and convenient way of adding the Secret Manager at the start of the program.

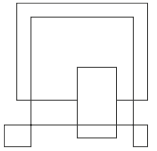
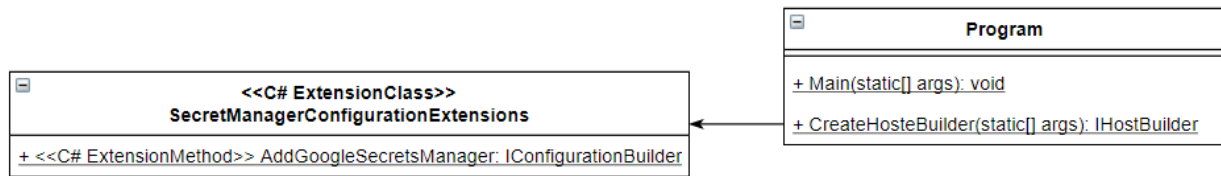


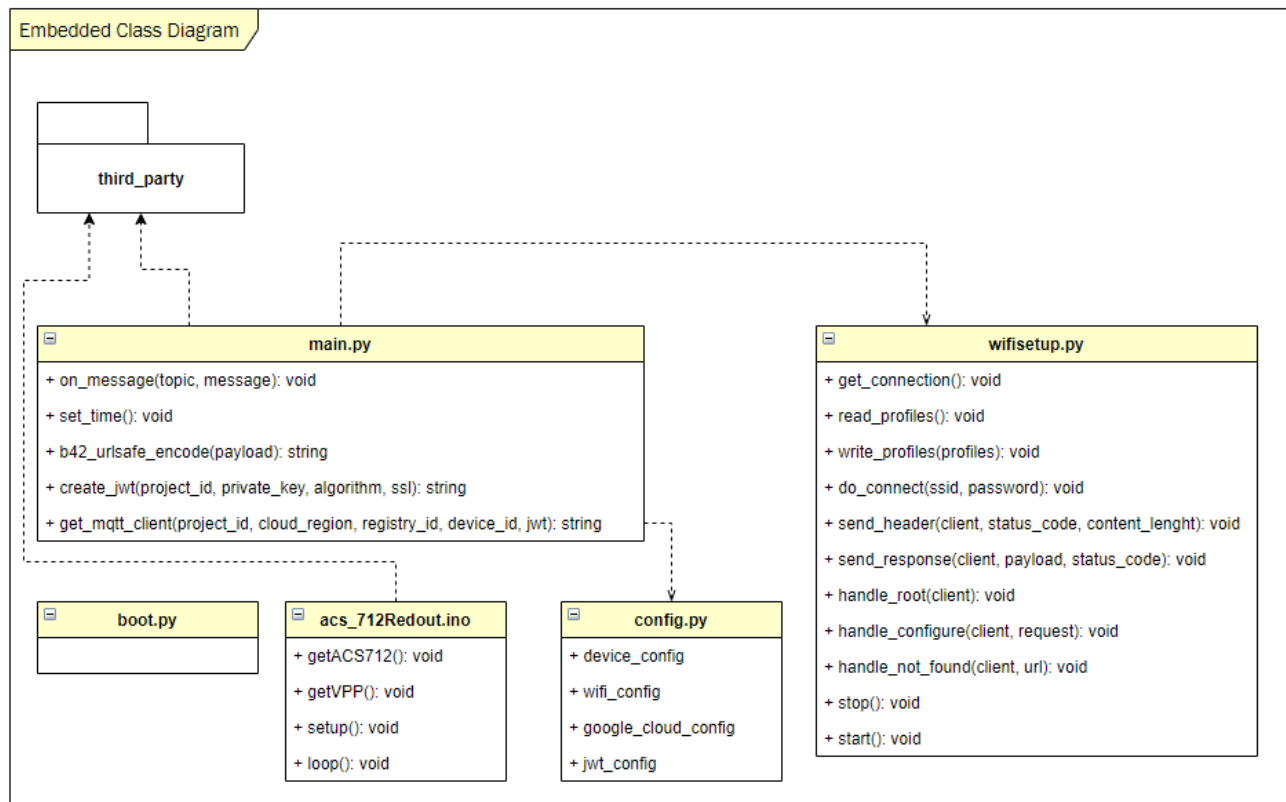
Figure 21 - Backend Application Class Diagram Part V

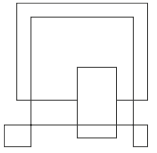


Embedded Application Class Diagram and Flow Diagram

The embedded application is designed as a functional system, because of that the class diagram does not bring too much value in explaining how the different functions interact with each other. To understand that better, a sequence and a flow diagram will be created that will underline how the system works internally. Nevertheless, it is still useful to visualize what the system consists of, therefore a simple class diagram that includes the files as *classes* and how they depend between themselves. The diagram can be found in Appendix P.

Figure 22 - Embedded Application Class Diagram



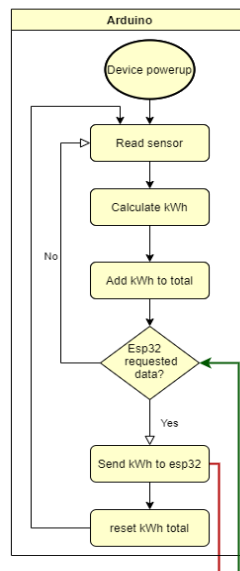


As can be seen in the diagram, five main files that are grouping together different related functionalities:

- *main.py* – the main class running on the device;
- *wifisetup.py* – class containing the functionality for connecting to the WIFI network;
- *config.py* – contains a series of configuration objects that are exported;
- *boot.py* – it does not contain any logic or functionality, but it is required by the device in order to boot-up;
- *acs_712Readout.ino* – an Arduino file, that has the logic for reading data from the sensor and sending it to the *main.py* through the UART interface;

In addition to the class diagram, a *Flow Diagram* was created that takes the concepts of the embedded application and shows how the flow of the application is defined. It is an essential diagram on visually understand how the system works and what is the flow within the application. The complete diagram can be found in Appendix Q.

Figure 23 - Flow Diagram Part I



The flow diagram includes the two modules:

- Arduino logic;
- ESP32 logic;

In the the crop above the Arudino logic is presented. The flow presents the logic neede for collecting sensor data about electricity consumption and passing it to the ESP32 via the *UART* channel.

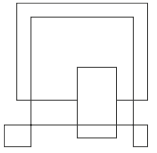
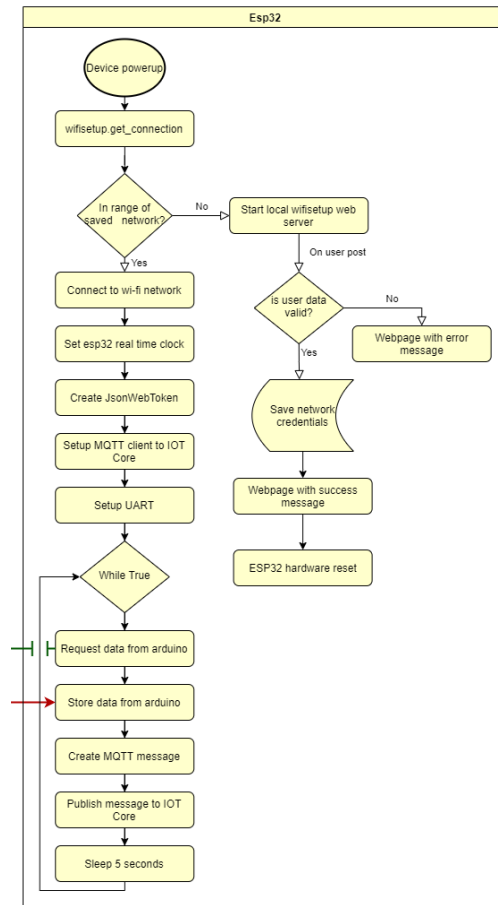


Figure 24 - Flow Diagram Part II



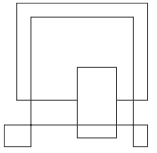
In second part presents the logic for ESP32 device. The flow includes logic for setting up the connection (mqtt and wifi) fetching the data from the Arduino and sending the data up to the IoT Core. The connection between the two device are signified with the *red* and *green* UART connection.

Device Schematics

The device schematic is used to show all the electrical connections between the different components that make up the embedded device. The device schematic can be found in the Appendix R.

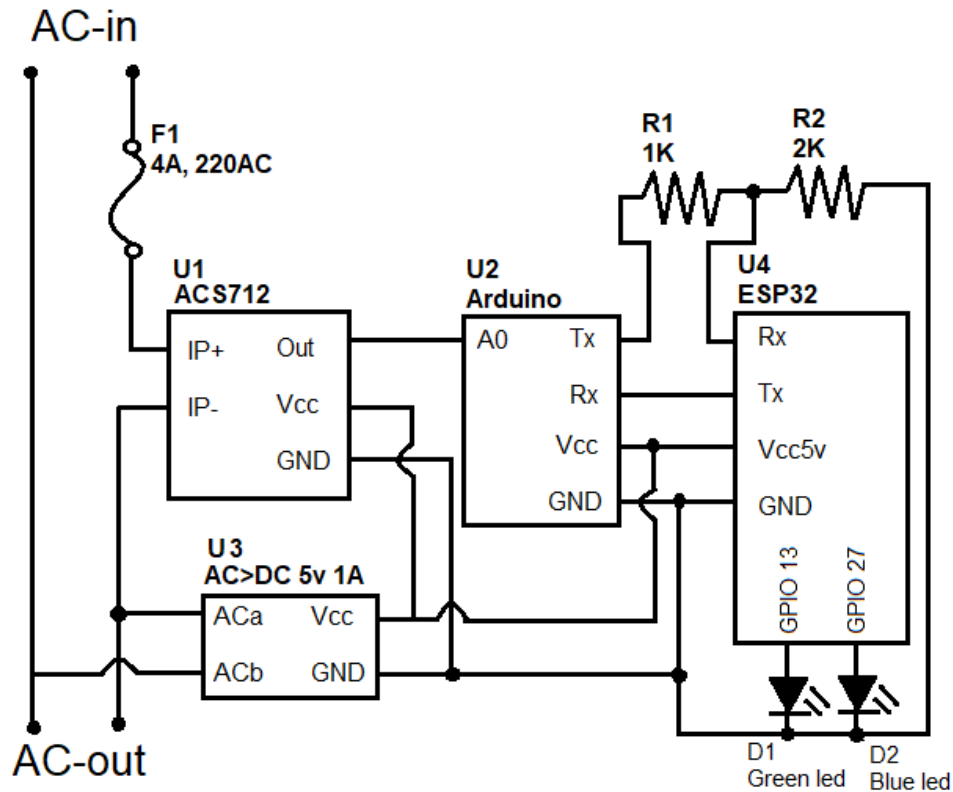
The schematic includes:

- U1 – The AC current measurement breakout board (ACS712)
- U2 – Off the shelf Arduino Pro/Mini 5V
- U3 – AC to DC converter, converts mains power to 5V to be used by the electronics
- U4 – ESP32 HUZZAH
- R1 – 1K Ohm resistor (part of the voltage divider)



- R2 – 2K Ohm resistor (part of the voltage divider)
- F1 – Glass fuse
- D1 – Green LED – used for interaction with the user
- D2 – Blue LED – used for interaction with the user

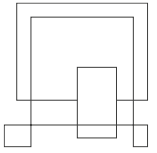
Figure 25 - Device Schematic



As can be seen in the schematics, the electronics are protected by a glass fuse to avoid critical failure of the electrical lines. The electronics are supplied with power through U3 in order to provide power only when the device is in use and to avoid the use of a battery.

The current gets measured by having the mains power intersected by U1 which then gets read out by U2. Next, U2 and U4 are connected by a two-wire serial interface – Tx, Rx – in order to allow for *UART* communication between the two boards.

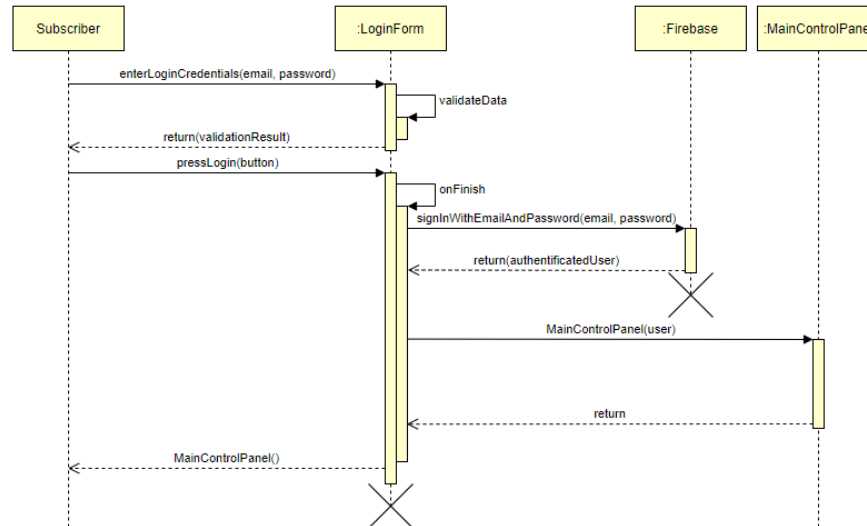
Because U4 input pins are rated for 3.3V and not the 5V that the pins of U2 are outputting it was needed to create a voltage divider with R1 and R2 to step down the voltage from U2 to 2.5V.



Sequence Diagram

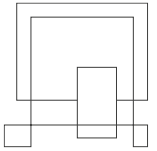
Sequence Diagrams help identify the flow of how the commands are passed throughout the system. In addition, it helps identifying flaws in the system at an early stage preventing high-cost mistakes to get into implementation. For this purpose, a presentation at the *Manage Devices Sunny Scenario* diagram will be made. Sequence Diagrams can be found in Appendix S.

Figure 26 - Sequence Diagram Part I



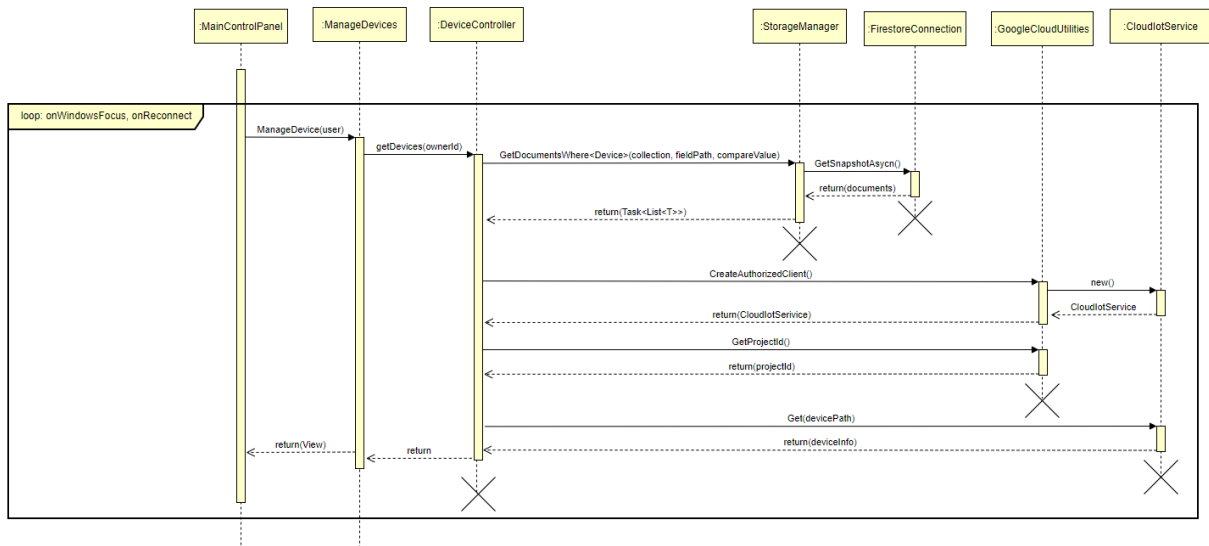
The purpose of the *Manage Devices* task is to display for the user all the existing devices registered under his/her profile and to provide information about their current state. The first part of the diagram presents the logging in feature and getting the subscriber to the *MainControlPanel*.

The process starts with the subscriber entering the login information. The *LoginForm* component validates entered data on an as-it-gets-entered basis, to ensure that the email has the correct form and the password the correct length.



When the subscriber presses the login button, its credentials are validated against records in the *Firebase* and it redirects the user to the *MainControlPanel*. Of course, it being a *Sunny Scenario* the entered credentials are correct, otherwise the subscriber will be informed.

Figure 27 - Sequence Diagram Part II



Once the subscriber selects the *ManageDevices* tab in the *MainControlPanel*, a process of fetching the devices and their states starts. Fire the *ManageDevices* component using the *Axios* and *React-Query* libraries sends an HTTP Get request to the *DeviceController* in the backend API.

In the backend first the controller fetches all the devices associated with the provided user id, through the *StorageManager* interface. After having the list of devices belonging to the user, the controller gets the *CloudIoTService* instance, which is an interface provided by Google to interact with the *IoT Core*.

With an instance of the *CloudIoTService* the controller is now able to fetch the latest status of the devices, create device objects and serialize them as JSON.

When the *ManageDevice* gets the response from the controller, it creates the user interfaces and presents the view to the subscriber.

Another important sequence diagram is for the flow of the embedded system. The diagram includes the calls made for achieved data collection and transportation, setup and the main loops of the systems.

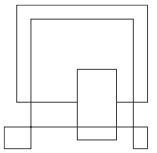
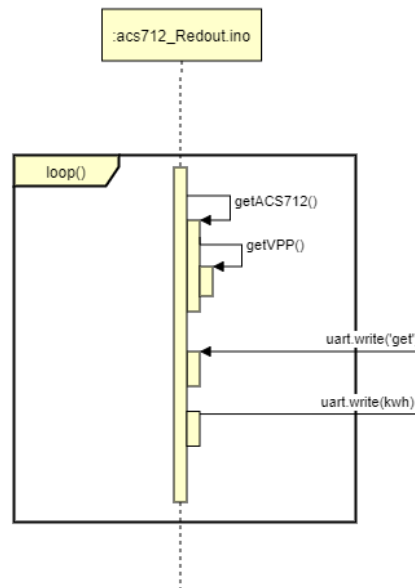
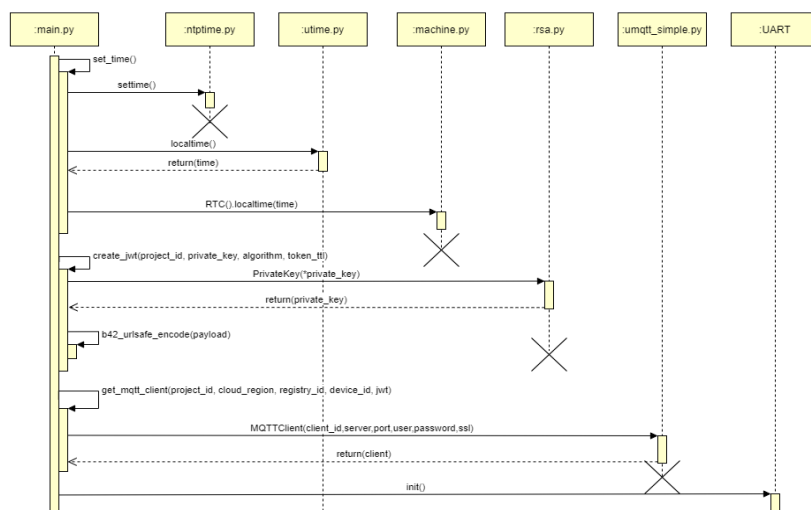


Figure 28 - Embedded Sequence Diagram Part I



As can be seen, everything is included in the main loop, the loop has the main role to collect the data and check for the commands that are sent via the *UART*. When the loop receives the *get* command, it writes the accumulated *kwh* since the last message over the *UART*.

Figure 29 - Embedded Sequence Diagram Part II



In the part above, the initialization of the esp32 is presented – when the WIFI is already setup. The process includes setting the time, creating the JSON Web Token, making the connection with the MQTT broker and initializing the UART.

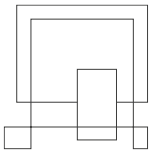
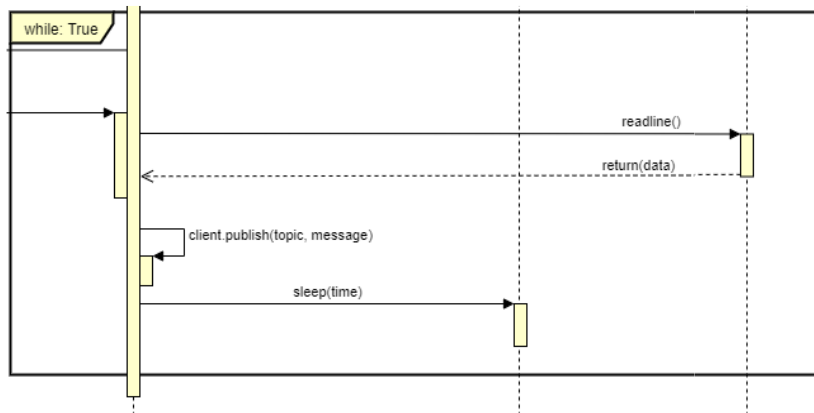


Figure 30 - Embedded Sequence Diagram Part III



The last part presents the main loop of the esp32 device. It includes reading the data from the *UART* channel, publishing the message to the IoT Core and then a time-based delay for the system to sleep. The complete diagram can be found in Appendix S.

Database Structure

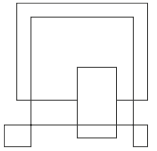
This project utilizes two types of databases:

- Relational Database – PostgreSQL – for storing timeseries data of electricity consumption;
- Document Database – Firestore – for storing model objects;

The databases have a simple structure, that offers an easy way of interacting with them. The database structure can be found in Appendix T.

PostgreSQL Database

The PostgreSQL database serves as a long-term storage for the electricity consumption data. The system interacts with it using *repoDB* library through object-relation mapping. Because of that rather than the classic relational database structure, it was defined in such way that allows using object-relational mapping.



What this means is that instead of thinking about the database as a series of tables connected with each-other through relationships, it is rather a representation of the .NET objects in SQL. The database consists of a single table called *ConsumptionRecord* that holds all the electricity consumption information and it maps one to one to the .NET class.

Figure 31 - ConsumptionRecord Table

| ConsumptionRecord | |
|-------------------|-------------|
| | Timestamp |
| | OwnerId |
| | DeviceId |
| | Consumption |

Firestore Database

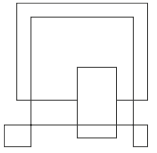
While it is hard to talk about a *structure* or *diagram* for document-based databases. In regards to this project, the documents are relatively uniform with similar fields and/or properties. Because of that a presentation of the existing structure will be made. It is important to note that the structure can be changed or modified as the documents do not guarantee to have the same structure, the only persistent element being the id of the documents and collections.

Firestore is used in this project to store models of the system: *User Model* and *Device Model*. It was decided to utilize a document, NO-SQL database for this, because it offers a high degree of flexibility.

Figure 32 - Firestore Database

| Device | |
|--------|----------|
| | id |
| | name |
| | owner_id |

| User | |
|------|------------|
| | user_id |
| | first_name |
| | last_name |
| | gender |
| | email |
| | devices |



User Interface and User Experience

The interaction with the system for the users is done through the Frontend Application. The application is build using the *React Framework* that offers a robust and reactive user experience with little overhead.

UI Library

While *React* is the framework that powers the application, its user interface is supported through UI frameworks. For this project a few UI frameworks were considered, that would have allowed creating a pleasant and easy to use environment for the users:

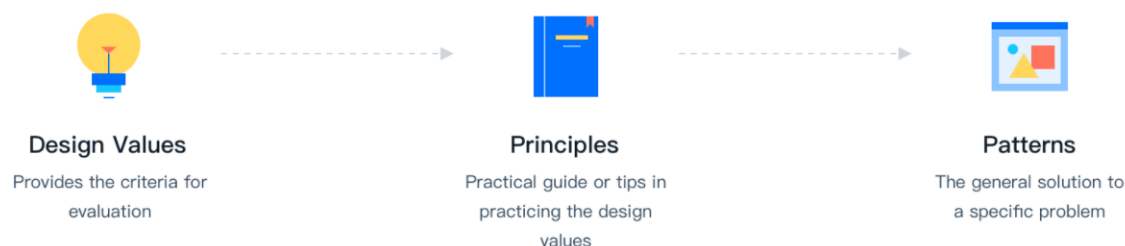
- Reactstrap / React-bootstrap –both libraries offer a similar design at the original Bootstrap originally developed at Twitter but geared towards React framework. They export prebuild components, highly customizable and easy to use; (Material Design, n.d.)
- Material Design React – is the counterpart of the Google’s Material Design UI framework that offers beautiful and easy to use components design using Google’s design principles;
- Carbon Design System – this library is created, maintained and used by IBM, with a focus on enterprise design of UI components; (Carbon Design System, n.d.)
- Ant Design – is a library born in China, following the same principles as Material Design, but being *React born*. The has powerful prebuild components, easy to utilize in projects, with a high variety of components; (Ant Design, n.d.)

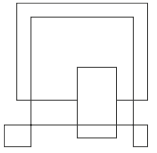
After careful consideration and weighing of the pros and cons for the listed libraries, the decision was made to use *Ant Design* as the project’s UI library.

Ant Design has strong design principles focusing on *natural user cognition and user behavior* with values circling around offering a comfortable user experience.

Ant Design has well defined and robust methods for creating general solutions to specific problems, but utilizing their design values, creating a set of principles from those values and then define patterns for solving a specific problem.

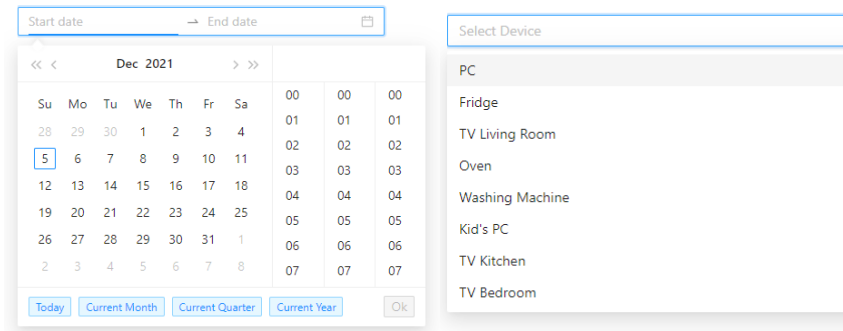
Figure 33 - Ant Design Problem Solving Road





In this project many times it is required to have selections – selection of devices, selection of dates – that are provided by using the pre-build components offered by Ant Design.

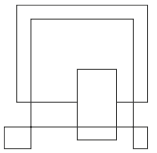
Figure 34 - Ant Design Selectors Examples



Charting Library

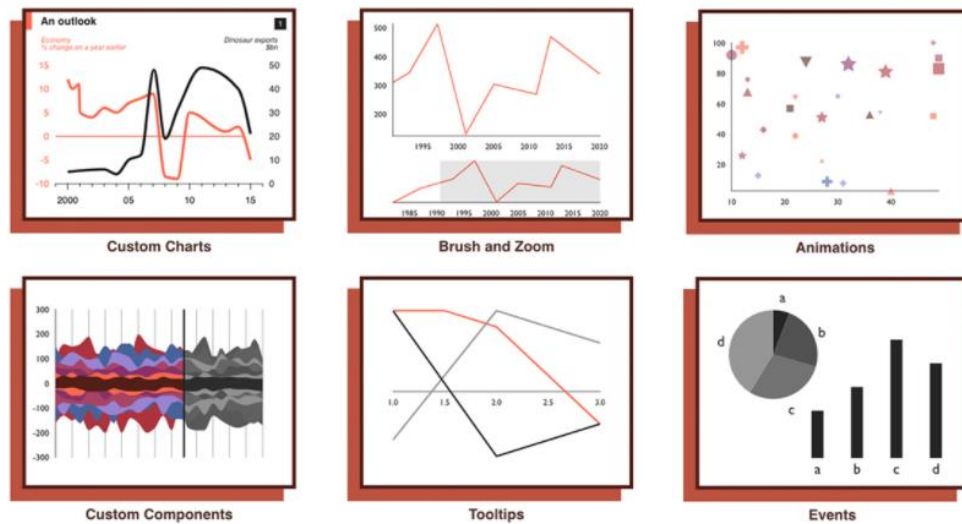
Another essential topic for this project is data visualization through plots. There are many libraries that offer beautiful visualizations of data in a graphical view, some of them were considered for this project:

- Plotly React – is a large library for creating charts and data visualization, with support for multiple languages including *JavaScript*, *Python* and *Julia*. *Plotly* is based on the *D3 charts* library, that offers capabilities for producing dynamic and interactive visualizations in web browsers; (Plotly, n.d.)
- Recharts – is a React based charting library, being considered one of the most reliable chart libraries available, with native support for SVG and light dependency on *D3* submodules; (Reactcharts, n.d.)
- Victory Charts – is a flexible charting library that offers an easy cross-platform – web and mobile – developed by Formidable; (Formidable, n.d.)
- Nivo Rocks – is another collection of React components developed on top of *D3 charts*, offering server-side rendering of plots; (Nivo Rocks, n.d.)
- React-Vis – is a library developed by Uber and focuses on ease of use and a very well-defined documentation; (Uber, n.d.)



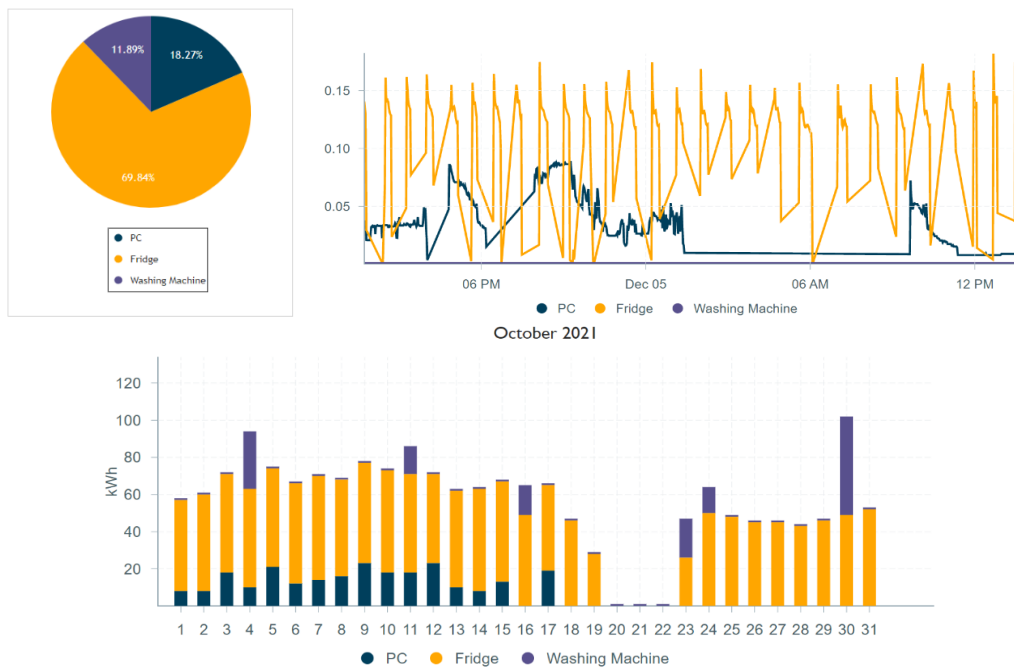
After weighting of the benefits and overheads of each library, the design was made to use *Victory Charts* in this project. Some of the benefits of using *Victory Charts* is that it is a stable library, with great documentation and offers future development possibilities for mobile devices.

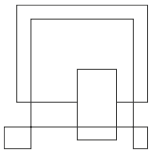
Figure 35 - Victory Charts Example



In this project the library is used to provide visualization capabilities of the electricity consumption data, either raw or aggregated as line, pie and bar plots.

Figure 36 - Victory Charts Examples in the project





Security

Nowadays, security is a critical part of any infrastructure. Ensuring that both the system's resources and user's data is safe from malicious intents is a must, and has severe repercussions if not. Even if this project attempts to present a proof of concept of a system, some attention was still paid into ensuring that best practices are used throughout the system.

Luckily for developers, there are many tools that can help in ensuring a higher level of security of the system. And it can be achieved through two main ways:

- Security through code and development practices – ensuring security practices through code that reflects that;
- Cloud configurations – a high level of security can be ensured by utilizing and configuring the resources offered by the cloud provider;

Cloud Provider Security Configurations


A great level of security can be achieved only by having correct *Identity and Access Management (IAM)* rules. Limiting access for services, to only be able to access what they need. This is done through assigning roles for services, and denying access if something without that role is trying to access a resource.

A good example can be the Backend Application, it needs to have quite a broad access level:

- Access to the database;
- Access to the Secret Manager;

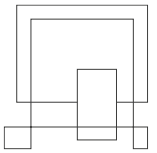
To aid the developer, the cloud provider creates service principals for each service, which is a representation of the service *persona*. The service principal can then be utilized to assign roles to it.

Figure 37 - App Engine Service Principal

| <input type="checkbox"/> Type | Principal ↑ | Name | Role |
|---|--|------------------------------------|--|
| <input checked="" type="checkbox"/>  | bachelorproject-324708@appspot.gserviceaccount.com | App Engine default service account | Cloud SQL Client Editor Secret Manager Secret Accessor |

As it can be seen the service has 3 roles:

- Editor – added by default and needed for the application to operate correctly;
- Cloud SQL Client – a role added to allow the service to access the PostgreSQL Database;
- Secret Manager Accessor – a role added to allow the service to access the Secret Manager;



These roles come with a certain access level, that is not the highest, but high enough to ensure that the application can function successfully. Without having these roles, the access to the database or Secret Manager is denied even if the accessor has the credentials.

Another good tool offered by the cloud provider is the *Secret Manager*. It offers an easy and secure way of storing application credentials. The secrets in the service are encrypted and the access is granted on a IAM basis, ensuring that only the allowed applications can access it. For the current project, the main credentials stored in the Secret Manager, are database credentials and the public key needed when configuring new devices.

Figure 38 - Secret Manager

| <input type="checkbox"/> | Name ↑ | Location | Encryption | Labels | Created | Expiration | Actions |
|--------------------------|------------------------|--------------------------|----------------|--------|-------------------|------------|---------|
| <input type="checkbox"/> | DatabaseConnectionName | Automatically replicated | Google-managed | None | 11/30/21, 4:42 PM | | ⋮ |
| <input type="checkbox"/> | DatabaselP | Automatically replicated | Google-managed | None | 10/22/21, 3:12 PM | | ⋮ |
| <input type="checkbox"/> | DatabaseName | Automatically replicated | Google-managed | None | 10/22/21, 3:17 PM | | ⋮ |
| <input type="checkbox"/> | DatabasePass | Automatically replicated | Google-managed | None | 10/22/21, 3:10 PM | | ⋮ |
| <input type="checkbox"/> | DatabaseUser | Automatically replicated | Google-managed | None | 10/22/21, 3:10 PM | | ⋮ |
| <input type="checkbox"/> | PublicKey | Automatically replicated | Google-managed | None | 12/2/21, 7:21 PM | | ⋮ |

Other configurations used for ensuring the security includes but are not limited to:

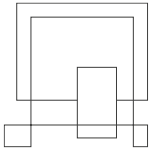
- Cloud SQL Proxy – to have a more secure connection to the database;
- Role based access between the Functions, Backend and Frontend;
- Public Key access for the devices to the IoT Core;

Security through code and development

Security through development refers to various practices that a developer can apply in order to ensure a good and secure environment when developing. A good example of this was to make sure that no secrets, keys, credentials or connection string are leaked through the version control and repositories. It is often the case that some sort of confidential information, is mistakenly committed to a repository creating an undesirable result for the project.

When it comes to security through code, in this project it refers to code parts that are defined only to ensure a greater level of security. A good example of this is the IoT device that needs to connect to the cloud. To ensure that only the approved devices are connecting to our IoT core, a security measure in form of having credentials presented upon connection is defined.

To be able to authenticate against the IoT Core the devices utilizes an **RSA256** private key and create a **JSON Web Token** that then is used for authentication. In this way it is ensured that only the allowed devices connect to the IoT Core.



Implementation

In this chapter, a look on how various modules of the system were implemented. It will not cover the entire code base but rather focus on highlighting some of more important parts of each application. The source code can be found in Appendix U.

Embedded Application

Embedded Application was building MicroPython and a small part of Arduino. The main code for running the application is made in MicroPython, Arduino only containing the logic for collecting data from the electricity sensor and calculations for transforming the raw sensor data to kwh.

The *main.py* file contains the code for setting up the connections – except WIFI connection –, requesting, receiving and sending data up to cloud.

Figure 39 - Setting up the MQTT Client

```
def get_mqtt_client(project_id, cloud_region, registry_id, device_id, jwt):
    client_id = 'projects/{}/locations/{}/registries/{}/devices/{}'.format(
        project_id,
        cloud_region,
        registry_id,
        device_id
    )

    client = MQTTClient(
        client_id.encode('utf-8'),
        server=config.google_cloud_config['mqtt_bridge_hostname'],
        port=config.google_cloud_config['mqtt_bridge_port'],
        user=b'ignored',
        password=jwt.encode('utf-8'),
        ssl=True
    )

    client.set_callback(on_message)
    client.connect()
    client.subscribe('/devices/{}/config'.format(device_id), 1)
    client.subscribe('/devices/{}/commands/#'.format(device_id), 1)

    return client
```

In the example above the function to setup the MQTT client. The function is depended on the *MQTTClient* library that provides the main logic for setting up the client. To setup the client, a few information needs to be provided: *client id*, *server name*, *port*, *user* and *password*. It is an important element of the of the system, as the MQTT client allows the communication with the IoT Core for transmitting the electricity consumption data to the backend.

The main loop, utilizes the different elements that were setup to be able to collect the data from the *Arduino* and sending it to the cloud via the internet connection.

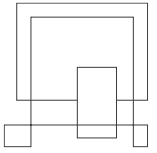


Figure 40 - Main Loop

```
while True:

    uart.write('get')
    data = uart.readline()

    message = {
        "DeviceId": config.google_cloud_config['device_id'],
        "Consumption": data
    }

    '''Create topic and send data to the Google IOT Core'''
    mqtt_topic = '/devices/{}/{}'.format(config.google_cloud_config['device_id'],
    'state')
    client.publish(mqtt_topic.encode('utf-8'), ujson.dumps(message).encode('utf-
8'))

    utime.sleep(5)
```

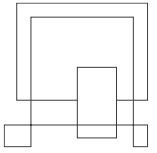
The *Arduino* code is responsible for getting the raw data and apply the calculations for getting the electricity consumption.

Figure 41 - Electricity Consumption Calculations

```
void getACS712()
{
    Vpp = getVPP();
    Vrms = (Vpp/2.0) *0.707;
    Vrms = Vrms - (calibration / 10000.0);
    Irms = (Vrms * 1000)/Sensitivity ;
    if((Irms > -0.015) && (Irms < 0.008))
    {
        Irms = 0.0;
    }
    power = (Supply_Voltage * Irms) * (pF / 100.0);
    last_time = current_time;
    current_time = millis();
    kWh = kWh+ power *(( current_time -last_time) /3600000.0);
}
```

After getting the raw sensor data from the *getVPP()* and then applies a series of calculations to determine what was the electricity consumption. It is a rolling sum that constantly increases the kWh value until it is sent to the esp32 when it gets reset.

Lastly, the implementation for setting up the WIFI settings to allow the device to connect to the internet is contained in the *wifisetup.py* file. The implementation sets a small webpage that is used for collecting the WIFI credentials and saves them in the device's configurations. To allow the user to



access the webpage, the device creates its own WIFI network that a user can connect to and access the page on the 192.168.4.1 IP address (port 80).

The user then is presented with all the available networks, that he/she can select and insert the password for the network. When the user has added the WIFI information, the data is saved under a *NETWORK PROFILE* on the local flash memory.

Figure 42 - Network Profile Setup

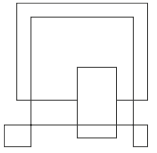
```
def write_profiles(profiles):
    lines = []
    for ssid, password in profiles.items():
        lines.append("%s;%s\n" % (ssid, password))
    with open(NETWORK_PROFILES, "w") as f:
        f.write(''.join(lines))
```

After the network profile is saved, the *do_connect* function uses the WIFI information to connect to the network.

Figure 43 - Connect to the WIFI network

```
def do_connect(ssid, password):
    wlan_sta.active(True)
    if wlan_sta.isconnected():
        return None
    print('Trying to connect to %s...' % ssid)
    wlan_sta.connect(ssid, password)
    for retry in range(100):
        connected = wlan_sta.isconnected()
        if connected:
            break
        time.sleep(0.1)
        print('.', end='')
    if connected:
        print('\nConnected. Network config: ', wlan_sta.ifconfig())
    else:
        print('\nFailed. Not Connected to: ' + ssid)
    return connected
```

The function has the logic for retries if it was unable to connect from the first try.



Backend Application

Backend Application was build using ASP.NET 3.1 and is a web application that implements a RESTful API. Its primary role is to be a connection point between the devices, storage and frontend, with roles in ingesting, storing, fetching and aggregating data.

The system files are grouped in six namespaces based on their role and functionality within the system:

- Configuration Manager – includes classes needed for setting up *Option Pattern* using *Google Secret Manager*;
- Controllers – contains the controllers of the system, that allow interaction with other applications of the solution;
- Extensions – a namespace to hold extension classes;
- Interfaces – groups all interfaces of the system in a single place;
- Models – represents the models used throughout the system, models that also can be found in the databases and other parts of the solution;
- Utilities – utility classes that hold specific functionalities for the system;

The first part of the system to look at is the *Startup.cs* file. Its role is to setup the components of the application, especially the dependency injection container, that allows injecting service in various parts of the system.

Figure 44 - Startup class

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddLogging(config =>
    {
        config.AddConsole();
    });

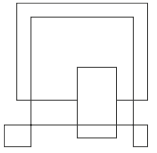
    services.Configure<AppConfigurations>(Configuration);
    services.AddCors();
    services.AddControllersWithViews();

    services.AddSingleton<IStorageConnections, StorageConnections>();
    services.AddSingleton<IProcessingScheduler, ProcessinScheduler>();

    services.AddTransient<IStorageManager, StorageManager>();

    services.AddTransient<IScopedServiceProvider<IStorageManager>,
    ScopedServiceProvider<IStorageManager>>();
}
```

In this method the services are setup, including the service responsible for logging, and other utility classes used in the system. The *Singleton* and *Transient* lifetimes are used for the services. An important thing to notice is that *IStorageManager* service class is injected two times. This is to allow



the possibility of utilizing them in a service with a longer lifetime, and will be described further in the paper.

One of the entities of the system is a device, it contains and describes information related to a device, and is used through the solution.

Figure 45 - Device Model Class

```
[FirestoreData]
public class Device : IDevice
{
    [FirestoreProperty("id")]
    public string Id { get; set; }

    [FirestoreProperty("owner_id")]
    public string OwnerID { get; set; }

    [FirestoreProperty("name")]
    public string Name { get; set; }

    [FirestoreProperty("state")]
    public int? State { get; set; }

    [FirestoreProperty("last_state_time")]
    public DateTime? LastStateTime { get; set; }
}
```

It can be seen that it implements the *IDevice* interface. In addition, it has attribute annotations that help for easier mapping when fetching device information from the *Firestore Database*.

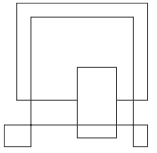
For getting access to various services of the cloud provider it is important to get a handle of the project id. But the project id can be fetched from the cloud provider when the application is deployed, unfortunately it cannot be done in the similar manner when running the application locally. To combat that, the implementation is split based on whether it is run in *DEBUG* (locally) or *RELEASE* (production) mode.

Figure 46 - Google Utilities Function

```
public static string GetProjectId()
{
    #if DEBUG
        var projectId = Environment.GetEnvironmentVariable("PROJECT_ID");
    #elif RELEASE
        var instance = Platform.Instance();
        var projectId = instance?.ProjectId;
    #endif

    if (string.IsNullOrEmpty(projectId))
    {
        return null;
    }

    return projectId;
}
```



In this way the implementation is independent from the project and can be easily run-in different environments. When running locally, the project id is provided via an *Environmental Variable* that is specific only to the developer's machine.

Another important functionality of the system, is to fetch data from the databases, this is achieved via the *StorageManager* class. When fetching data from the *Firestore* database, it was important to make the methods generic. In this way there is no need for different implementations based on the object fetched but rather only informing the method what object the caller is expecting.

Figure 47 - Firestore generic fetcher

```
public async Task<List<T>> GetDocumentsWhere<T>(string collectionName, string
fieldPath, object compareValue)
{
    var resultSnapshot = await _storageConnection
        .FirestoreConnection
        .Collection(collectionName)
        .WhereEqualTo(fieldPath, compareValue)
        .GetSnapshotAsync();

    var resultList = new List<T>();

    foreach (var documentSnapshot in resultSnapshot)
    {
        resultList.Add(documentSnapshot.ConvertTo<T>());
    }

    return resultList;
}
```

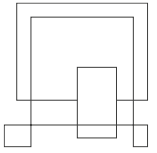
It can be seen that when calling the method, a reference of what type of object is passed via the *T* element, same element describing what type of object should the returning list hold.

Additionally, inserting and fetching data in the *PostgreSQL* database is another essential part of the system. In the example below can be seen two overload methods that can either insert a single element or perform a batch insertion of multiple elements.

Figure 48 - PostgreSQL fetcher

```
public async Task InsertConsumptionRecord(ConsumptionRecord consumptionRecord)
{
    using var connection = new
NpgsqlConnection(_storageConnection.PostgreSQLConnectionStringBuilder.ConnectionString);
    await connection.InsertAsync(consumptionRecord);
}

public async Task InsertConsumptionRecord(List<ConsumptionRecord> consumptionRecords)
{
    using var connection = new
NpgsqlConnection(_storageConnection.PostgreSQLConnectionStringBuilder.ConnectionString);
    await connection.InsertAllAsync(consumptionRecords);
}
```



The injection of the classes is made through the constructors of the receiving class. An example can be seen in the *ProcessingScheduler* class, that get injected with the *Logger* and *StorageManager*.

Figure 49 - Dependency Injection example

```
public ProcessingScheduler(ILogger<ProcessingScheduler> logger,
IScopedServiceProvider<IStorageManager> storageManagerScopeProvider)
{
    _storageManagerScopeProvider = storageManagerScopeProvider;
    _logger = logger;
}
```

It can be seen that instead of injecting directly the *StorageManager* class, it is injected via the *IScopedServiceProvider* interface that allows getting a scope for the service, without risking to not have the class correctly disposed.

The implementation of the *ScopedServiceProvider* is simple but powerful, it allows for an elegant solution to have an easy way of injecting the scope of the services, without having to rely on the *IServiceProvider* container.

Figure 50 - ScopedServiceProvider example

```
public class ScopedServiceProvider<T> : IScopedServiceProvider<T> where T : class
{
    private readonly IServiceScopeFactory _scopeFactory;

    public ScopedServiceProvider(IServiceScopeFactory scopeFactory)
    {
        _scopeFactory = scopeFactory;
    }

    /// <summary>
    /// Gets the requested service and the scope for it.
    /// </summary>
    /// <returns>Disposable Scope Class containing the service.</returns>
    public DisposableScope<T> GetScopedService()
    {
        var scope = _scopeFactory.CreateScope();
        var requestedService = scope.ServiceProvider.GetRequiredService<T>();
        return new DisposableScope<T>(scope, requestedService);
    }
}
```

Lastly, an example from the *DataController* where data is aggregated to find what was the maximal value of an hour withing a given period. To easily achieve this the *Linq* library is utilized that provides a very *fluent* and *functional* way of working with data.

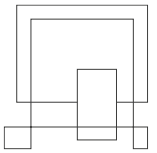


Figure 51 - Data aggregation using LINQ

```
MaxHour = record
    .GroupBy(item => new
    {
        Date = item.Timestamp.DayOfYear,
        Hour = item.Timestamp.Hour
    })
    .Select(g => new
    {
        Date = g.First().Timestamp,
        Value = g.Aggregate(0.0, (total, next) => total + next.Consumption,
        (finalValue) => Math.Round(finalValue, 4))
    })
    .MaxBy(item => item.Value)
    .First()
```

Frontend Application

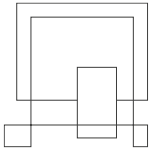
The Frontend Application is build using *React* framework, moreover it implements the components in a functional way, that is the new recommended style for React. The application implements many components as doing in this manner allows for easier re-rendering of specific components rather than the entire page. Since the rendering is done on the client side, it means it is less expensive and feel *snappier* for the end user.

The entry point to the application is the *App* components. It implements a switch which redirects the user to the correct views depending on

Figure 52 - Routing in App Component

```
<Switch>
  <LoggedInUserRouter path='/signup' component={SignUp} />
  <PrivateRoute exact path='/'>
    <Redirect to='/home' />
  </PrivateRoute>
  <PrivateRoute path='/home' component={MainControlPanel} />
  <Route component={NotFound} />
</Switch>
```

As can be seen, custom components are utilized for the routes, namely *LoggedInUserRouter* and *PrivateRoute*. These two components make sure to redirect the user if the user attempts to access a route that requires logging in (it redirects it to the signup page where the user has the possibility to log in). Vice versa if the user is trying to access the login page, while already being logged in, it will be redirected to the home page. Both empty path and home path lead to the home path, while anything else is caught by the *Not Found* component.



With version 16.8 React introduced *Hooks* which are similar to specialized functions that provide additional functionalities that it usually maintained across rendering. In this project React Hooks are used heavily as it makes the code shorter and easier to understand.

One of the most important components is the *MainControlComponent* as this is where a subscriber can navigate through the different features offered by the system. Next will be presented the functional way of declaration of a component as well as some hooks used in the application.

Figure 53 - Component declaration and React Hooks

```
const MainControlPanel = () => {  
  const { push } = useHistory();  
  const [user, loading, error] = useAuthState(auth);  
  const [collapsed, setCollapsed] = React.useState(false);  
  const [selectedView, setSelectedView] = React.useState(siderMenuItems[0]);
```

In this example, three types of hooks are presented:

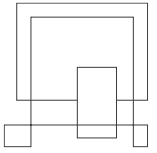
- useHistory() – which allows React to access Router's history object, in order to navigate between previous pages;
- useAuthState() – custom hook for fetching the state of the currently logged-in user. It offers information about the user, the state of fetching the user, and any errors if present.
- useState() – that is by far one of the most used and important hooks in this project, it allows to persistently store data over multiple re-renders, thus allowing the system to maintain the same versions of view, in this component useState is storing two instances:
 - a. collapsed – to know whether the side menu is collapsed or not;
 - b. selectedView – to know what view/menu was selected by the users;

The states cannot be assigned directly, thus it requires having a set method that is used for assigning.

One last thing to showcase from this component is how the side menu items are defined. It showcases very well the functional capabilities of *JavaScript*.

Figure 54 - Functional object field declaration

```
const siderMenuItems = [  
  {  
    key: 0,  
    icon: <BulbOutlined />,  
    textValue: "Monitor Electricity Consumption",  
    component: (ownerId) => (<MonitorElectricityConsumption ownerId={ownerId}/>)  
  }  
]
```



In this code above, we define a simple array, that holds the objects representing the side menu. Each object has four fields, that define the side menu, including its icon, key and text value. The last field is the component that it is supposed to render, and it is defined a function that takes the id of the current user and passes it to another function – in this case the *MonitorElectricityConsumption*. This way of declaration makes it very easy to iterate over different menus and render the correct ones.

One important library used within the application is *React-Query*. React query is an extremely stable and convenient way of making http calls to the API. It provides various hooks, depending on what is needed, and is very flexible and configurable.

In the example below the *useQuery()* hook is presented, that allow the application to make get requests to the backend for fetching electricity consumption data.

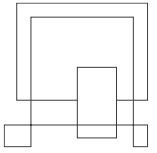
Figure 55 - useQuery Hook example

```
const { status, data, error } = useQuery(
  ['consumptionData', fetchInterval, selectedDevices],
  () => {
    var interval = fetchInterval.values();
    return getConsumption(
      props.ownerId,
      interval[1],
      interval[0],
      selectedDevices.map(device => device.Id)
    );
  },
  {
    enabled: (props.ownerId == null ? false : true) &&
      (selectedDevices.length > 0 ? true : false),
    refetchOnWindowFocus: true,
    refetchOnReconnect: true,
    refetchInterval: refetchFrequency.values
  }
);
```

The hook returns three variables, the current status of the fetching process that can be *success*, *loading* or *error* which allows for easily rendering elements that will let user know what is going on. The actual fetched data, and the error field if any occurred.

In its initialization, the hook takes a few elements:

- an array containing, the hook's id and a set of dependencies that when changed will trigger an re-fetching process;



- a function for fetching data, that in this case calls another function called *getConsumption* with the necessary time and devices restrains;
- a set of settings:
 - o enabled – sets when the hook is enabled, in this case only when the user id is not null and if there is at least one device selected;
 - o re-fetching options – on window focus, on reconnect and on interval;

As a last thing, the backward – from child to parent – communication will be presented. In React the components communicate only downwards, from parent to child. In this way it is ensured a correct rendering process. But sometimes it is needed to pass messages upwards between components, react provides a few ways of doing so. In this example we are utilizing one, namely passing a callback function that the child component can notify to change the state in the parent component.

The example below presents two components:

- *MonitorElectricityConsumption* – used for fetching data based on the selected fetch interval and passing it to the components responsible with creating visualizations;
- *FetchIntervalComponent* – which a component responsible for offering the user a way of changing the period for which to fetch data;

The parent component in this case is the *MonitorElectricityConsumption* and uses the *FetchIntervalComponent* – as per diagram in the *Frontend Application Class Diagram*. When a user changes the fetch interval the child component needs to inform the parent component about this change, so that the data can be re-fetched based on the new interval.

To do so, a simple function is defined in the parent space, that is receiving the new interval and is changing the state, causing a re-render.

Figure 56 - Simple State setter example

```
const handleFetchInterval = (newFetchInterval) => {  
  setFetchInterval(newFetchInterval);  
}
```

The function is then passed to the child component as a property.

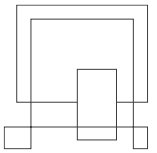


Figure 57 - FetchIntervalSelector component usage

```
<FetchIntervalSelector
  menuItems={menuItemsFetchInterval}
  selectedItem={fetchInterval}
  callBack={handleFetchInterval}
/>
```

Lastly the child component can now use this function to change the state in the part component with the selected fetch interval.

Figure 58 - Callback function call

```
const FetchIntervalSelector = (props) => {
  const handleMenuClick = (e) => {
    var selectedItem = props.menuItems.find(item => item.key === parseInt(e.key));
    props.callBack(selectedItem);
  };
};
```

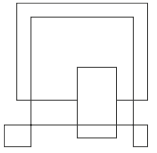
CI/CD Implementations

In the end, a few comments on implementing the *CI/CD Pipelines* will be made. Even if there are similarities between the frontend and backend pipeline, there are a few differences to be noted.

The frontend pipeline is build using the *cloudbuild.yaml* that describes the steps needed to create and deploy the application.

Figure 59 - Cloud Build Frontend

```
steps:
  # Build the container image
  - name: 'gcr.io/cloud-builders/docker'
    args: ['build', '-t', 'gcr.io/$PROJECT_ID/${_SERVICE_NAME}:${SHORT_SHA}', '.']
  # Push the container image to Container Registry
  - name: 'gcr.io/cloud-builders/docker'
    args: ['push', 'gcr.io/$PROJECT_ID/${_SERVICE_NAME}']
    timeout: '1800s'
  # Deploy container image to Cloud Run
  - name: 'gcr.io/google.com/cloudsdktool/cloud-sdk'
    entrypoint: gcloud
    args:
      - 'run'
      - 'deploy'
      - '${_SERVICE_NAME}'
      - '--image'
      - 'gcr.io/$PROJECT_ID/${_SERVICE_NAME}:${SHORT_SHA}'
      - '--region'
      - 'europe-west3'
    timeout: '1800s'
  images:
  - 'gcr.io/$PROJECT_ID/${_SERVICE_NAME}:${SHORT_SHA}'
```



As can be seen, there are three main steps in the pipeline: create the container, push the container image to the *Container Registry* and deploy the container image to *Cloud Run*.

The application is built to a *Docker* image, and the blueprint for building the image is defined in the *Dockerfile*.

Figure 60 - Dockerfile for the frontend application

```
FROM node:16-alpine as react-build
WORKDIR /app
ENV PATH /app/node_modules/.bin:$PATH
COPY package.json ./
COPY package-lock.json ./
RUN npm ci --silent
RUN npm install react-scripts -g --silent

COPY . ./
RUN npm run build

# server environment
FROM nginx:alpine
COPY nginx.conf /etc/nginx/conf.d/configfile.template
COPY --from=react-build /app/build /usr/share/nginx/html
ENV PORT 8080
ENV HOST 0.0.0.0
EXPOSE 8080
CMD sh -c "envsubst '\$PORT' < /etc/nginx/conf.d/configfile.template > /etc/nginx/conf.d/default.conf && nginx -g 'daemon off;'"
```

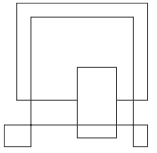
Last a definition for the configurations of the *nginx* server is contained within the project.

Figure 61 - nginx configuration file

```
server {
    listen      $PORT;
    server_name localhost;

    location / {
        root    /usr/share/nginx/html;
        index   index.html index.htm;
        try_files $uri /index.html;
    }

    gzip on;
    gzip_vary on;
    gzip_min_length 10240;
    gzip_proxied expired no-cache no-store private auth;
    gzip_types text/plain text/css text/xml text/javascript application/x-javascript application/xml;
    gzip_disable "MSIE [1-6]\.";
}
```



The pipeline of the Backend Application is a bit simpler, as it is built to a docker image *under-the-hood* which means that the developers have less things to define.

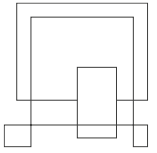
Figure 62 - Backend Application Pipeline

```
steps:

# RESTORE AND PUBLISH ASP.NET 3.1 APPLICATION
- name: 'mcr.microsoft.com/dotnet/sdk:3.1'
  entrypoint: 'dotnet'
  args: [ 'publish', '-c', 'Release' ]
  dir: './'

# DEPLOY APPLICATION
- name: 'gcr.io/cloud-builders/gcloud'
  args: ['app', 'deploy', './bin/Release/netcoreapp3.1/publish/app.yaml', '--
verbosity=debug']
  timeout: '1200s'
```

It includes two main steps – restore and publish the application, and deploy it using *app.yaml* file, that only describes the environment where the application should run.



Testing

Testing is one of the cheapest and easiest ways of making sure that bugs are not promoted into the production. The tests in this project try to cover all parts of the system, starting with code tests – *unit* and *rendering tests* – to the overall system functionality – *system* and *acceptance tests*.

Unit Tests

Unit testing ensures that each *unit* works as intended, in isolation – as opposed to integration tests that focuses on the testing the system as a whole. Even if it can be cumbersome to test all the paths, covering at least the edge cases, where most of the issues occur.

In the Backend Application, to facilitate the testing process a few libraries were used that allow for a better flow in the testing process:

- *xUnit* – open-source testing framework, that organizes test in *Facts* and allows for better integration (xUnit, n.d.);
- Fluent Assertions – a set of .NET extension methods that allow for an organic way of specifying test outcomes (Fluent Assertions, n.d.);
- *Moq* – is a testing framework addon, that allows for mocking behavior of different dependencies to ensure a better way of isolating methods (Moq, n.d.) ;
- *AAA pattern* – even if not a framework, it is a great pattern for better organizing internal structures of a unit test, by splitting it into the three items:
 - o Arrange – prepare what is needed for the unit test;
 - o Act – perform the act that needs to be tested;
 - o Assert – observe what is the outcome (expected vs actual);

Having all the tools ready, now the tests can be presented. In this example, the tests for the *DataController* will be presented. Many times, developers are tempted to skip over *simple* implementations, as those seem not *worthy* for testing which in turn can result in missing bugs. Because of that, in this project, there was effort in testing even simple implementations, to make sure that as few bugs as possible are promoted into production.

First it is needed to setup the test suit, by mocking all the dependencies.

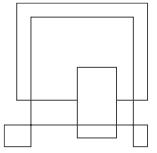


Figure 63 - DataController Unit Testin - Setup

```
public class DataControllerTests
{
    private readonly IServiceProvider _serviceProvider;
    private readonly Mock<IProcessingScheduler> _processingScheduleMock = new
Mock<IProcessingScheduler>();
    private readonly Mock<IStorageManager> _storageManagerMock = new Mock<IStorageManager>();

    public DataControllerTests()
    {
        IServiceCollection services = new ServiceCollection();

        services.AddSingleton(provider => _processingScheduleMock.Object);
        services.AddTransient(provider => _storageManagerMock.Object);

        _serviceProvider = services.BuildServiceProvider();
    }
}
```

As can be seen, the *ServiceCollection* is created and mocks of the needed services are added to it, so that the dependency injection framework can now be used when creating the tested class.

Figure 64 - DataController Unit Test I

```
[Fact]
public void IngestData_WhenMessageEmpty()
{
    // Arrange
    var dataController = new DataController(
        _serviceProvider.GetRequiredService<IProcessingScheduler>(),
        _serviceProvider.GetRequiredService<IStorageManager>());

    var message = new JsonElement();

    // Act
    Action act = () => dataController.IngestData(message);

    // Assert
    act.Should().ThrowExactly<InvalidOperationException>(
        "Operation is not valid due to the current state of the object.");
}
```

Next the AAA pattern is used, where first the *DataController* class is instantiated with the needed dependencies, and a test *JsonElement* message is created. The test message is empty as the first edge case.

Because it is expected the method to throw an error, it is not recommended to call the actual method directly, but make it into an action as a function call. In this way, in the *Assert* section, the action can be used to assert whether the method has thrown with the expected error and error message.

After testing the edge cases, it is also needed to test the normal scenario. In this case an actual *valid* message is provided to the call, and then it is checked if the correct calls were performed during the handling of the method.

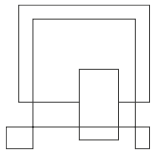


Figure 65 – DataController Unit Test II

```
[Fact]
public void IngestData_WhenMessageCorrect()
{
    // Arrange
    var dataController = new DataController(
        _serviceProvider.GetRequiredService<IProcessingScheduler>(),
        _serviceProvider.GetRequiredService<IStorageManager>());

    JsonElement message =
    JsonDocument.Parse(@"{""Message"":{""Data"":[97,110,111,116,104,101,114,32,108,97,115,116,32,105,109,11
2,111,114,116,97,110,116,32,109,101,115,115,97,103,101],""Attributes"":{},""MessageId"":""{0}"",""Publi
shTime"":{""Seconds"":{1},""Nanos"":{2}},""OrderingKey"":"""",""TextData"":""{""DeviceId"":
\\\"proper_device_id_1\\\"\",\"Consumption\\\": 30}\\\"}\""},\"Subscription\":\"\"\"}).RootElement;

    // Act
    dataController.IngestData(message);

    // Assert
    _processingScheduleMock.Verify(
        mock => mock.ScheduleWork(
            It.Is<Message>((item) =>
                item.MessageId.Equals(\"3240392496288454\") &
                item.PublishTime.Nanos == 417000000 &
                item.PublishTime.Seconds == 1634928943)),
        Times.Once);
}
```

In the example above, first a *valid* message is created using the *JsonElement* instance, and is passed in the *IngestData* method.

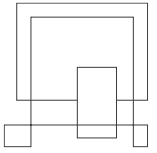
It is known that the *IngestData* method is supposed to de-serialize the message into a *Message* object and passed it into the *ProcessingScheduler* class. Since the *ProcessingScheduler* is not the actual object but the mock that was provided from the service provider, it is possible to verify, what calls were made to the object using what parameters.

Rendering Tests

Rendering tests allows the UI of the system to be tested as *if* a user was to interact with the system but in a more structure and automatized way. The guiding principle is to abstract from a component's implementation and focus the tests to simulate a user's interaction.

In doing so the project utilizes two main libraries:

- *Jest* – is a JavaScript test runner, that offers an interface for creating, running, asserting and structuring the tests (JEST, n.d.);
- *React Testing Library* – is a library that builds on to of *DOM Testing Library* and provides an API for working with *React Components* (Testing Library, n.d.);



To perform the tests first it is needed to import all the required libraries.

Figure 66 - Rendering Tests - importing

```
import React from "react";

import '@testing-library/jest-dom'
import { render, screen } from "@testing-library/react";

import FetchIntervalSelector from './FetchIntervalSelector';
```

First it is imported the *React* library, then the testing library and specific components, and the component to be tested *FetchIntervalSelector*.

Next it is needed to prepare the test, following the AAA testing pattern, it is the *Arrange* phase.

Figure 67 - Rendering Tests - Arrange

```
var menuItems = [
  {
    key: "1",
    text: "Interval 1"
  },
  {
    key: "1",
    text: "Interval 2"
  },
  {
    key: "999",
    text: "Interval Selected"
  }
];

var selectedItem = {
  key: "999",
  text: "Interval Selected"
}
```

At this point, the needed elements for the testing are defined. In this particular test, the needed menu items and the selected menu item.

Last part is to *Act* and *Assert* the test result. In this case, acting is represented by rendering the component into the *DOM*, and asserting is representing by checking if the selected item is rendered in the component.

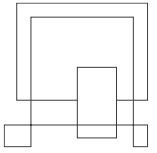


Figure 68 - Rendering Test - Act and Assert

```
describe(
  "<FetchIntervalSelector />",
  () => {
    test(
      "Renders the interval selector component, with the selected item",
      () => {
        //Act
        render(
          <FetchIntervalSelector
            menuItems={menuItems}
            selectedItem={selectedItem}
            callback={() => { }}
          />;

        //Assert
        expect(
          screen
            .getByText(/Interval Selected/i))
            .toBeInTheDocument();
        }
      );
    }
  );
};
```

In addition to the test itself, a few more elements are provided to enrich development process:

- A description of what component is being tested;
- A description of what the test should achieve;

After running the test in the console by using *npm test* command, the results are displayed as follows.

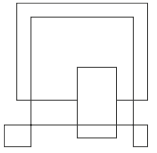
Figure 69 - Rendering Tests - Result

```
 PASS  src/Components/FetchIntervalSelector/FetchIntervalSelector.test.js (7.824 s)
   <FetchIntervalSelector />
     ✓ Renders the interval selector component, with the selected item (85 ms)
```

Usability Tests

Usability tests are a great opportunity to let an end user to interact with the system, and observe how *usable* the system is. It is essential in discovering flaws that cannot be covered by *Unit* and *Integration Tests* as well as spot future opportunities. (Moran, n.d.) In this project's context the usability tests include two participants:

- The tester – an end user that will perform the test;
- The moderator – one of the developers of the system, observing how the user is going through the tasks;



Scenarios

The scenarios were created as *small* stories that include an introduction of the user to the test as well as what he/she is expected to do for this task. The language used in defining the tasks was non-technical and somewhat *interpretable* to simulate as close as possible a real-life scenario. All the scenarios can be found in Appendix V.

Figure 70 - Usability Testing Scenario

Scenario: Check the Online Devices

You have several devices in your house, and you are curious which of the devices are online. In order to do that, you would like to check what devices are registered in the system and which of them are online.

As in the example above, the user is asked to identify the devices that have the current **state** set to **connected**. It is important to note that neither *state* nor *connected* were used in the scenario, to allow the tester to identify it themselves.

Process

The process involved giving the testers the tasks and observe how they are able to perform them. The moderators took note of when the tester tried to access a part of the system that was not part of the task. In those cases, the testers were asked on why they tried to do so, so that more information can be gathered.

The testers were not helped in performing the tasks, in order to maintain the impartiality, only small indication was given (and noted) when a tester forgot about some of the requirements of the task. The testers were given a clean browser session – to ensure that there are no cookies stored – and were asked to read the

Results

The results were quantified by counting how many testers were able to perform the task – this makes the difference between positive and negative results. The neutral result is assigned when the task was achieved with objectively too many steps. Moreover, the distinction between *positive/negative* and *partially positive/negative* is based on the effort that the tester had to make to perform the task.

All results can be found in Appendix V.

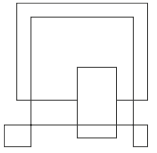


Table 6 - Usability Testing Results

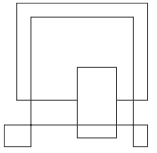
| | | |
|------------------------------------|---|--|
| CHECK PAST DATA | 1 - Positive 3 – Neutral 2 – Partially Negative 1 – Negative | Most users did not realize after initially that they need to click on the bar chart to drill down. One user did not complete the task, two users did not drill down to <i>day</i> scale. |
| COMPARE CONSUMPTION DEVICES | 3 – Positive 4 – Partially Positive | All users completed the task, some users had a hard time initially navigating to the <i>Consumption Reports</i> . When asked why, most of them said that the name does not reflect that. Two users didn't use the <i>current month</i> button. |
| COMPARE CONSUMPTION PERIOD | 7 – Positive | All users completed the task, might be because the users where accustomed from the previous task. |

In the above example extracted from the set of results, it can be seen three variations of results. The first is an overall bad result, only one *positive* result, and six *neutral*, *partially negative* and *negative*. From these results we can conclude that the feature is not very user friendly/usable. In the comments some explanations for the results are included, which is useful feedback to improve how the feature performs.

The second example, presents a result that is overall *Ok*, but there are still some things that can be improved so that the user can easier perform the task.

The last example, is an all-good result, where the users easily understood and found what was needed to be done. But as can be seen from the comments, it is not possible to be very confident in the correctness of the assessment as the *user knowledge* for performing the task, was possibly improved from the previous task.

These results, are great feedback for the development team on what needs to be re-worked and improved. In additional, it would be a good idea to perform focus group testing, that can provide further ways to improve the system.



Acceptance Tests

The Acceptance Tests were conducted with the same tester base as the *Usability Tests* the only different that the focus was not so much on observing their way of interacting with the system. For these tests, the focus was on following the steps described in the test suits as close as possible and filling in the assessment of each step. The tester did not have as much freedom – as they had in the *Usability Tests* – as they had to follow the steps as accurate as possible.

As the tests progressed, the missing parts – Actual and Assessment – were filled in. The *Actual* section describing how the system responds to the users' actions currently, and the *Assessment* contains information about how the *Actual* compares to *Expected*. There are three assessment categories:

- Expected – the system responds to the users' action in the expected way;
- Different – the system responds to the users' action in a different way;
- Missing – the system is missing the functionality, making the user unable to interact with it as described;

Table 7 - User Acceptance Test Result

| ID | Details | Results | | Assessment |
|----|-------------------------------|--------------------------------------|--|------------|
| | | Expected | Actual | |
| 1. | Access Main Control Panel | Display Main Control Panel | Display Main Control Panel | Expected |
| 2. | Access Electricity Monitoring | Display Electricity Monitoring | Display Monitoring Electricity Consumption | Expected |
| 3. | Select devices | Display data for the selected device | Display data for the selected device | Expected |
| 4. | Change refresh rate | Display data for the selected device | Display data for the selected device | Expected |
| 5. | Change interval | Data updates accordingly | Data updates accordingly | Expected |

In the test above it can be seen that in all the steps performed by the *Subscriber* the system responded in the expected way, which means that the system was implemented accordingly to the requirements.

In the test below, in some steps the system did not act as expected, acting differently or having the step not implemented.

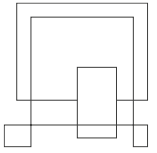


Table 8 - Acceptance Tests Generate Reports

| ID | Details | Results | | Assessment |
|----|------------------------------|--|-------------------------------------|------------|
| | | Expected | Actual | |
| 1. | Access Main Control Panel | Display Main Control Panel | Display Main Control Panel | Expected |
| 2. | Access Reports Panel | Display Reports Panel | Display Consumption Reports | Expected |
| 3. | Select Compare Devices | Display Compare Devices | Display Compare Consumption Devices | Expected |
| 4. | Select devices and periods | Generate report | Generate report | Expected |
| 5. | Select Compare Periods | Display Compare Periods | Display Compare Consumption Periods | Expected |
| 6. | Select devices and periods | Generate report | Generate report | Expected |
| 7. | Select Define Reports | Display Define Reports | Display Under Construction Page | Different |
| 8. | Enter reports configurations | Validate entered information | Not applicable | Missing |
| 9. | Enable reports | Create reports based on configurations | Not applicable | Missing |

The system overall acts as expected with a few of the implementations missing, due to the reasons covered in *Results and Discussions* as well as the *Process Report* found.

The Acceptance Tests offered essential insights into how well the system was implemented from the end-user's perspective. It is crucial to the overall assessment of solution and provided valuable feedback for the development team. Another important advantage is that it was conducted with the end-users interacting with the system, which always uncovers areas that were not captured by the unit and integration tests.

System Tests

The *System Test* allow both developers and the *Product Owner* to understand better to what degree the *promised* solution was created. Using the system tests developed in the *Analysis Chapter* at this stage the overall completion of the solution can be assessed.

Each test is verified and is given an *Assessment* that can be either *Achieved*, *Partially Achieved* or *Not Implemented*. All the tests and overview can be found in Appendix K.

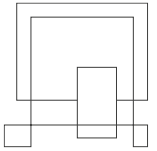


Table 9 - System Test Result

| M As a subscriber, I want to monitor electricity consumption, so that I can see how much electricity a consumer(s) is using. | | |
|--|--|------------|
| ID | System Test | Assessment |
| 1. | The device must be able to measure the electricity consumption | Achieved |
| 2. | The device must be able to collect and format data so it is easy to transmit | Achieved |
| 3. | The device must be able to transmit data to a cloud system at predetermined intervals (collection rate) | Achieved |
| 4. | The system must be able to observe and collect data transmitted by the device | Achieved |
| 5. | The system must be able to store, transform, aggregate and present data to the subscriber | Achieved |

An example of a system test that is not as successfully can be extracted from the *Should* category of requirements.

Table 10 - System Test - Not Successful

| S As a subscriber, I want to receive reports for certain defined intervals, so that I can have an easy-to-read overview of electricity consumption. | | |
|---|---|--------------------|
| ID | System Test | Assessment |
| 1. | The system must be able to use stored data for creating periodical reports | Not Implemented |
| 2. | The system must be able to send/present the reports to the subscriber | Partially Achieved |
| 3. | The system must be able to regenerate reports confidently | Partially Achieved |

As can be seen, in this case not all the sub-requirements were implemented, some being partially implemented. After assessing all the tests, an overview of the results was compiled (found in Appendix K) that includes to what extent each requirement was fulfilled, each requirement was fulfilled and overall weighted average completion was computed.

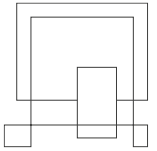


Table 11 - System Tests Results

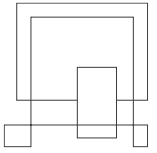
| | |
|--|----------------|
| Must – contribution 50 % | 100 % |
| As a subscriber, I want to monitor electricity consumption, so that I can see how much electricity a consumer(s) is using. | 100 % |
| As a subscriber, I want to be able to access monitoring information from any place with internet access, so that I can monitor electricity consumption remotely. | 100 % |
| As a subscriber, I want to be able to see the current status of the devices, so that I can easily understand which devices are online and which are offline. | 100 % |
| Should – contribution 37.5 % | 76.56 % |
| As a subscriber, I want to receive reports for certain defined intervals, so that I can have an easy-to-read overview of electricity consumption. | 32 % |
| As a subscriber, I want to have a profile that I can login to so that I can have an overview of my devices and configurations. | 100 % |
| Won't – contribution 0 % | 0 % |
| As a subscriber, I want to access the system on a mobile device (with internet access) through a dedicated mobile/pc application so that I don't have to use the browser for that. | 0 % |
| As a subscriber, I want to receive assistance/support from the developers of the system, so that they can fix the system in case of malfunction. | 0 % |
| Weighted Average Completion (excluded Won't) | 78.71 % |

As can be seen the *Must* requirements were completed in proportion of 100 % which is excellent since it proves the viability of the solution and product. The *Should* requirements were achieved in proportion of 76.56 %, which again, is a good result, considering that they are not marked as essential, but rather add-ons. The *Could* and *Won't* categories were not achieved (0 %). For the *Could* category, a higher degree was desirable, but as it is neither essential, nor supportive requirements but rather future additions it does not affect at high degree the overall quality of the product. The *Won't* category was not planned on being implemented.

Each category, had a weight of contribution to the overall degree of achievement for the solution as follows:

- Must – 50 % contribution;
- Should – 37.5 % contribution;
- Could – 12.5 % contribution;
- Won't – 0 % contribution;

Following the above-mentioned contribution, the overall degree of achievement of the solution is 78.71 % which is greater than proposed in the initialization phase (*Project Description – Appendix A.*)



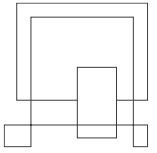
Results and Discussions

In this chapter a closer look at the results of this project will take place. The project will be analyzed from a few perspectives including the requirements, both functional and non-functional fulfillment and the overall state of the system.

First will be presented the current state of the system that is in a working form:

- Hardware - a first iteration of the hardware was designed and assembled circling around the EPS32 and Arduino development boards. The hardware operates as intended; the system is able to register electricity consumption using the ACS712 sensor.
- Embedded system – an embedded system was created to run on the hardware in order to facilitate reading out the raw data, performing the calculations for extracting the electricity consumption, providing the means to connect to the internet via the WIFI interface and send data to the cloud provider. The system is able to do in a consistent way, providing the user with visual feedback via the two (green and blue) LEDs;
- Function – a function was developed to get triggered when new data is registered in the IoT Core via the Pub/Sub system and forward data to the backend. The function is deployed and running and operates as intended;
- Backend and persistent storage – the backend of the solution is ready, deployed and running in production, the system is able to ingest data coming from the device (via de IoT Core and Function) and store it in an SQL database. In additional a No-SQL database is present for storing model data. At the same time the system exposes a REST-full that the front end can interact with;
- Frontend – a user interface via a web page was created to allow the user to interact with the system. The user interface provides means for the user to use the system towards his benefits for monitoring and measuring electricity consumption;
- Security – the system offers a high level of security by utilizing the Google Cloud's tools in all the parts of the system, starting with encrypting the connection and messages from the device and finishing with user authentication;
- Stability – the system presents itself overall stable, it has been running in production, and various user have been interacting with it (or some version of it) for over a month;
- Both the process and project are documented;

Overall, the project resulted in a stable, secure and functioning system, that is deployed in production and accessible over the internet.



Next will be assessed the results based on the requirements. When defining the project scope, a goal of achieving 75 % of the features was set. The goal was set such that it will allow space for improvement if the estimations were pessimistic.

The requirements were categorized using the MoSCoW framework, with each category having a different contribution for calculating the completion of the requirements, as *Must* requirements have a higher contribution as compared to *Should* or *Could*.

After counting and weighting the completed requirements, at this stage the requirements are completed in proportion of 78.71 % which is just a bit higher than the estimated one. The *Must* requirement are completed in proportion of 100% which is crucial for the system.

Similarly, a presentation of the non-functional requirements will follow:

1. This is ensured through using the Google Cloud provider with the services that ensure less than 1% downtime due to their distributed infrastructure;
2. The system's embedded part is developed using *MicroPython* as the main programming language;
3. The system backend is developed using *ASP.NET Core* and *C#* as the programming language;
4. The fronted utilizes *React* as its framework;
5. The system UI has been proved (using *Ant Design* library and *Usability Testin*) to be intuitive and easy to use;
6. The system is modular, by being split up in several services: *embedded*, *function*, *backend*, *frontend* each of them being able to be replaced (or expanded) without affecting the other parts of the system.
7. The main cloud services used for the system – *Cloud Functions*, *Cloud Run* and *Cloud Engine* – are highly focused around being scalable (at a higher cost of maintenance).
8. The system's services overall latency is below the proposed threshold, in cases of a latency higher than the threshold the system will inform the user that the service is unavailable at the moment and restart the service that causes the latency.

The system performance is measured using Google Chrome's profiler that offers valuable insights on how the system performs in production. Various tasks were measured for the performance:

- Loading the page to the Main Control Panel;
- Loading the data for 2 device for the past 12 hours;
- Loading the state of the devices;
- Loading the comparison between 3 devices for one month;

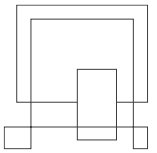
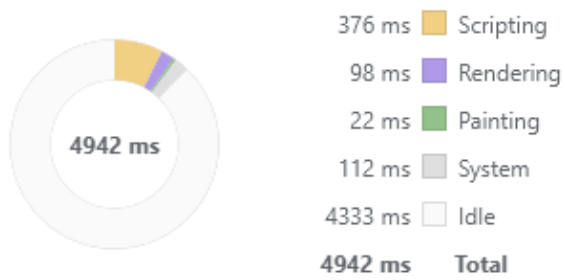
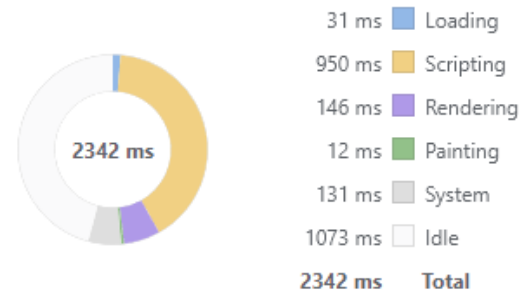


Figure 71 - Performance Measurements

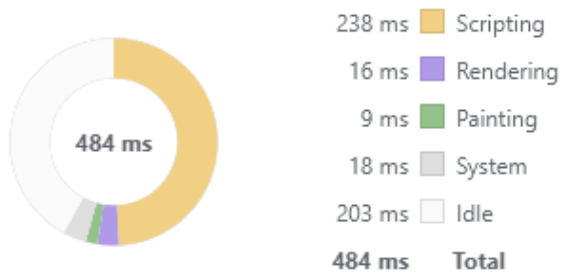
Range: 38 ms – 4.98 s



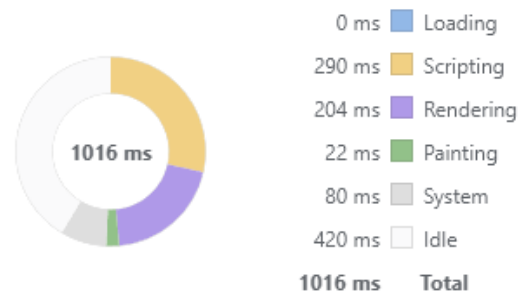
Range: 11 ms – 2.35 s

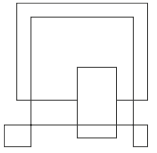


Range: 556 ms – 1.04 s



Range: 311 ms – 1.33 s





Conclusions

All in all, the project has achieved what was proposed to in the *Project Description*. During the project description a set of guiding/helper questions were created to allow an easier follow-up on the progress of the project.

How raw data will be aggregate, transformed and presented to the user so that it provides richer (compared to current solutions) insights about the electricity consumption?

Current solutions, as mentioned in the *Introduction*, mostly focus on provided a *snapshot* view of the data without possibility of drilldown and comparisons. By providing the user with ways of drill down the consumption the user can get insights into how was the data used in the past. In addition it offers ways of comparing consumption over periods or/and devices.

How to ensure security of consumption and user data?

The security of the data is ensured through following provided principles for creating a cloud infrastructure that creates a high level of security. Both code principles and configuration of different services in such a way that does not leave space for malicious acts. In addition, utilizing services that are focused for strengthening the security (*Google Secret Manager* or *Google Cloud SQL Proxy*) allows to ensure an overall high level of security.

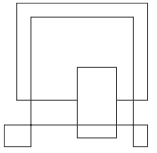
How to ensure high availability (little down-time) of the monitoring system so that the user can explore the consumption at any time without restrictions?

The system is deployed on Google Cloud Platform that ensure a high available with little to no downtime. In addition, the services used to deploy the system, have features as: restarting or rolling back to a working version; multiple instances that allows for easy change of the instance if something goes wrong; hot swap when new version is released it is possible to replace the existing one without having to stop it;

How to configure the device without having to directly program it (network configurations)?

The initial configuration of the device to setup the WIFI credentials, are done through a network and website exposed by the ESP32 devices. When the credentials are missing, instead of trying to connect to a network, the device will create a WIFI network itself, that will allow the user to connect to it via any device – that supports a browser and WIFI – and access the configuration page.

By answering this question, it is possible to demonstrate that the system has solved the problem statement described at the beginning of the project, through a software development process.



Project Future

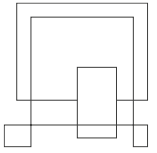
Since the project at the current stage is only a proof of concept and it is built in a modular way, there are various ways of the project to continue and develop.

First of all, there is still space for improvement of the project at the current state without adding any new features. Improving the projects quality, through better documenting the code using, refactoring, increasing the test coverage and system stability. By no means the current project is a final product, and it still have ways it can be improved to offer a higher quality system.

After that, a few areas of focus that were considered even from the beginning of the project but did not make it into the project because of the resource limitations are:

- PDF report generation, based on predefined schedules;
- Adding a relay to the hardware to allow for interrupting the AC current, in that way it will be possible to create schedules and IFTTT rules;
- Adding a more robust self-recovery that will work without the users input
- Adding a bypass switch to turn of the device without interrupting the ac throughflow
- Developing a hardware test server for automated unit tests of the hardware and its software.
- Protecting the AC power connection so that even when the device is forced open the user is not susceptible to touching the high voltage lines;
- Communication with the device to have better control of its configurations;
- Implementing a way of calculating the expenses based on the electricity consumption based on different electricity providers and/or countries, regions, etc;
- Since ESP32 and Arduino are development boards, creating own circuit that will allow for a smaller and more customized device that will offer only what is needed;
- Implementing machine learning into the system – since the project will have access to high amounts of data, it will be very beneficial to implement machine learning algorithms for various purposes:
 - o Predicting consumption;
 - o Analyzing consumption patterns;
 - o Identifying way of electricity waste;
 - o Etc.
- Mobile application and compatibility with home assistants;

All in all, this project offers great possibilities for expansion and building on top of it as it being only a proof of concept.



Source of Information

Agile Business Consortium. (n.d.). *Chapter 10: MoSCoW Prioritisation*. Retrieved from Agile Business Consortium:

https://www.agilebusiness.org/page/ProjectFramework_10_MoSCoWPrioritisation

Alpert, S. (2015, 10 07). *React v0.14 changelog*. Retrieved from React:

<https://reactjs.org/blog/2015/10/07/react-v0.14.html#stateless-functional-components>

Ambler, S. W. (n.d.). *User Stories: An Agile Introduction*. Retrieved from Agile Modeling:

<http://www.agilemodeling.com/artifacts/userStory.htm>

Ant Design. (n.d.). *Ant Design of React*. Retrieved from Ant Design:

<https://ant.design/docs/react/introduce>

Ant Design. (n.d.). *Design Values*. Retrieved from Ant Design: <https://ant.design/docs/spec/values>

Assosiation Franco-Chinoise du developpment urbain durable. (2021). *The History of Smart Homes*.

Retrieved 2021, from <https://www.afcdud.com/fr/smart-city/422-how-the-history-of-smart-homes.html>

Atlassian. (n.d.). *DevOps: Breaking the development operations barrier*. Retrieved from Atlassian:

<https://www.atlassian.com/devops>

Axios. (n.d.). *Getting Started*. Retrieved from axios: <https://axios-http.com/docs/intro>

Carbon Design System. (n.d.). *The Carbon ecosystem*. Retrieved from Carbon Design System:

<https://www.carbondesignsystem.com/all-about-carbon/the-carbon-ecosystem/>

Dharmendra_Kumar. (2019, May 21). *GeeksforGeeks*. Retrieved from Software Engineering | SDLC

V-Model: <https://www.geeksforgeeks.org/software-engineering-sdlc-v-model/>

divyanshu_gupta1. (2021, Jul 05). *Geeksforgeeks*. Retrieved from Software Development Life Cycle

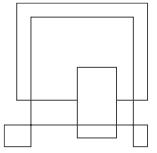
(SDLC): https://www.tutorialspoint.com/sdlc/sdlc_overview.htm

Firebase. (n.d.). *Cloud Firestore*. Retrieved from Firebase Documentation:

<https://firebase.google.com/docs/firestore>

Fluent Assertions. (n.d.). *Introduction*. Retrieved from Fluent Assertions:

<https://fluentassertions.com/introduction>



Formidable. (n.d.). *Victory: Charting for React and React Native*. Retrieved from Victory:
<https://formidable.com/open-source/victory/about>

Google Cloud Platform. (n.d.). *App Engine*. Retrieved from Google Cloud.

Google Cloud Platform. (n.d.). *Cloud Build*. Retrieved from Google Cloud:
<https://cloud.google.com/build#section-6>

Google Cloud Platform. (n.d.). *Cloud Functions*. Retrieved from Google Cloud:
<https://cloud.google.com/functions>

Google Cloud Platform. (n.d.). *Cloud Pub/Sub*. Retrieved from Google Cloud:
<https://cloud.google.com/pubsub/docs>

Google Cloud Platform. (n.d.). *Cloud SQL*. Retrieved from Google Cloud:
https://cloud.google.com/sql/?utm_source=google&utm_medium=cpc&utm_campaign=emea-emea-all-en-dr-bkws-all-all-trial-e-gcp-1010042&utm_content=text-ad-none-any-DEV_c-CRE_169076595312-ADGP_Hybrid%20%7C%20BKWS%20-%20EXA%20%7C%20Txt%20~%20Databases%20~%20Cloud

Google Cloud Platform. (n.d.). *Container Registry*. Retrieved from Google Cloud:
<https://cloud.google.com/container-registry>

Google Cloud Platform. (n.d.). *Google Cloud*. Retrieved from Cloud Run:
<https://cloud.google.com/run>

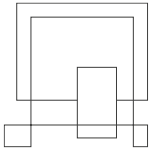
Google Cloud Platform. (n.d.). *Internet of Things*. Retrieved from Google Cloud:
<https://cloud.google.com/iot-core>

Institute for Energy Research. (2021). *History of Electricity*. Retrieved March 2021, from
<https://www.instituteforenergyresearch.org/history-electricity/>

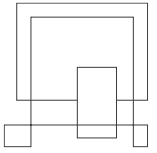
JEST . (n.d.). *Getting Started*. Retrieved from JEST: <https://jestjs.io/docs/getting-started>

Larkin, K., & Anderson, R. (2021, 05 26). *Options pattern in ASP.NET Core*. Retrieved from Microsoft Docs: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options?view=aspnetcore-5.0>

Larkin, K., Smith, S., Addie, S., & Dahler, B. (n.d.). *Dependency injection in ASP.NET Core*. Retrieved from Microsoft: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0>



- Larman, C. (2005). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Addison Wesley Professional.
- lodash. (n.d.). *lodash*. Retrieved from lodash: <https://lodash.com/>
- Material Design. (n.d.). *Design*. Retrieved from Material Design: <https://material.io/design>
- MicroPython. (n.d.). *MicroPython documentation*. Retrieved from MicroPython: <http://docs.micropython.org/en/latest/>
- Mohammed Ghazal, S. I. (2015). *Smart plugs: Perceived usefulness and satisfaction: Evidence from United Arab Emirates*. Research Gate.
- Moq. (n.d.). *Read Me*. Retrieved from github: <https://github.com/moq/moq4>
- Moran, K. (n.d.). *Usability Testing 101*. Retrieved from Nielsen Norman Group: <https://www.nngroup.com/articles/usability-testing-101/>
- MQTT. (n.d.). *FAQ*. Retrieved from MQTT: <https://mqtt.org/faq/>
- Nivo Rocks. (n.d.). *About*. Retrieved from Nivo Rocks: <https://nivo.rocks/about>
- Plotly. (n.d.). *React Plotly.js in JavaScript*. Retrieved from Plotly: <https://plotly.com/javascript/react/>
- postgresql. (n.d.). *About*. Retrieved from postgresql: <https://www.postgresql.org/about/>
- pp_pankaj. (2019, Aug 09). *Geeksforgeeks*. Retrieved from System Testing: <https://www.geeksforgeeks.org/system-testing/>
- pp_pankaj. (2019, Apr 30). *Geeksforgeeks*. Retrieved from Acceptance Testing | Software Testing: <https://softwaretestingfundamentals.com/acceptance-testing/>
- Reactcharts. (n.d.). *API*. Retrieved from Reactcharts: <https://recharts.org/en-US/api>
- reactjs. (n.d.). *React - A JavaScript library for building user interfaces*. Retrieved from reactjs: <https://reactjs.org/>
- repoDB. (n.d.). *Repositories*. Retrieved from repodb: <https://repodb.net/feature/repositories>
- RepoDB. (n.d.). *Welcom to RepoDB*. Retrieved from repodb: <https://repodb.net/>
- Smart Energy International. (2006). *The history of the electricity meter*. Retrieved 2021, from <https://www.smart-energy.com/features-analysis/the-history-of-the-electricity-meter/>



Tanstack. (n.d.). *React Query Introduction*. Retrieved from react query: <https://react-query.tanstack.com/>

Tech Empower. (n.d.). *Round 20 Web Framework Benchmarks*. Retrieved from TechEmpower: <https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=plaintext>

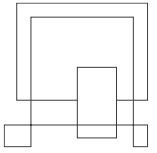
Testing Library. (n.d.). *React Testing Library*. Retrieved from Testing Library: <https://testing-library.com/docs/react-testing-library/intro>

TimescaleDB. (n.d.). *Welcome to the TimescaleDB Documentation*. Retrieved from TimescaleDocs: <https://docs.timescale.com/timescaledb/latest/>

Uber. (n.d.). *Welcome to React-vis*. Retrieved from React-vis: <https://uber.github.io/react-vis/documentation/welcome-to-react-vis>

Wengel, G. (n.d.). *ASP.NET Core IHostedService, BackgroundService and error handling*. Retrieved from Tinkerer: <https://www.gustavwengel.dk/difference-and-error-handling-between-hostedservice-and-backgroundservice>

xUnit. (n.d.). *About xUnit.net*. Retrieved from xUnit.net: <https://xunit.net/>



Appendices

Appendix A – Project Description

Appendix B – User Guide

Appendix C – Admin Guide

Appendix D – Market Research

Appendix E – Use Case Diagram

Appendix F – Use Case Descriptions

Appendix G – Activity Diagrams

Appendix H – System Sequence Diagram

Appendix I – Domain Model

Appendix J – Acceptance Tests and Results

Appendix K – System Tests and Results

Appendix L – Architecture Diagrams

Appendix M – Component Diagram

Appendix N – Data Flow Diagram

Appendix O – CI-CD Diagrams

Appendix P – Class Diagrams

Appendix Q – Flow Diagram

Appendix R – Device Schematics

Appendix S – Sequence Diagrams

Appendix T – Database Diagrams

Appendix U – Source Code

Appendix V – Usability Tests and Results

Appendix W – Draw.io Diagrams File