João Pedro Delgado Lopes

Bachelor of Science in Electrical and Computers Engineering

A Customizable IoT Platform Developed Using Low-Code

Dissertation submitted in partial fulfillment of the requirements for the degree of

Master of Science in
Electrical and Computers Engineering

Adviser:	Filipe de Carvalho Moutinho, Assistant Professor, NOVA University Lisbon

Examination Committee

Chair:	João Miguel Murta Pina
Rapporteur:	Rogério Alexandre Botelho Campos Rebelo
Member:	Filipe de Carvalho Moutinho

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

November, 2020

A Customizable IoT Platform Developed Using Low-Code

This document was created using the (pdf)LaTeX processor, based on the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my grandparents.

“The passion for stretching yourself and sticking to it, even (or especially) when it’s not going well, is the hallmark of the growth mindset. This is the mindset that allows people to thrive during some of the most challenging times in their lives.”

Carol S. Dweck, Mindset: The New Psychology Of Success

Acknowledgements

Thank you to my supervisor, professor Filipe Moutinho, for his consistent support throughout this difficult period of my life. For his neverending availability, mainly via remote calls and even during weekends; for his patience, for the knowledge he shared with me and for understanding my lack of time. I am truly grateful.

I would also like to thank Noesis for providing me with the technical skills required to do this thesis and for giving me all of the conditions to finish it. A special appreciation to Inês Brilhante, Inês Mariano, Margarida Morais, Miguel Silva and Tiago Fernandes for advising me; for the great leadership and for making me happy to go to work every day.

My sincere gratitude to Luís Santos Monteiro and José Pedro Proença for helping me in so many ways: with technical and personal advices; for reviewing my dissertation, providing corrections and improvements and for helping in the maintenance of an Outsystems component that will be further mentioned in this thesis.

I am specially grateful to José Pedro Proença and José Silva Pinto for the infinite support; for sharing their growt-mindsets with me; for never giving up and never letting me give up; for all the things I learned from them; for all of the laughs and for encouraging me to believe in myself and to finish my thesis and for saving time to read and review my dissertation, thank you!

To Ayala, Duda, Zé, Ricardo and Tiago, thank you for all the laughs, beers and adventures, for the great friendship and for not letting me give up.

A note of appreciation to my family, namely my parents and my sister, for all of the efforts they made for me to finish my masters. Without them, this would not have been possible. I would also like to remember my grandparents. They will always be carried in my heart. My maternal grandfather said that I would be an engineer since I was a little kid. This is for him!

Last but not least, I would like to thank the most important person of my life, Inês Vitória Gemas Jorge. For being my confident in all occasions; for always being available to calm me down; for being my daily motivation; for never letting me give up; for the good advices; for never letting me feel alone; for understanding me; for making my life brighter and making me look forward to a beautiful future. For all of this and much more, thank you! I would also like to thank João, Laurinda and Pedro, for making me feel at home, caring about me and for giving me important advices.

Abstract

In nowadays' societies and businesses' ecosystems of acceleration, it is of most importance the digitalization of the formers. In this sense, Internet of Things (IoT) emerges to connect devices to the internet, allowing the access to large amounts of data and, through its analysis, to act upon it.

An IoT platform allows, among others, to manage IoT devices. Nonetheless, its development is still a complex and expensive process that requires high technical knowledge. To accelerate developments when building IoT applications, this thesis proposes the use of Low-Code platforms, such as Outsystems.

To improve the user experience, an embedded customizable dashboard is included in the platform. To achieve this, a reusable Forge component was built and is currently available for download and in use by the community. This component allows to provide drag and drop functionalities to both web and mobile applications. In this case, it will allow the reordering of dashboard cards to personalize dashboards. Another component that enables barcode/qr code reading by Zebra devices is also included in this thesis.

To assess the developed IoT platform, an IoT device prototype was created, using an Arduino with an ESP module, as well as a set of sensors and actuators that communicate with the IoT platform via the Message Queuing Telemetry Transport (MQTT) protocol.

Keywords: Internet of Things; IoT; Low-Code; Dashboard; Platform; Arduino; Outsystems; Web; Mobile; Application; MQTT; Forge; SortableJS; Customization; Dashboard.

Resumo

No ecossistema de aceleração em que a sociedade e as indústrias se encontram actualmente, é da maior importância a rápida digitalização das mesmas. O conceito de *Internet of Things* (IoT) surge então para conectar uma grande diversidade de dispositivos à internet, possibilitando o acesso a uma grande quantidade de dados e, aquando da análise dos mesmos, optimizar um conjunto de operações.

Plataformas IoT são ferramentas bastante úteis em projectos IoT. Estas permitem, entre outros, a gestão de dispositivos IoT, no entanto, o seu desenvolvimento é um processo complexo e demorado que requer elevados orçamentos e conhecimentos técnicos. Para acelerar o desenvolvimento deste tipo de plataformas, esta dissertação propõe o uso plataformas de desenvolvimento *Low-Code*, nomeadamente Outsystems.

Para melhorar a experiência de utilizador, neste trabalho é proposta uma plataforma IoT com *dashboards* personalizáveis, utilizando uma arquitetura modular. Para tal, foi desenvolvido um componente reutilizável, publicado na *Forge* e que está atualmente disponível e a ser utilizado pela comunidade. O mesmo permite adicionar a funcionalidade de *drag and drop* em aplicações *web* e *mobile*. Neste caso, o componente permite arrastar cartões, de forma a personalizar *dashboards*. Um outro componente que permite a leitura de códigos qr/de barras foi também desenvolvido e encontra-se disponível para download.

Para validar a plataforma desenvolvida, foi criado um protótipo de um dispositivo IoT, utilizando um *Arduino* com um módulo ESP e um conjunto de sensores e atuadores, que comunica com a plataforma através do protocolo MQTT.

Palavras-chave: Internet das Coisas; IoT; *Low-Code*; Platforma; Arduino; Outsystems; *Web*; *Mobile*; Aplicação; MQTT; *Forge*; SortableJS; Customização; *Dashboard*.

Contents

List of Figures

List of Tables

Listings

Glossary

Aggregate	From the Outsystems documentation: tool to "fetch data using an optimized query. Aggregates can load data from the server of the local database, and they support combining several Entities and advanced filtering."
Automatic Activity	Defined in the Outsystems documentation as "(...) work to be carried out automatically in your process (...) without the need for human intervention. (...) can have output parameters and local variables.".
Exception	Defined by Outsystems as "an exceptional circumstance that prevents your application flow from running normally".
IoT	Acronym for Internet of Things. Term that contains all devices/objects connected to the internet. With a network of these objects, it is possible to collect high amounts of data, analyse it and act upon it.
Preparation	Defined by Outsystems as an action containing the logic that is executed before the screen or Web Block (WB) renders.
Timer	From the Outsystems documentation: "(...) allows executing application logic periodically on a scheduled time" and are known as batch processes. Multiple Timers can run at the same time, in parallel, but each timer can only have one scheduled execution time at a time.
Transducer	Term that describes both sensors and actuators.
Web Block	Defined by Outsystems as a reusable screen part that can implement its own logic. It defines not only reusable User Interface (UI), but also the underlying logic.

Acronyms

4LC	4-Layer Canvas
AD	Analog-Digital
AJAX	Asynchronous Javascript and XML
AP	Access Point
API	Application Programming Interface
B2B	Business-to-Business
BL	Business Logic
BLE	Bluetooth Low Energy
BPT	Business Process Technology
CoAP	Constrained Application Protocol
CPS	Cyber Physical Systems
CRUD	Create, Read, Update and Delete
CS	Core Services
CSS	Cascade Style Sheets
DB	Database

DnD	Drag and Drop
DS-Pnet	Dataflow, Signals and Petri Nets
FK	Foreign Key
GSM	Global System for Mobile Communications
HMI	Human-Machine Interface
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IP	Internet Protocol
JSON	JavaScript Object Notation
LBPT	Light Business Process Technology
LCDP	Low-Code Development Platform
LCS	Low-Code Solution
LDR	Light Dependent Resistor
LED	Light-Emitting Diode
LoRa	Long Range
LoRaWAN	LoRa Wide Area Network
LPWAN	Low Power Wide-Area Network
M2M	Machine to Machine

MCS	Mobile Core Services
MQTT	Message Queuing Telemetry Transport
OS	Outsystems
PA	Precision Agriculture
PaaS	Platform-as-a-Service
PK	Primary Key
POC	Proof of Concept
QOS	Quality of Service
REST	Representational State Transfer
RF	Radio Frequency
RGB	Red-Green-Blue
SaaS	Software-as-a-Service
SOAP	Simple Object Access Protocol
SotA	State of the Art
SQL	Structured Query Language
SSID	Service Set Identifier
UHF	Ultra High Frequency
UI	User Interface
US	User Story

UX	User Experience
WB	Web Block
XML	Extensible Markup Language

*

Chapter 1 Introduction

The XXI century can be described as an "(...) ecosystem of acceleration", with an "(...) ever-increasing pace by digital technology that is transforming the way we live and work" [1]. Currently, in the midst of a pandemic and resulting economic crisis, the ability to change, either from an individual, or from a business perspective is even more demanding [2]. According to [1], it is important to reinvent the way to work, by using technology in our favour, relying on the use of applications running on the cloud. IoT can be one of the ways technology is used to aid in this reactive/fast-changing world.

To help understand the meaning of IoT, Madakam et al. [3] divided it into two: "Internet" and "Things". The first "is a global system of interconnected computer networks that use the standard (...) (TCP/IP) to serve (...) users worldwide", whereas the second includes, not only electronic devices, but everyday objects as well. In other words, these "things" are "real objects in this physical or material world". Furthermore, Madakam et al. ends up defining IoT as: "an open and comprehensive network of intelligent objects that have the capacity to auto-organize, share information, data and resources reacting and acting in face of situations and changes in the environment".

IoT represents the future of computation and communication, as it is estimated that there will be over 21 billion IoT devices in 2025, or over 30 billion devices in 2023, according to [4] and [5], respectively. Other than the wide increase in expected devices over the next years, the range of IoT applications is also vast: smart agriculture; autonomous and connected vehicles; smart watches, fitness trackers and wearables; industry and supply chains; and cities-management (energy, transportation, parking...) [5].

Summing up, there is the need to cope with change; the reactivity to change fast; the relying on applications running on the cloud and IoT. But how do these concepts relate? Well, as IoT consists of objects connected to the internet, it is responsible for the increase in productivity in companies and peoples' lives and, consequently, to keep up with the fast-pace requirements nowadays. In IoT, the need to connect devices to the internet is not, itself, of much help. It is also required to handle those objects, as well as its data and use this information wisely. For that, cloud-based applications can be used. So far nothing new, but what if one could improve on these concepts?

1.1 Motivation

One way one could accelerate the reactiveness to perform changes and to change rapidly is to use a Low-Code platform. In [6], Gartner estimates that "by 2024, Low-Code application development will be responsible for more than 65% of application development activity.". As someone who has been working with a Low-Code platform named Outsystems[1] in big-scale projects for almost two years, I think that these kind of platforms are incredibly efficient; decrease the time-to-market; are less prone to errors; ease the code maintenance, development, deployment and the control of project versions; etc.

With the capability to build cloud-based applications in a novel, fast way, Outsystems, with the new coding paradigm comes in handy in the aforementioned "ecosystem of acceleration". It allows to build applications faster than ever before and to react quickly to changes/new requirements. Additionally, the platform optimizes code (*e.g.* in selecting attributes to select in Structured Query Language (SQL) queries).

Moreover, one can also develop applications (namely, an IoT platform) in a fast way, while maintaining efficiency. The development efficiency can be further increased with a modular architecture, separating the application's concepts and functionalities in different modules. Moreover, every developer can build and publish Outsystems components, which can later be used by the development community.

Other than that, given the multitude of IoT appliances, developing a different application for each one might be expensive and time consuming. Instead, one could build a web and mobile IoT platform as generic as possible, so that it is compatible with different scenarios, different devices, different protocols, among others. This application, developed using the Low-Code paradigm, has to be user friendly and highly customizable, so that different users with different use cases, could profit from using the application.

1.2 Proposal

Before diving into the proposal, *per-se*, it is important to distinguish two concepts: platform and dashboard. A dashboard, in the scope of this thesis, is defined as an interface (web and mobile), that allows to gather, organise and present information in a way that is easy to read. The definition from Gartner, [7], is the following: "Dashboards help improve decision making by revealing and communicating in-context insight (...) using intuitive visualization, including charts, dials, gauges and "traffic lights" that indicate

[1] https://www.outsystems.com/

the progress (...) toward defined targets". Its main component is therefore the Human-Machine Interface (HMI), with the function to assemble and depict data from the physical world into a easy to understand exhibit [8]. A platform, on the other hand, is a broader concept and can contain a dashboard.

This thesis proposes the development of a platform (to manage users, devices, locations and dashboards). The dashboards are fully customizable in order to provide the data that interests the most. If a user is in charge of a small part of a large system, he can opt to only show the sensor's data or actuators' controllers regarding that "part" and can sort the information from the most important/relevant to the least. He can therefore optimize the dashboard to his preferences and to increase efficiency. In addition, users can create and select multiple dashboards and each dashboard is related to a location. As an example, if one is monitoring aquariums, then a location can be called "Aquarium". On the other hand, if the system being monitored is a factory, then a location can be called "Station".

Finally, other than showcasing the versatility of Low-Code platforms and that it is possible to develop complex and highly efficient applications managing high volumes of data and with a fast pace development, it is also important that this IoT application can be learnt with low effort.

The aforementioned an IoT platform will be developed with the aid of the Outsystems Low-Code platform and with a modular architecture. Finally, some contributions were made to the Outsystems community, not only with this dissertation, but also by publishing an article and some Outsystems components with the main one being a React Drag-And-Drop component. This component was used in the embedded dashboard so that the user could configure it (*eg.* changing the order of plots) to his own taste while improving the User Experience (UX) simultaneously.

Furthermore, a prototype circuit will be designed to showcase a real-life scenario and, to make sure it can communicate successfully with the platform, an IoT communication protocol will be used: MQTT.

1.3 Dissertation Structure

The dissertation structure is as follows:

- In **Chapter 2**, the literature review takes place, namely, regarding the IoT protocols, the low-code development platforms and the overview of Outsystems;
- In **Chapter 3**, the architecture will be thought and designed, ending with the used modules;
- In **Chapter 4**, the web and mobile developments will be presented;
- **Chapter 5** shows the developed forge components, namely, SortableJS;

- **Chapter 6** depicts the hardware prototype that was built in order to have transducers data and test/validate the results;
- In **Chapter 7**, the developed platform and this thesis' results are analysed;
- In **Chapter 8**, the conclusions and future work take place;
- **Appendix A**, **Appendix B** and **Appendix C** depict the mobile plugin developments, data bootstrap to generate data and the Arduino code, respectively.

Chapter 2

Literature Review

The focus of this dissertation's State of the Art (SotA) is as follows: in Section 2.1, the study of different IoT platforms available takes place, together with their advantages and disadvantages; moreover, although it's not the focus of this thesis, IoT protocols are reviewed (Section 2.2); Section 2.3 contains the literature review regarding the concept of a Low-Code Development Platform (LCDP) and in **Section 2.4**, the LCDP chosen for this thesis, Outsystems, is studied.

2.1 IoT Platforms

The study of the commercial IoT platforms already available in the market is important, not only to analyse the characteristics of those solutions, but also to evaluate if certain features should be included in this thesis' solution. Just to remind, this solution is an IoT platform to collect data from multiple sources; store that data and manage devices. Also, this platform will allow to create and edit dashboards, as well as it will be build using Low-Code and with a well thought modular architecture.

The first platform that is analysed is Thingsboard[1]. Its first advantage is that it is compatible with industry standard IoT protocols, such as MQTT and Constrained Application Protocol (CoAP) and supports both cloud and on premises deployments [9]. These protocols will be discussed in Section 2.2. It also allows to customize the configuration of a dashboard and to attribute the dashboard to specific users, thus, it has a role management integrated. Additionally, it can be integrated with Node-RED, which will be discussed further in this section. Finally, the platform allows to create and manage alarms, which is an important feature to alert users about certain thresholds that were surpassed, or even trigger some actuators. Additionally, it is open source and adapts to

[1] https://thingsboard.io/

mobile devices [10]. As drawbacks, some important integrations are in the paid version, such as IBM Watson and AWS IoT; most of the customizations are also paid; has a small community and some features are still under development [10].

Another platform available in the market is Ubidots[2]. As in the previous solution, this platform eases the collection of data and turns it into perceptible information. It allows to connect to hardware using Hypertext Transfer Protocol (HTTP), MQTT and it is possible to use protocols such as SigFox or Long Range (LoRa). Additionally, it is possible to improve the data analytics using Application Programming Interface (API)s such as the IBM Watson or Google Locations; it stores an history of data; it is possible to create dashboards using a point-and-click interface, where several widgets can be added, or even custom HTML code; has an embedded users and roles management interface and the platform can suit the businesses/users' needs by changing parameters such as colours, labels and images.

A rather different tool is Node-RED[3]. Similarly as Thingsboard and Ubidots, it's an MQTT Dashboard, meaning that it allows both control and display of data from MQTT devices. It allows to develop control logic through nodes, instead of the conventional lines of code. Several nodes can be connected to each other and messages (topics and content) flow through these nodes. It is also possible to embed Javascript in logic nodes. Figure 2.1, shows an example of these nodes where, in this case, a message with the temperature data from a sensor is received and several actions are performed. Moreover, the `node-red-dashboard` allows to add tabs and boards (create and edit dashboards) in a modular way. These dashboards adapt to mobile devices as well. Figure 2.2 depicts a dashboard developed for a Precision Agriculture (PA) system, where several plots; gauge displays; and personalized text showing data from several sensors can be seen. It is also possible to add custom HTML and CSS to a dashboard. Both Figure 2.1 and Figure 2.2 are a result of an academic project that took place at FCT/UNL. Node-RED also contains a Library, similar to Forge in Outsystems (see Section 2.3), where different users can create share flows.

[2]https://ubidots.com
[3]https://nodered.org/

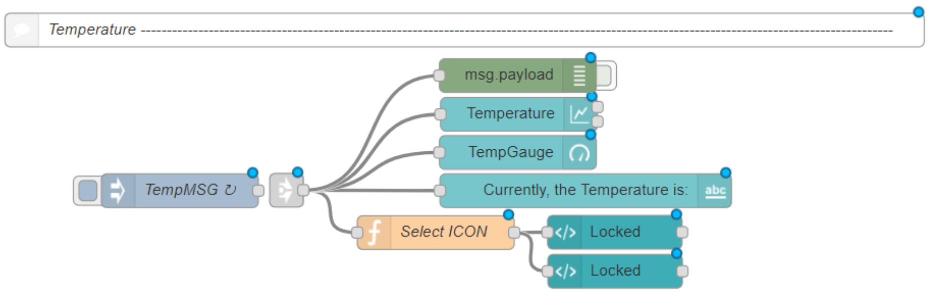
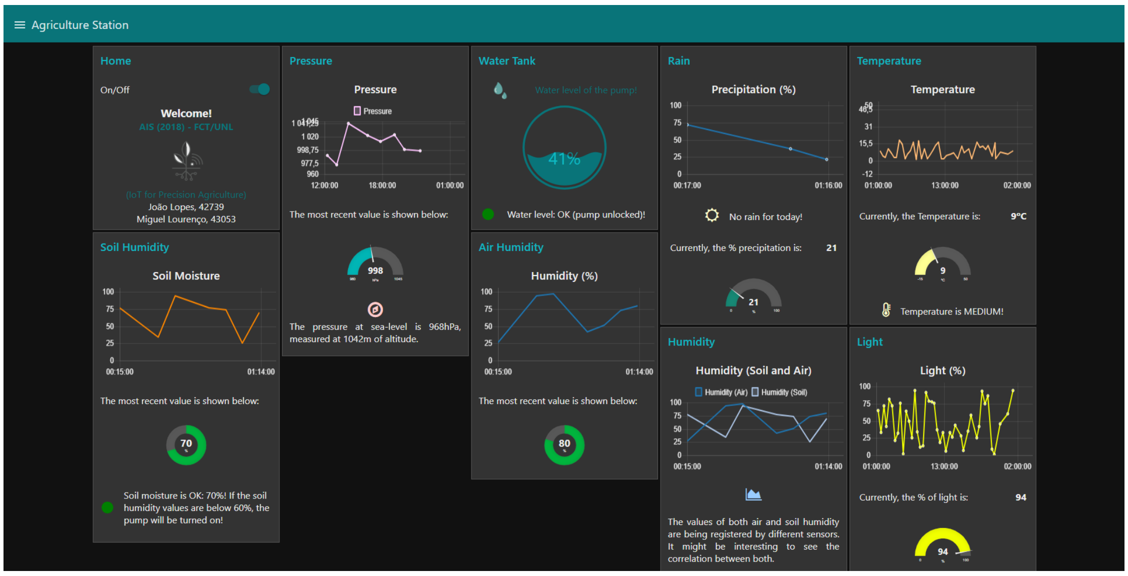


Figure 2.1: Collecting temperature values from an Arduino using Node-RED for Precision Agriculture.



The most recent value is shown below:

The pressure at sea-level is 968hPa, measured at 1042m of altitude.

Water level: OK (pump unlocked)!

No rain for today!

Currently, the % precipitation is: 21

Currently, the Temperature is: 9°C

Temperature is MEDIUM!

The most recent value is shown below:

Soil moisture is OK: 70%! If the soil humidity values are below 60%, the pump will be turned on!

The most recent value is shown below:

The values of both air and soil humidity are being registered by different sensors. It might be interesting to see the correlation between both.

Currently, the % of light is: 94

Figure 2.2: Agriculture Dashboard implemented in Node-RED.

Plenty more solutions could have been mentioned, such as Kaa[4]; freeboard[5]; ThingSpeak[6]; thethings.io[7]; thinger.io[8]; IoTtweet[9]; and IOT HOOK[10]. All of these platforms are very similar, although some have some interesting features such as artificial intelligence and machine learning integrations. Some of the platform characteristics that were analysed *a-priori* will be included in this thesis' platform. These include the creation and personalization of multiple dashboards, that can be available to different users and different usages; a screen to manage devices; the possibilty to personalize the overall look of the platform with characteristics such as colors; the possibility to filter each sensor and actuator by location (*e.g.* one could be monitoring two different agriculture fields, or locations); the possibility to create and manage users; among others. It will be seen in the Section 2.3 that the platform will be built in a way that allows faster developments, improvements and deployment of features.

2.2 IoT Protocols

The first thing that is important to evaluate when building any IoT solution is the multiple protocols that exist. For the purpose of this thesis, only three types of protocols will be evaluated. These are: Communication/Transport; Data Protocols and Semantic.

[4]https://www.kaaproject.org/
[5]https://freeboard.io/
[6]https://thingspeak.com/
[7]https://thethings.io/
[8]https://thinger.io/
[9]https://www.iottweet.com/
[10]https://iothook.com/en/

2.2.1 Communication/Transport Protocols

Starting with the Communication/Transport protocols, the technologies can be devided into 3 types:

- Short-range wireless communications (such as WiFi, Bluetooth, Bluetooth Low Energy (BLE) and ZigBee);
- Cellular communications (2G/3G/4G/5G);
- Low Power Wide-Area Network (LPWAN) communications (e.g. LoRa and SigFox).

A summary of the most known wireless technologies is in Table 2.1, where the subGHz frequencies are around 800 – 900 MHz or 400 MHz. Figure 2.3 also helps organizing and comparing these protocols.

Table 2.1: Communication/Transport Protocols' characteristics, based on [11].

Technology	Frequency	Data Rate	Range	Power Usage	Cost
WiFi	subGHz; 2.4 GHz; 5 GHz	0.1 – 54 Mbps	< 100 m	Medium	Low
Bluetooth	2.4 GHz	1, 2, 3 Mbps	∼ 10 m	Low	Low
ZigBee	subGHz	< 50 kbps	∼ 100 m	Low	Medium
2G/3G/4G/5G	Cellular Bands	10 – 100 Gbps	Several Kilometers	High	High
LoRa	2.4 GHz	250 kbps	2 – 15 km	Low	Medium
SigFox	subGHz	< 1 kbps	Several Kilometers	Low	Medium

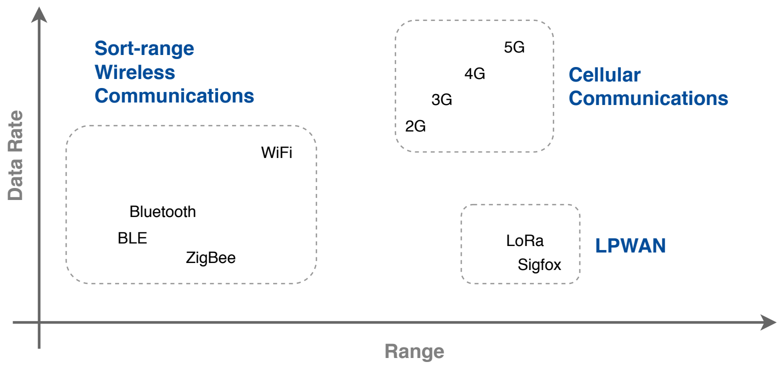


Figure 2.3: Communication Protocols: Range vs. Data Rate Comparison, adapted from [11].

In regard to short range communications, WiFi uses radio waves, Radio Frequency (RF), for multiple devices to communicate with each other. Usually, an internet router is used to communicate with devices such as computers and laptops, but multiple devices can interact directly without the need of a gateway [12]. As sending data at a higher rate will impact on the power consumption and will not have a long range, WiFi might is not the best option for IoT in some cases as IoT devices may not need to send high amounts of data, but just a few bytes instead and the information needs to be sent over high distances [13]. In addition, the power requirement might be an issue, as IoT devices often need to operate on batteries for long periods of time [13].

An alternative is Bluetooth. Compared to WiFi, the transmitted signal can only be received at shorter distances. This protocol is often used in small devices that connect to smartphones or computers. It uses Ultra High Frequency (UHF) radio waves to transmit data [12]. Additionally, BLE does not compromise the device's battery as much as Bluetooth. On the other hand, achieves a slightly lower data rate and range. This standard is rather recent (from 2006), and is used in devices such as fitness trackers and smart watches [12].

The IEEE 802.15.4, ZigBee, designed to be used in Machine to Machine (M2M) networks [12]. Its characteristics allow to maximize a device's battery and the protocol also offers encryption [12]. It enables to connect multiple devices to each other with the so called mesh network, where several nodes behave as a single network and each client can connect directly to any of the other nodes. Mesh networks allow IoT applications to cover more ground and therefore, gathering more data. In IoT there are three types of nodes that may exist within a mesh network. These are a Gateway, which are devices that are connected to devices not in the mesh, to pass messages; a Repeater, that are devices that transmit messages between nodes; and Nodes, that are mesh devices that transmit messages [14].

Moving on to the Cellular Communications, or Global System for Mobile Communications (GSM), all protocols, such as 2G/3G/4G/5G, have very high ranges and data rates and, the higher the generation, the higher these two parameters. The biggest drawback, however, is the power consumption. The latter may present as a big disadvantage in IoT applications where devices are not connected to an electricity source [15]. In GSM, a signal is sent to nearby cell towers, which then send signals back down [15]. These cell towers are connected to wired ground stations that help to transmit the signal at greater distances [15]. One drawback is that GSM does not operate on open frequencies, as each cellular carrier operates at his own frequency and one must pay to use this frequency. Whether or not it is a suitable option for IoT, it all depends on the use cases, as seen above.

LPWAN protocols are meant to send small data packages over long ranges taking the battery into consideration, by using RF. One of these protocols is LoRa. In contrast with GSM, LoRa operates on an open frequency, so it does not require a license. The disadvantage is the resultant low data rate and possible interferences. Moreover, the range of the LoRa technology largely depends on the type of environment. The main idea behind LoRa is that multiple sensors, with a lifespan of around 2 years on battery transmit data to multiple gateways, which are connected to the internet [16]. There are 2 main components of a LoRa network: nodes (transducers - slave devices) and gateways (routers, receiving data from nodes and send it over an Internet Protocol (IP) network).

Finally, Sigfox is defined in [17] as a cellular network for things. It is a "cellular style, long range, low power, low data rate form of wireless communications (...) for devices like remote sensors, actuators and other M2M and IoT devices" [18]. It solves some of the problems with GSM appliances to IoT, such as the fact that it is focused on voice and high data rates; are not suited to low data rates due to the complexity of the radio interface; high cost; and high power consumption [18]. With these characteristics, the Sigfox is intended to be applied in low cost M2M where large areas are needed to be covered. Its functioning is as follows, a Sigfox module (or node) sends messages, which are automatically authenticated using a private key specific for each device. Then, a Sigfox base station (connected to the internet) listens to these messages, receives and forwards them to the Sigfox cloud which then redirects the message to a dedicated cloud using the HTTP Protocol, for instance. Moreover, it uses unlicensed radio bands and the number of messages sent per day is limited. According to the selected plan, this limit may differ.

To conclude, there is no perfect protocol as its choice always depends on the application. If smart-home devices are to be used, then perhaps, WiFi is the suitable option; on the other hand, for wearables, Bluetooth; and for applications such as autonomous driving, GSM technologies are the most suited.

The next section will present the different data protocols and a comparison between them will be made.

2.2.2 Data Protocols

In this section, some of the most well known data protocols for IoT will be presented, namely: HTTP; websockets; MQTT; and CoAP.

Starting with HTTP, it is often used by devices that need to send a lot of data [19]. With a request/response or client/server architecture running via TCP or UDP, it uses a normal IP header to transmit packets [19]. Data is not encrypted before transmission (has no embedded data security protocols) and it can only send data to one client at a time and [19].

On the other hand, websockets act as a handshake between browsers and servers. The bi-directional communication in websockets is more suited for monitoring systems, comparing to the request/response type of communication used in the HTTP protocol [20].

However, with the newest version of HTTP, HTTP2, which now includes bi-directional transactions, websockets usage is likely to diminish [20]. It allows TCP multiplexing to perform several requests, as well as to prioritize requests and header compression.

Moving on to MQTT[11], it is a M2M (as a client communicates with a server) protocol [21] that can be used by sensors and small devices, being a suitable solution for IoT devices. MQTT devices and applications communicate through a message broker running

[11]http://mqtt.org/

on a server and all communication routes are centralized (publish and subscribe mechanism) in this server [21]. For the MQTT broker, Node-RED (mentioned in Section 2.3) can be used. In addition, the protocol is designed for TCP/IP networks [22]. Furthermore, several authentication and data security mechanisms are supported [22]. These are defined on the broker and the client must follow them [22].

By taking into account bandwidth, battery and the versatility of features, MQTT is superior to HTTP and websockets, since neither were designed with M2M in mind [20].

Next, CoAP[12] is also a M2M protocol. It uses Representational State Transfer (REST), it can use either IP/UDP or UDP to save some bandwidth and it takes care of security [23]. CoAP intends to emulate the REST features of HTTP, but in a more accessible way [24]. Extending its definition, CoAP is a software protocol specially useful for low power sensors, switches, valves and similar devices that need to be supervised or controlled at a distance [24].

At this point, REST was mentioned. It is an API, which simply means that it allows two applications to talk to each other. Its functioning is simple: the client makes a call to the server, which then replies, both over HTTP. In conclusion, REST is an architecture for the design of networked applications and is resource-based, which indicates that every server object can be created and destroyed. It uses HTTP verbs to indicate actions. Also, it is important to say that it is a stateless web service, which is translated on the fact that every communication (request and response) is independent and transmitted/received at once.

As MQTT and CoAP are both suitable for IoT, the Table 2.2 and Figure 2.4 present a comparison between both. The most import aspect to notice is that, whereas MQTT uses a publish/subscriber paradigm, CoAP uses a request/response one [25]. Moreover, MQTT uses a centralized solution to receive and send messages from/to the clients, whereas CoAP is a one-to-one protocol, similar to HTTP [25]. Finally, while MQTT is an event-oriented protocol, CoAP is more used for state transfer [25].

Table 2.2: CoAP vs. MQTT Data Protocols

	MQTT	**CoAP**
Base Protocol	TCP	UDP
Communication Node	M:N	1:1
Power Consumption	Higher than CoAP	Lower than MQTT
Paradigm	Pub/Sub Model	Req/Response Model (RESTful)

2.2.3 Semantic Protocols

In this section, the most common semantic protocols for Web: JavaScript Object Notation (JSON) and Extensible Markup Language (XML) [26] are compared. The Table 2.3 sums up the differences between these two protocols.

[12] https://coap.technology/

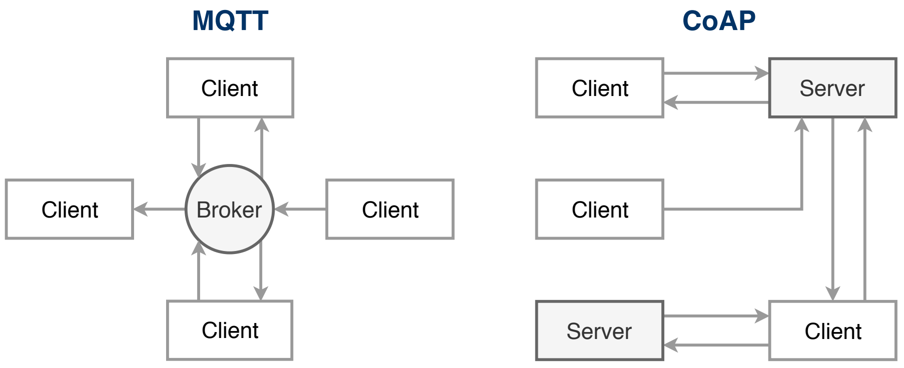


Figure 2.4: MQTT vs CoAP protocols.

Table 2.3: Comparison between JSON and XML, based on [26].

	JSON	XML
Applicability	Allows to transmit data in a parseable manner.	Data in a structure manner. Can be used to save metadata. Parse the scripts.
Popularity Reason	Less verbose and faster.	Uses more words to describe the intention. Parsing it is slow and consumes a more memory.
Data Structure	Map, similar to key/value pairs Useful when interpretation and predictability is required.	Tree representation of data. More time consuming when working with it.
Data Information	Preferred for data transmition between servers and browsers.	Preferred for storing information on the server.
Metadata Tagging	Not allowed directly.	Simple with the use of attributes.

While JSON is “an open-standard file format that is used for browser-server communications (...)” (a data interchange protocol) and a “language-independent data format.”, XML is a “set of rules that help the users to encode documents in a human-readable format and machine-readable.” (markup language) [26]. Moreover, JSON is easier to comprehend and learn (is data oriented) versus the document oriented XML; glsJSON is less structured; JSON only supports text and number data types and XML supports images, charts, among others [26]. Another JSON advantage (in this thesis) is that the Outsystems platform (Section 2.3) has built-in mechanisms to parse data from JSON to structures and vice-versa, which eases the implementation time.

2.3 Low-Code Development Platforms

According to [27], LCDPs are “visual-based, integrated development environments (IDEs) comprising many of the same tools and functionalities developers and IT teams use separately to design, code, deploy, and manage an application portfolio". These type of solutions allow to develop both fast and with minimum hand-code [28]. LCDP may [28]:

- A visual IDE (for defining the user interface; workflows; data models and hand-written code compatibility;
- Allow to connect various back-ends/services, handling structures, storage and retrieval of data.
- Have an application lifecycle manager, with tools to build, debug, deploy and maintain the application in test, staging and production.

The origin of this concept relates to the rapid progression of the software development world and businesses/societies in general. To react to these changes, called Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS) can be used. Whereas the first is a software that can only be accessed via browser, the second is one level below [29] and provides an online platform to develop, manage and run applications [29].

Moreover, Hendriks another concept is related to these platforms: model-based platform. A model-based platform uses models as the center of application. Segments are then drag and dropped to to be linked together and create an application [29]. Often, these platforms are LCDP [29]. The platform will finally generate code based on the model that the user created [29].

One widely known LCDP is Outsytems, a PaaS that provides development and deployment environments for applications from the cloud [30]. The benefits of it are: development speed, as "Reusable components, visual IDEs, abstraction and automation streamline application development, provisioning and deployment" can be used [30]; scalability, since everything is using the cloud (big advantage over on-premises solutions), among others.

2.4 Outsystems Overview

The aim of this section is to give the reader an overview of the Outsystems platform, as it will be used to develop the web and mobile IoT applications. As a Low-Code Solution (LCS), LCDP or model-based platform, it aims to aid with projects were the complexity involved is high [31], enabling developers to start with the high level design model, including the system behaviour and data structures, instead of writing software code. Citing [32], the "(...) goal of Outsystems (OS) is to enable and accelerate the development and delivery of enterprise web and mobile applications by providing a development environment that generates code and can deploy it to an enterprise-grade full stack system (...)".

2.4.1 React vs. Traditional Web

As nowadays, for web development, it is possible to choose between React or Traditional Web, a brief comparison between both will take place. The front-end architecture based on ReactJS avoids the traditional expensive server-side rendered applications, reducing the pressure on backend servers, allowing the application to an easier scaling up and to improve the overall UX [33]. According to [33], the advantages of using this library are: **faster** applications; a **single development experience**, since this paradigm was already being used for mobile application's development, but only recently was introduced for web development as well and it uses a **state-of-the-art architecture**, abstracting the developer's need to select and learn the constantly evolving Javascript framework. The three main characteristics introduced with Reactive are [33][34]:

- **Client-side logic** is available, other than the server-side logic, already present in the traditional web;
- There is no preparation, as the **data is fetched asynchronously**;
- Although in the traditional web, one had to perform Asynchronous Javascript and XML (AJAX) refreshes, when using React, explicit server-side rendering is present.

The Outsystems platform can be separated into 4 applications (Service Studio, Service Center, LifeTime and Integration Studio), each with a different goal to aid the developer in the designing, management and maintenance processes.

2.4.2 Service Studio

Service Studio, visible in Figure 2.5, contains the development environment that allows the creation of mobile and web applications. As it can be seen from the figure, it is similar to a conventional Integrated Development Environment (IDE). Still in the figure, on the left, it is possible to observe a web page with components, such as a button, a table and a plot. A sidebar is also visible, containing several pre-built widgets, that can be used to construct a web page or a mobile screen. Additionally, in the top-right corner of Figure 2.5 it is possible to see that there are four tabs: Processes; Interface; Logic and Data.

2.4.2.1 Interface

Starting with the Interface, it is possible to create, both new screens and new Web Blocks. As an example, in Figure 2.5, one has defined a web screen, which is using the WB `Menu` (the top menu that will be reused in every web page). Moreover, WBs can be used, not only inside web screens, but also inside other WB, although, a WB can not reference itself, since that will result in infinite calls. The platform is prepared against that. Another important note regarding WBs is that they have their own scope and can not access the scope of the screen/WB they are in, implicitly. In the same way, the parent (screen/WB

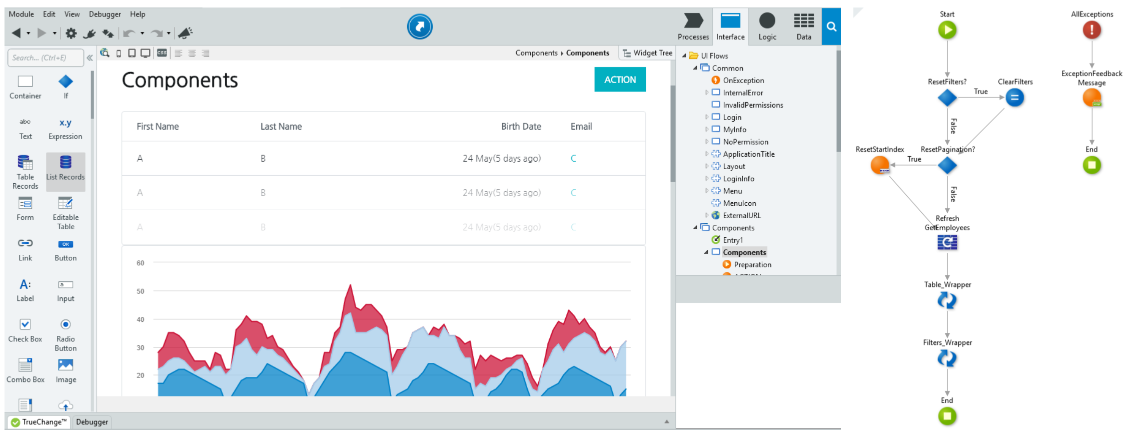

Module Edit View Debugger Help
Processes Interface Logic Data
Components
ACTION
First Name Last Name Birth Date Email
Container If Text Expression Table Records List Records Form Editable Table Link Button Label Input Check Box Radio Button Combo Box Image
UI Flows Common OnException InternalError InvalidPermissions Login MyInfo NoPermission ApplicationTitle Layout LoginInfo Menu MenuIcon ExternalURL Components Entry1 Components Preparation
TrueChange™ Debugger
Start ResetFilters? ClearFilters True False ResetStartIndex ResetPagination? True False Refresh GetEmployees Table_Wrapper Filters_Wrapper End
AllExceptions ExceptionFeedback Message End

Figure 2.5: Service Studio interface: screen designing on the left and logic fluxogram on the right.

that contains another WB) can not directly access its child variables directly. Events, which are an asynchronous way to notify the parent and pass its variable(s), are required for this. The advantage of using WBs is maintainabilit, since if some change is required in several screens using that WB, it is only required to make that change in one place; as well as encapsulation and abstraction of logic from a screen.

2.4.2.2 Logic

In Outsystems, when one has to encapsulate logic that implements the business rules of an application, actions are needed [35]. In Figure 2.5b, one can see some front end logic. Instead of written lines of code, Outsystems allows to program visually, with a resemblance of a fluxogram. Some differences between traditional web and React exist when building application logic.

Starting with the traditional web, a Preparation can be included, usually, to fetch data from the database, which will later be displayed. There is no need to call the preparation, as it is automatically called before the rendering [35]. On the other hand, screen actions allows the website to run server-side logic when the user interacts with the screen (*e.g.* click on a button or link) [35].

In React, however, a Preparation is not needed as data is fetched asynchronously from the database. Also, actions can now be executed on the client-side and not only on the server. A client action (formerly known as screen action) can call server actions, but not the other way around. Also, performance-wise, it is not advised to call several server actions inside a client action, as it will result in multiple server calls and might delay the screen rendering and decrease the UX. Moreover, each Aggregate can have an after fetch event, which can execute more logic. Finally, there are actions called data actions, which are executed in the same way as the Aggregates, asynchronously and are generally used to execute SQL queries.

It is also possible to consume or expose Simple Object Access Protocol (SOAP) web services and REST APIs, as well as manage roles. The latter allow managing what an user can and can not access. It is possible to place roles on screens and decide if certain components should be shown inside a page. Outsystems provides a function called `Check<RoleName>Role` that return true or false, depending if the user has the `RoleName` permission.

Furthermore, it is possible to create Exceptions. After creating a new Exception it is possible to trigger them in the code and create handlers for those exceptions. An `AllExceptions` handler is visible in Figure 2.5b, which is the most general exception handler. If, for a certain exception, more specific handlers exist, those will be executed instead.

2.4.2.3 Processes

To perform asynchronous operations, one can create Timers and Asynchronous Processes, with the latter, also known as Business Process Technology (BPT), allowing to manage business processes, regarding the life cycle of an entity [36]. For instance, taking as an example, the IoT use case of this thesis, it might be required to archive sensor data that is older than x weeks, so that data can be loaded faster. For that, BPTs are used. A process can either start when an entity is created or by a trigger in an action [36]. This said, BPTs can be compared to threads. When a process starts, a process instance is created and every time a process activity is found, an activity instance is created [36]. Regarding Processes, it remains saying that BPTs are designed for activities that wait long times, such as months, until an event occurs. For large scale batch processing cases (eg. event broker), Outsystems recommends the use of Light Weight BPTs.

2.4.2.4 Data

Firstly, OS permits the creation of database entities (tables), as well as its attributes. To access entities on the screen, one can use Aggregates and SQL, with the difference that the former is optimized by the platform. However, some things can not be achieved without SQL, such as bulk operations and unions, for instance. As it was seen prior, when in traditional web, to fetch data to screens, a Preparation had to be used but when using React, these are not needed and data is fetched asynchronously, either through Aggregates, or data actions. Also, react allows to fetch data from the database when the user enters the screen or on demand, whenever needed. Plus, one of the ways Outsystems optimizes applications is by executing Aggregates in a specific order, if the result of one is required in another.

Other than that, it is also possible to create structures. These are simply custom data types, that can be composed of attributes, both of simple and complex data types. A structure can even have a datatype of another structure. One great advantage inhere is the possibility to convert data structures to JSON and vice-versa, by using functions built in the Outsystems platform. Finally, it is also possible to create Site Poperties, which are variables that do not change often and Resources which are files that are stored in directories.

2.4.3 Service Center, LifeTime, Integration Studio and Forge

Starting with Service Center, it allows to manage applications and monitor its logs, including error logs. Moreover, it is possible to manage timers; integrations; processes; the environment health and security, among others. All of this can be accessed on the browser.

Lifetime is also a Web application that extends the Service Center's capabilities across the different servers in the life cycle of the application. It is possible to track applications and its versions.

Integration Studio will not be used in this thesis but allows the creation/management of extensions, which are a set of actions, structures, and entities, that increments OutSystems and allows the integration with external systems [37].

Finally, the Outsystems Forge is a source of downloadable components that can help speed up application development and delivery. In Chapter 5, the creation of a component to rearrange a dashboard using drag and drop takes place.

Chapter 3

Architecture

Before starting the developments, one of the key aspects to ensure their success and avoid technical debts in the future is to design a good architecture, which, together with the modularization of the applications' modules, allows to develop new functionalities and maintaining the code in an easier and faster way.

3.1 Introduction

Figure 3.1 shows this thesis (and its platforms) high level diagram. Firstly, Outsytems (see Section 2.4) was used to develop an IoT management platform (web and mobile). Transducers are connected to the platform through devices. Finally, Outsystems allows reusable components to be created and shared to the developer's community. The gray blocks in Figure 3.1 represent the initial developments, after which, a second application for mobile devices was developed, reusing the previous work.

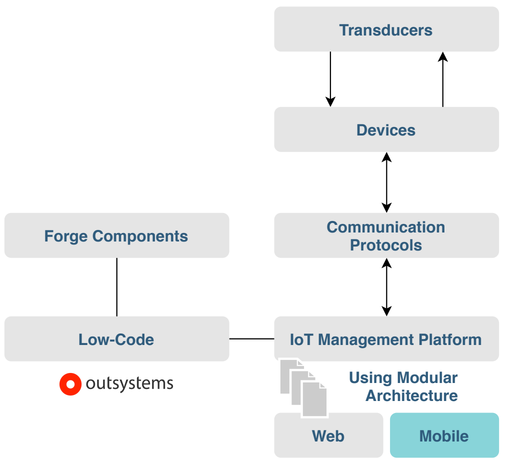


Figure 3.1: High level overview.

There are 3 main steps [38] on the architecture design process: **disclosure** (Section 3.2); **organization** and **assembling** (both in Section 3.3). The thinking process behind each of these steps, applied to this thesis, will now be discussed.

3.2 User Stories and Database Design

The first step when designing an application architecture is the disclosure, which includes figuring out the business concepts and integration needs. To help with this, user stories will be created.

According to [39], a User Story (US) "is the smallest unit of work in an agile framework". It is "an end goal, not a feature, expressed from the software user's perspective (...) often expressed as persona + need + purpose":

As a ______ , I want to ______ , so that ______ .

Where a persona is a person that needs to solve a problem: a representation of the user base [40]. By creating personas, one not only thinks about the problem that needs solving, but also, the person that needs that problem solved [40]. Different personas can have multiple roles, for instance, a persona can be an administrator and a devices' manager at the same time. Nonetheless, for this thesis, one will assume that each person has a single role, therefore, there will be 4 personas. The result is depicted in Table 3.1. The functionalities of the different users will become more clear, once the USs are defined.

Table 3.1: Users and groups permissions.

Roles	**Description**
Admin	Will have all of the permissions and no other user will be able to change his status.
UsersManager	Can create other users; create user groups; add users to groups; and add roles to users and groups. Note that it can not add the Admin role to a user/group.
DevicesManager	Can create and edit devices, as well as change its location.
LocationsManager	To create and edit IoT locations, as well as add other users for those to be able to visualize all of the dashboards associated with that location.

More roles can be added in the future and, since the architecture will be counting on it, the development effort will not be as high. Given the different types of personas, one can start writing the USs and, through a non-technical language, see the full picture of this project and design the application's architecture.

3.2.1 Users and Groups Management

The users/groups' management USs are shown in Table 3.2. Essentially, one will need a web page to manage both roles and users, as well as a page to create users.

Table 3.2: Users and groups management user stories.

	User Stories
US 1.1	As an admin/users' manager, I want to create other users, so that they can access and work on IoT projects.
US 1.2	As an admin/users' manager, I want to create groups with users, so that users can be grouped in teams.
US 1.3	As an admin/users' manager, I want to manage the roles and permissions of users/groups, so that different users have different permissions.

Furthermore, the Table 3.2's content allow us to design the initial part of the database (users/groups' entity diagram), presented in Figure 3.2. To fulfil the USs' requirements, one would need a table for: the users; groups; to assign groups to users and two others to assign roles to both users and groups. These are all system tables.

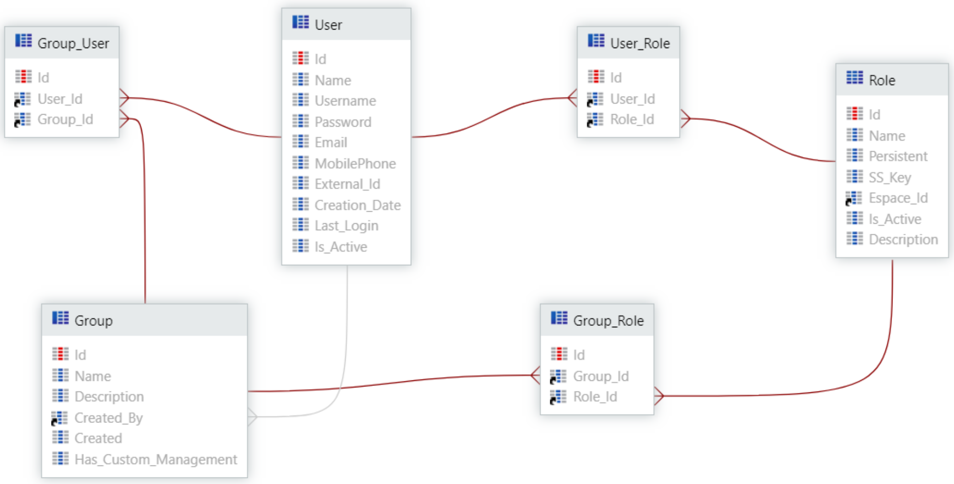


Figure 3.2: Users and groups database entities and relations.

3.2.2 Devices and Locations Management

For the remaining USs, (Table 3.3), regarding the devices, locations and dashboards functionalities, for both the web and mobile applications, the database entity diagram shown in Figure 3.3 was created. On this diagram the relations between users and locations, as well as the relations between devices and users, and between locations and devices are represented.

Table 3.3: Devices and locations management user stories.

	User Stories
US 2.1	As an admin/locations' manager, I want to be able to create new IoT locations, where each location contain devices, so that I can control that location.
US 2.2	As an admin/locations' manager, I want to be able to decide which users can access which locations, so that only users with permissions can access a specific location.
US 3.1	As an admin/devices' manager, I want to be able to create and install devices on the platform, so that new devices can be included in IoT locations.
US 3.2	As an admin/devices' manager with permission to access certain locations, I want to be able to assign devices to those locations, so that each device always has an IoT location.
US 3.3	As an admin/devices' manager, I want to be able to activate/deactivate those devices, so that devices that are not being used at the time, are not receiving/sending topics.
US 3.4	As an admin/devices' manager, I want to be able to set my device's parameters, so that it can communicate with my IoT platform.

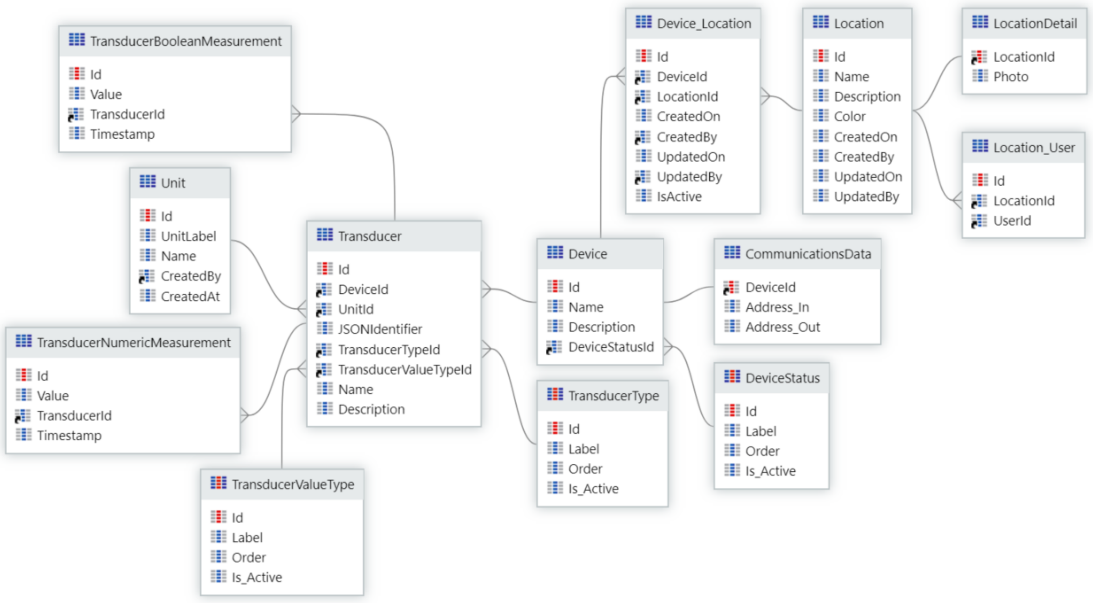


Figure 3.3: Devices and locations database entities and relations.

The database entity diagram in Figure 3.3 contains a table for storing locations, with the following attributes: name; description and color (for personalization and user experience purposes). This covers US 3.3. Attributes for storing the location's creator and the time of creation were also included. Moreover, to ease the identification of a location by a user, the location's picture is also included in a separate entity ($1-1$ relation), as it is a good practice to place attributes such as photos in a separate entity, so that they are not loaded when refreshing a query or reloading a page, causing the web page to be heavier and, therefore, slower. Another table, named `Location_User` allows to identify which users have access to a specific location. This covers the USs 2.1 and 2.2. For the US 3.2, the table `Device_Location` associates locations with devices with a many to many relation. Other than the location and device Foreign Key (FK)s, it contains an history of the creation and last update (date and user), as well as a flag indicating if the current device-location relation is active, useful because a device can be associated with one location, but then moved to another, keeping the old's location data.

For the USs 3.1 and 3.3, the table `Device` contains a name; description; and a FK pointing to the device status. You may have noticed that the protocol associated with the device is not in Figure 3.3 (to simplify the project). Currently, the device status can be of 3 types: **Pending Installation** (when a device gets registered in the database with missing information; **Active** and **Inactive**. This `Device` is connected to a `Transducer` ($1-N$ connection), which can be of type sensor and actuator and can measure/send multiple value types (in Figure 3.3 represented with the tables `TransducerBooleanMeasurement` and `TransducerNumericMeasurement` for boolean and decimal values' measurements, respectively). The `Transducer` also has units' information, by containing a FK pointing to the `Unit` table. This table has the label itself (*eg*. °C) and a name to depict in plots, for instance, Temperature. Finally, a `Transducer` contains a `JSONIdentifier`: a `Text` that arrives in the device message payload (JSON) identifying the Transducer inside a device.

Finally, the entity `CommunicationsData` allows to connect to several protocols without worrying about its specificities. This entity contains the sender and recipient addresses which, in case of MQTT would be topics, but are also suitable for other protocols. This address can contain an IP:Port, for instance.

3.2.3 Dashboard Configuration

As the user needs to create/configure dashboards, add cards and change their position, it is required to create some USs for these use cases: Table 3.4. The database entities and their relations are shown in Figure 3.4.

First, one needs a table to store dashboards. Other than a descriptive name and description, it will also be associated with a location.

Then, each template can contain several cards and a `Card` can contain different types of data and present it in different ways. For that, the table contains the FKs of the `CardType` and `CardSize` static tables. The first indicates if it is an spline plot, a gauge plot, a table,

Table 3.4: Dashboards user interactions user stories.

	User Stories
US 4.1	As an admin/locations' manager, I want to be able to create new dashboard templates, so that different IoT use cases are associated with different templates.
US 4.2	As an admin/locations' manager, I want to be able to add cards to a template, so that the template has the required devices' information.
US 4.2	As an admin/locations' manager, I want to be able to change the position of the cards on a template, so that each snippet of information is positioned where it is preferable, to increase efficiency.

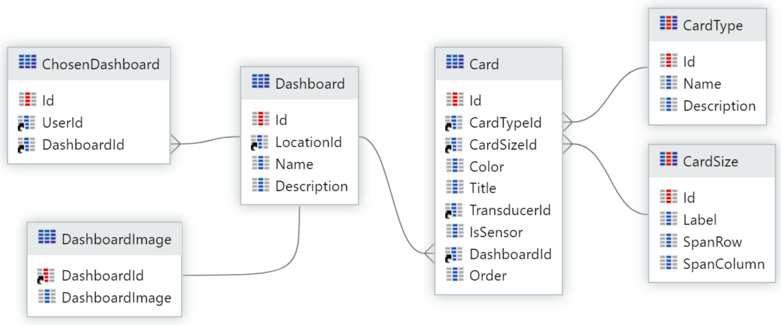


Figure 3.4: Dashboard interactions database entities and relations.

a value indicator, or a button; whereas the second contains the size of the card (note the attributes SpanRow and SpanColumn): if it is a (1×1), (2×2) or (2×4), depicting the amount of space the card will occupy on a dashboard's grid. Moreover, each card will have a title; an indication (boolean) if this card belongs to a sensor or to an actuator; and the transducer identifier FK. Also, there is a color attribute, for personalization and UX purposes. Finally, a card as an integer, `Order`, to organize the way it is presented on a template.

3.2.4 Mobile Application

Finally, the USs for the mobile application are shown in Table 3.5, containing functionalities such as a screen to manage devices and locations, as well as the possibility to visualize dashboard data (US 5.1). Also, to showcase the mobile functionalities of the novel application, devices' data will be shown on the dashboard even if the device is offline. This will require local storage entities, as seen in Figure 3.5. It can be seen that some parameters such as `CreatedBy` and CreatedAt were not included. This is because it is advised to reduce the number of attributes in local storage.

This concludes Section 3.2, where both the USs and Database (DB) entity diagrams were presented. By creating USs, even considering that they should be more specific, it is easier to see the big picture of this project, which eases the design of the database.

Table 3.5: Mobile Application User Stories.

	User Stories
US 5.1	As an IoT user, I want to be able to access all of the main functionalities of the web application on a mobile phone, so that I can work remotely.
US 5.2	As an IoT user, I want to be able to access and view my dashboards on a mobile phone, even if I do not have an internet connection.

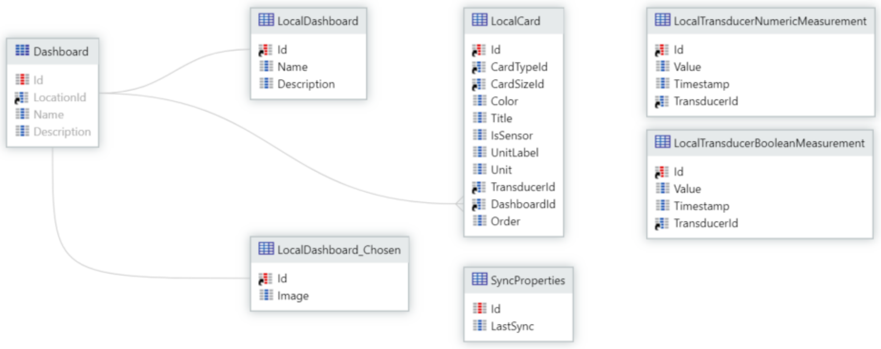


Figure 3.5: Local database for mobile application (Offline Synchronization).

3.3 Information Architecture Organization and Assembling

The remaining steps when creating an architecture are **organization** and **assembling**. For these, following an architecture canvas yields faster development times; a common understanding; and it eases the architecture assessment. Outsystems used to provide a framework, 4-Layer Canvas (4LC), but it is now obsolete.

First, one needs to organize the application's key concepts on the canvas and review those concepts. Important aspects to keep in mind at this stage are: a good architecture should not be based on technical concepts; strongly connected concepts should be joined; concepts that are too different should not be joined; recommended patterns should be used and one should not encapsulate in excess, as it contributes to a poor service abstraction, making it harder to manage dependencies.

When designing an architecture, an iteration occurs between identifying concepts and defining modules (Figure 3.6). Outsystems, mentions that it "is a continuous process (...) as a solution evolves and new concepts and needs emerge (...)" [41].

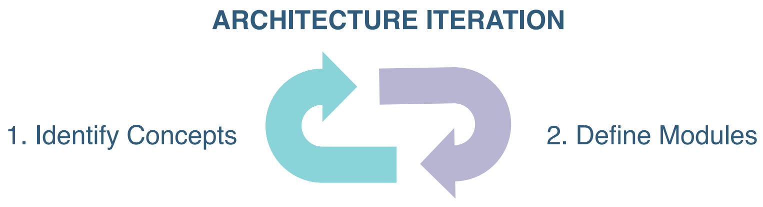


Figure 3.6: Architecture Iteration (Adapted from [41])

The different architecture layers will be used [41]:

- **End-User**, which are business interfaces, processes and logic that provide the features available to end-users. Reuses core and library components;
- **Core Business**, isolating components around business concepts, shared by different end-user modules. The components can be: core entities; change operations; business logic and asynchronous data processing;
- **Library**, with highly reusable components that may encapsulate services and protocols from external providers split by functional groups.

Some rules worth mention are: first, there can not be any upward references. This means that, for instance, a Library module can not consume a Core module. Finally, there should not exist any circular reference, so that publish a module *A* will require publishing a module *B* and publishing a module *B* will require publishing the former. For this thesis, the Outsystems naming conventions will be used, so that each module has a name related to its functionality. Starting with the **Library** modules, there will be an `IoTWeb_Th` and `IoTMobile_Th` that will contain the theme and all of the Cascade Style Sheets (CSS) rules, for the web and mobile applications, respectively, as well as an `IoTWeb_Pat` and an `IoTMobile_Pat`, that will contain reusable UI patterns for the layout, both web and mobile. In the future, in this layer, there can also be included some integration services, for instance, to consume REST services.

Moving on to the **Core** modules, those will use the suffixes Core Services (CS) and Mobile Core Services (MCS), containing public entities and actions, where the latter are specific to the mobile use cases. For the CS, the modules are `Devices_CS`; `Locations_CS` and `DashboardConfiguration_CS`. The first one will contain all of the database entities and actions (namely, Create, Read, Update and Delete (CRUD) actions) regarding IoT devices; the second has the same for locations and the latter, for creating and configuring a dashboard. Additionally, for the mobile application, `Dashboard_Config_MCS` will also be included. The need for this module relies on the fact that a mobile device's screen is smaller and, because of that, specific configurations are required.

Also, there will be a module named `Users_CS`; Protocols_CS and, for this thesis, a module named `MQTT_CS`. Any other protocols added in the future will also have a core service module associated.

Once the complexity of this application increases, a portion of the CS modules may become a Business Logic (BL) module.

Finally, regarding the **End-User** modules, there will be 2 modules: `IoT_Mobile`, `IoT_Web` and `IoT_Mobile`, which will implement the end-user interfaces for the web and mobile applications, respectively, for the user to interact with. Also, there will be a backoffice to manage the database (mainly during tests). These modules will consume the core-business and library modules mentioned *a-priori*.

The modules considered at this initial stage are shown in Figure 3.7, organized by layer. Nevertheless, the reader must keep in mind that, if any improvements are required in the future, changes can be done without much effort, although it is a good practice to implement a good architecture from the beginning to avoid time consuming changes in the future. It will be shown in Chapter 7 that this modular architecture allows to develop for both the web and mobile applications at the same time. This decreases the development times and eases code maintenance and bug-fixing, as a module is reused in both applications. In Chapter 4, the developments are shown.

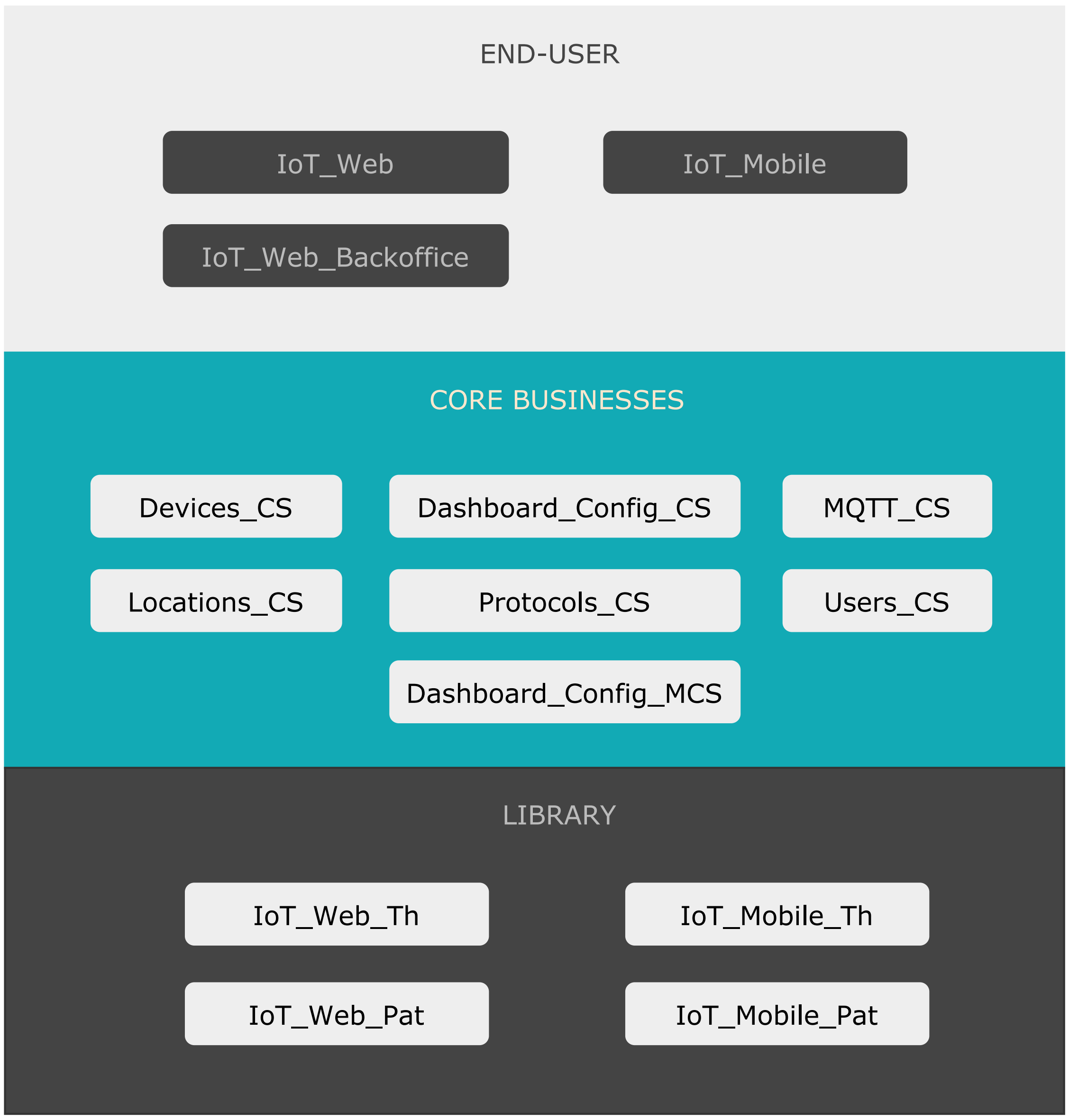


Figure 3.7: Project Modules (Architecture).

Chapter 4

Developments

After building the architecture, it was time to develop the Outsystems actions. This can contain either CRUDs, screen or business logic and can be either server or client actions.

4.1 Locations Actions

A location is the physical place where a device will be located. For instance, in an agriculture use case, there may be several crops and each will contain its own devices. Thus, each crop can be classified as a location. The locations' actions that will be developed (with its inputs and outputs) are shown in Table 4.1 and Table 4.2.

Table 4.1: Locations create or update actions.

(a) Location_CreateByName Action

Inputs
• Name
• UserId
• Description
• Color
• IsActive
• Photo

Outputs
• Error
• LocationId

(b) Location_Update Action

Inputs
• Location
• LocationDetail
• UserId

Output
• Error

In Figure 4.1, the Create and Update operations are shown. The Create action, Figure 4.1a, receives the Location data as input: name; description; color; status; an optional photo; and the identifier of the creator user. To create a novel location, first, one checks if the provided location name is not an empty string. If so, an aggregate will fetch a single row with a `Location` with the same name as the provided location name. If there is already a location with that name, an error is thrown and the `Success` value is set to `False`, as well as an error message is defined. Otherwise, the `Location` is created and the output `LocationId` is set with the output of the `CreateLocation` entity action. Furthermore, the user that created the location is automatically assigned to the location, calling the `User_ToggleInLocation` action. If no photo was provided, nothing more happens, otherwise, the `CreateLocationDetail` entity action is called.

Furthermore, `Location_Update` contains the `Location` and `LocationDetail` entity records as inputs. In this action (Figure 4.1b), after checking if the `Location` is not empty and if a location with the same name and a different identifier already exists, the `Location` is updated. After this, if there was a photo, previously, but no photo was provided now, the old photo is deleted.

4.2 Location-User and Device Actions

As shown in Table 4.2, 2 actions were developed to add or remove users and devices to or from locations, respectively. Due to the fact that both actions are similar, only the `Devices_ToggleInLocation` implementation in Outsystems is presented in Figure 4.2. First, an aggregate fetches the `Device` inner joined with the `Location`, where `Location.Id =` ↪ `LocationId` (to check if the location exists) and left joined with the `Device_Location` (to check if the device is already active in any location). If the aggregate is empty, this means that either the location or the device were not found and an error is thrown. Otherwise, the device is added as non-installed and the `IsAdd` variable is set to `True`. If the latter is set to `False`, then the sensor is removed if it is not in a different location. When an error is thrown, the error message is either logged and/or shown to the user.

Table 4.2: Actions to add or remove devices and users to/from locations.

(a) Device_ToggleInLocation Action

Inputs
• DeviceId
• LocationId
• IsAdd
• UserId

Outputs
• Error
• DeviceLocationId

(b) User_ToggleInLocation Action

Inputs
• UserId
• LocationId
• IsAdd
• CurrentUserId

Output
• Error
• LocationUserId

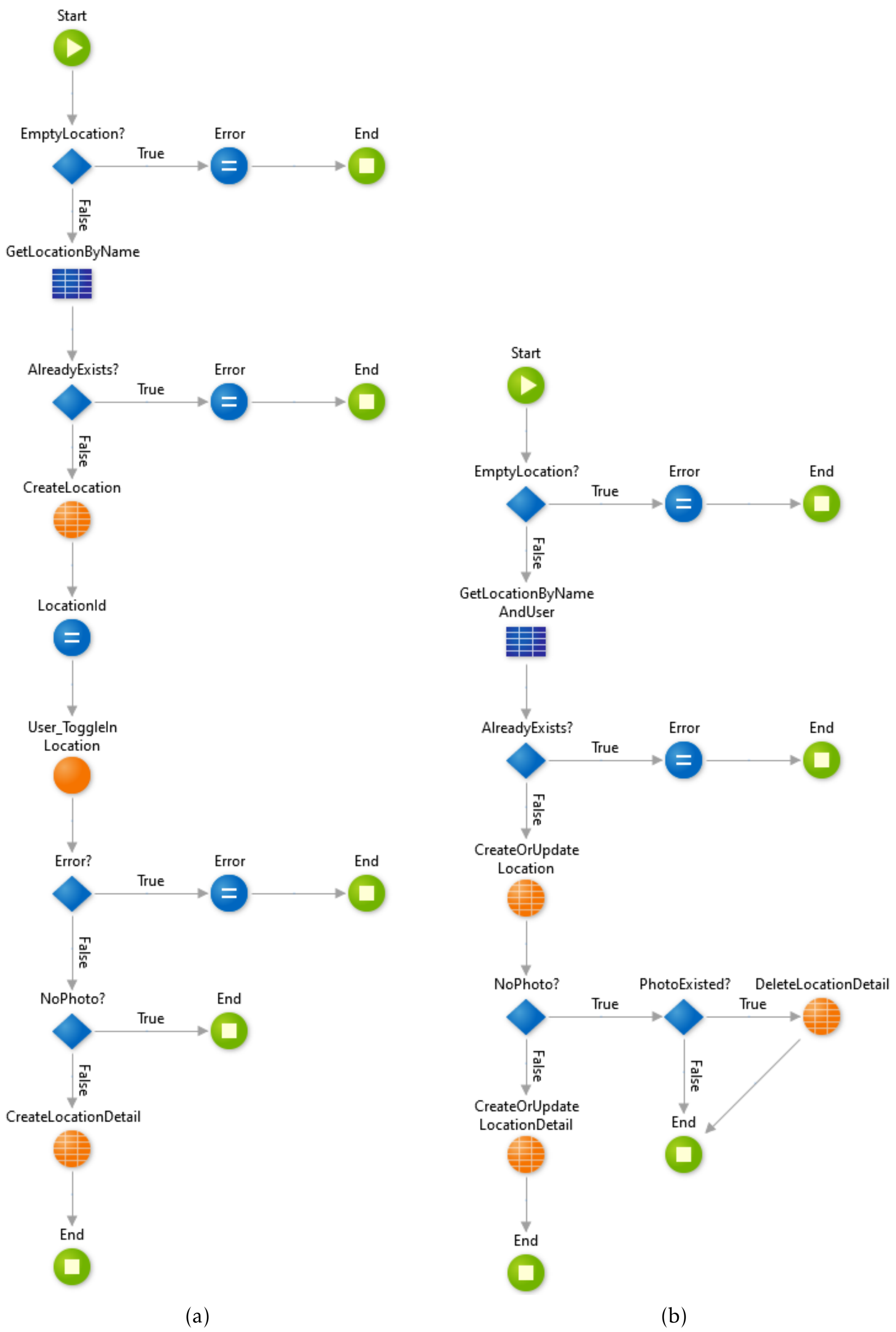


Figure 4.1: Locations Create or Update Actions.

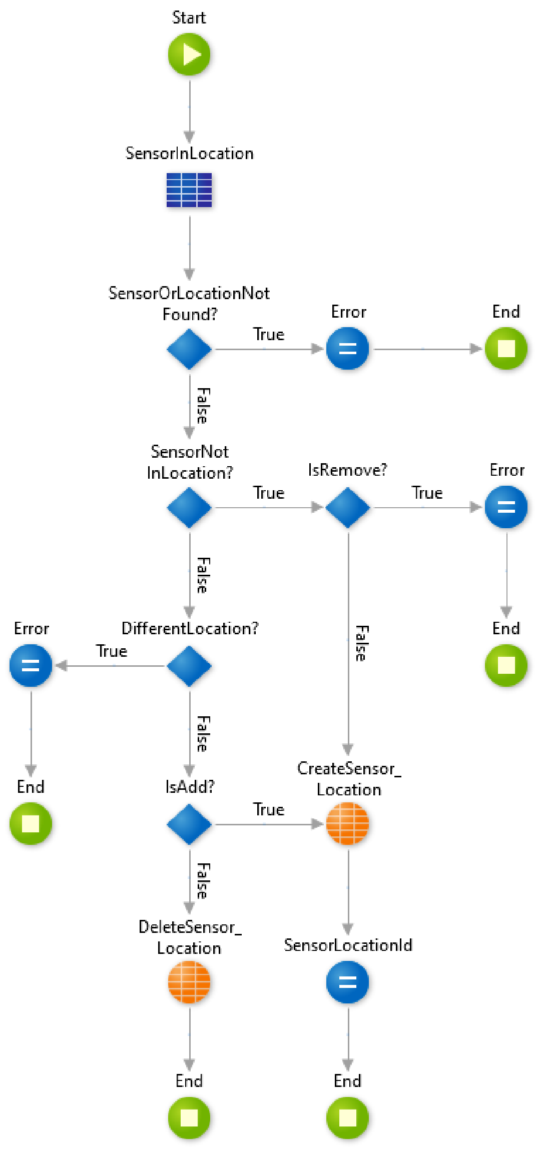


Figure 4.2: Outsystems action to either add or remove sensor to/from a location.

4.3 Locations List Screen

This screen has an aggregate to fetch all `Locations` left joined with both `User_Location` and `LocationDetail`. It only fetches records where the `User_Location`'s user is the one in session. Next, a data action to get the `Location`'s details is refreshed whenever the user clicks on a location item on the screen list. This action (Figure 4.3) receives the `SelectedLocationId` ↪ and the `ClickedItemId` as inputs. It saves the provided location identifier on a screen variable and then refreshes the data action. After storing location identifier and fetching the devices and users associated with that location, through 2 different aggregates, the resulting filtered data is appended to outputs that will be used on the screen, in tags as seen in Figure 7.3c).

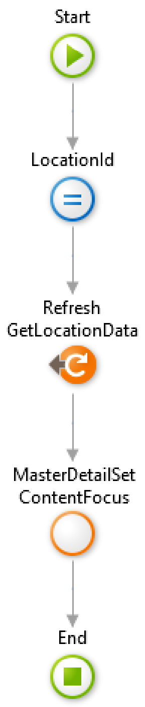


Figure 4.3: OnLocationListItemClick Client Action (Location List Screen).

4.4 Locations Detail Screen

When clicking on the buttons "Edit" or "Add Location", on Locations List screen, the user is redirected to the screen detail, where he can edit or create a new location.

The action to save a location is shown in Figure 4.4. First, it is verified if the form is valid. Moreover, a server action is called. If this action results in an error, an error message is shown to the user. Otherwise, a success message is shown. In case a new `Location` is created, the `Location` identifier is updated.

Next, follows the client actions to add or remove a device to/from a location. These are shown in Figure 4.5a and Figure 4.5b, respectively.

Starting with the the action to remove devices from a location, as it is simpler, it calls the server action `Device_ToggleInLocation`, already explained above (review Figure 4.2) to update the database. If there are no errors, the position of the sensor in the `Tags` list is found, after which, that device is appended to the dropdown list variable and removed from the respecitve `Tags` list.

The client action to add a device to a location (Figure 4.5a) starts by verifying if the plus `Icon` was pressed. If so, the sensor identifier of the sensor selected on the dropdown is stored in a local variable. Otherwise, the same server action as before (`Device_ToggleInLocation`), is called and a similar procedure is followed. If there are no errors, the index of the sensor in the dropdown list is found, after which, that sensor is appended to the `Tags` list and removed from the `SearchDropdown` list variable. A success message is shown to the user.

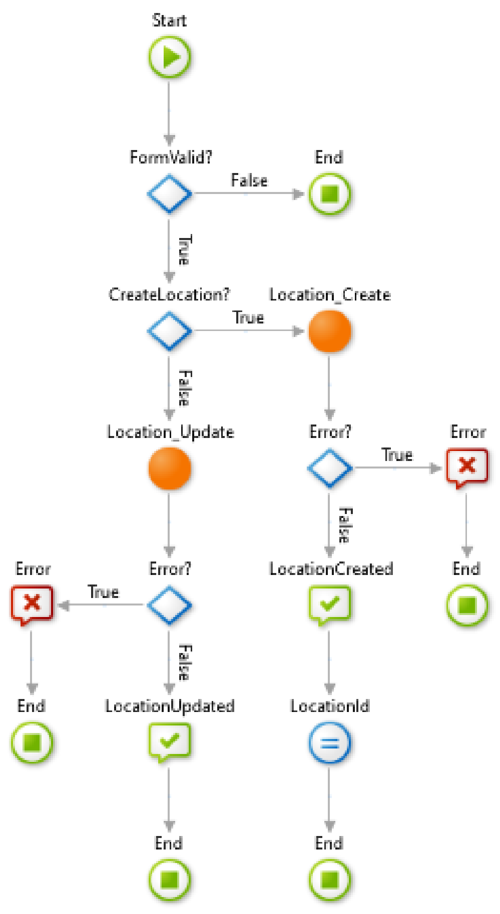


Figure 4.4: OnLocationSaveClick Outsystems Client Action (Location Detail screen).

At last, the screen has logic to decide which parts of the screen to show or to enable. For instance, `If` widgets are used to decide whether to show the lists with the sensors/actuators/users tags or to show messages saying that the information is not available; `Expressions` are used to vary the information shown on the labels; the `DropdownSearch` is disabled in case the location does not exist yet and the "Add" link is enabled if a location exists and if there is a device/user selected on the respective `DropdownSearch`. The latter can be observed in Section 7.4, Figure 7.4 where the "plus" `Icon` is disabled, as there is no device selected on the respective `DropdownSearch`. Furthermore, roles are taken into account, for instance, to decide whether to show the link to add/remove users.

4.5 Devices Actions

The requirements for the devices can be inferred from Table 3.3, but in sum are: create and install a device on the platform; choose its protocol; assign that device to a location and be able to set its configuration parameters. The actions developed to create or update devices, with its inputs and outputs are depicted in Table 4.3 and its implementation is shown in Figure 4.6.

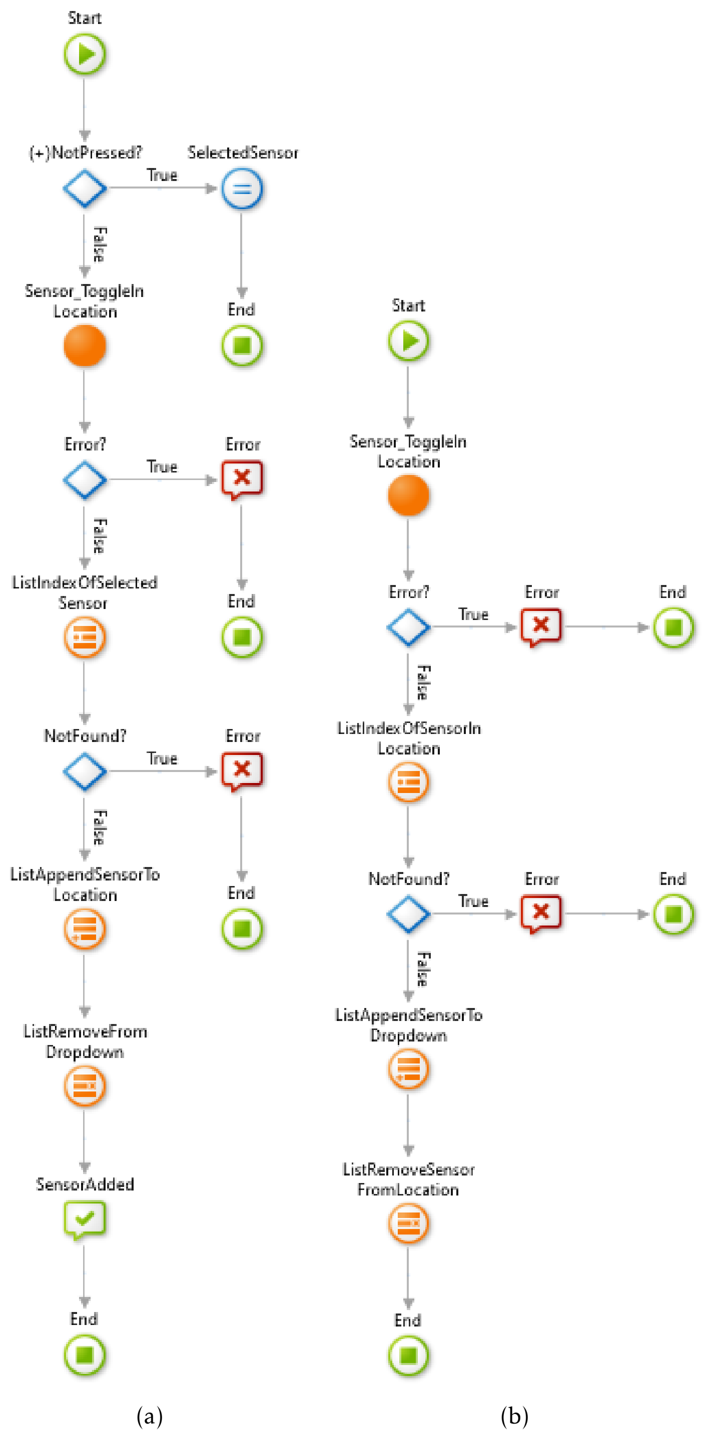


Figure 4.5: Outsytems client actions to add (a) or remove (b) devices from a location.

Table 4.3: Actions to create or update devices.

(a) Device_CreateOrUpdate Action

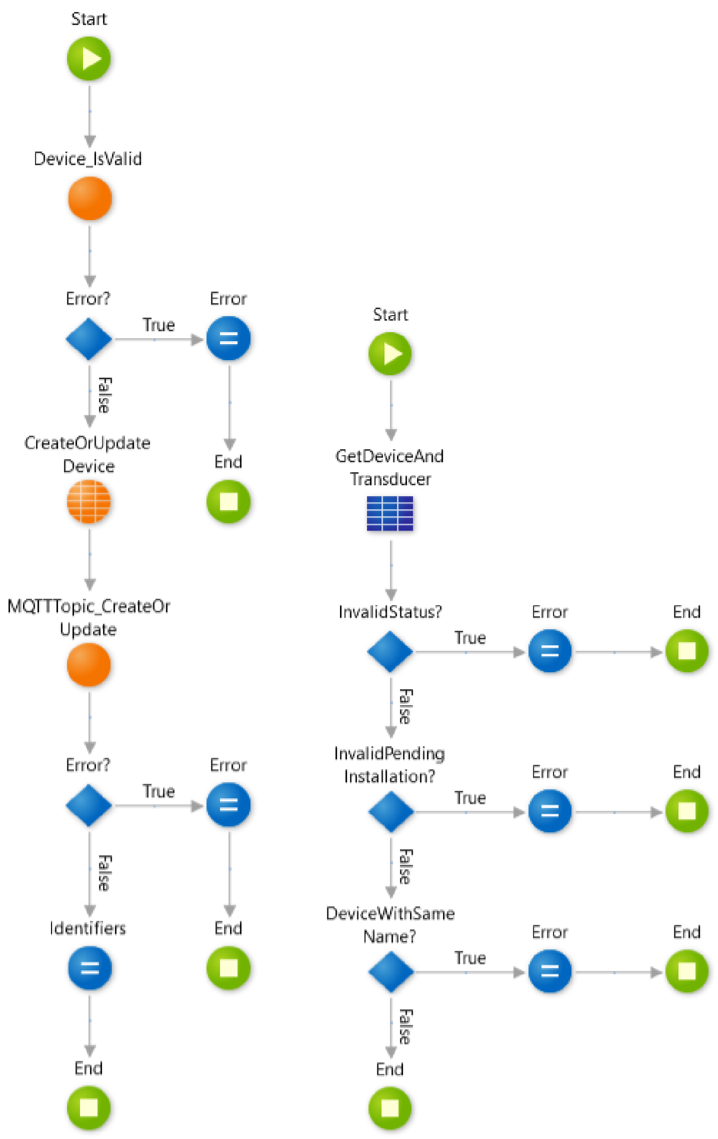
Inputs
• Device_In
• UserId
• CommunicationsData

Outputs
• Error
• DeviceId
• CommunicationsDataId

(b) Device_IsValid Action

Inputs
• Device_In

Output
• Error



(a) (b)

Figure 4.6: Actions to create, update and assess a device.

Starting with the action to validate a device (Figure 4.6b), it starts by verifying if the status is invalid, which happens if the input `Device` has either no `Name` or no `Description` and, at the same time, the provided status is different than `PendingInstallation`. The opposite is also verified: if the device has both a `Name` and a `Description` and, at the same time, the provided status is `PendingInstallation`. Next, it is assessed it there is a `Device` that has the same `Name`, but a different identifier of the input `Device`. If this happens, it means that there is already a different device with the same name registered on the DB. An error is thrown if any of this conditions is met.

The action to create or update a `Device`, in Figure 4.6a, starts by verifying if it is valid, by using the action explained previously. If is is valid, a novel `Device` is created or updated on the database. Next, using the `DeviceId` and the `CommunicationsData` provided on the action input, the `MQTTTopic_CreateOrUpdate` action is called to create or update the communications data. The actions related to the protocols are explained in Section 4.10. If there are no errors, both the `Device` and `CommunicationsData` identifiers are returned.

There are 2 Devices-related screens, the Device List and Device Detail. Screenshots of these can be seen in Chapter 7, Section 7.4, Figure 7.4.

4.6 Devices List Screen

To present the devices, this screen contains a table with its name, status and location as columns. The source of the table is an aggregate with the `Device` left joined with a `Device_Location`, inner joined with `Device status` and the `Device_Location` is left joined with the `Location`. Moreover, this aggregate can be filtered, either by location (selected through a dropdown input populated by a different aggregate), or by device name (searched by inserting text in an input). Both the selected location and the search text are saved in session variables. This way, if the user goes to another screen and returns later, his search preferences will remain. These session variables are used inside the aggregate to filter it and every time each of the filter parameters changes, a client action is called to refresh the aggregate, which will then refresh the table on the screen. By clicking on the "Create Device" button or on the device name, on the table, the user is redirected to the `Device` details, to create or update a device, respectively.

4.7 Devices Detail Screen

This screen has 2 aggregates, to fetch the `Device` details given the input `DeviceId`, left joined with the `Device_Location` and `CommunicationsData` and the locations, as one can change a device location through a dropdown input; and has one data action, shown in Figure 4.7a, which will get all transducers associated with the device. If no device identifier exists, the data action does nothing. Otherwise it will fetch the transducers associated with the device. Next, a filter will separate the sensors from actuators and both will be appended to 2 different outputs of the server action.

The screen contains two lists that show the sensors and actuators separated and is possible to remove those by clicking on an "x" icon. The action to remove a transducer is shown in Figure 4.8a Otherwise, if the lists are not populated, they show a message such as "No sensors added to this device.". An `OnAfterFetch` event for the aggregate that fetches the device is used to assign the status of the device and of the toggle widget on the screen. This action is shown in Figure 4.7b. To add a device, after filling all of the form details, the action in Figure 4.8b is called and the device is saved.

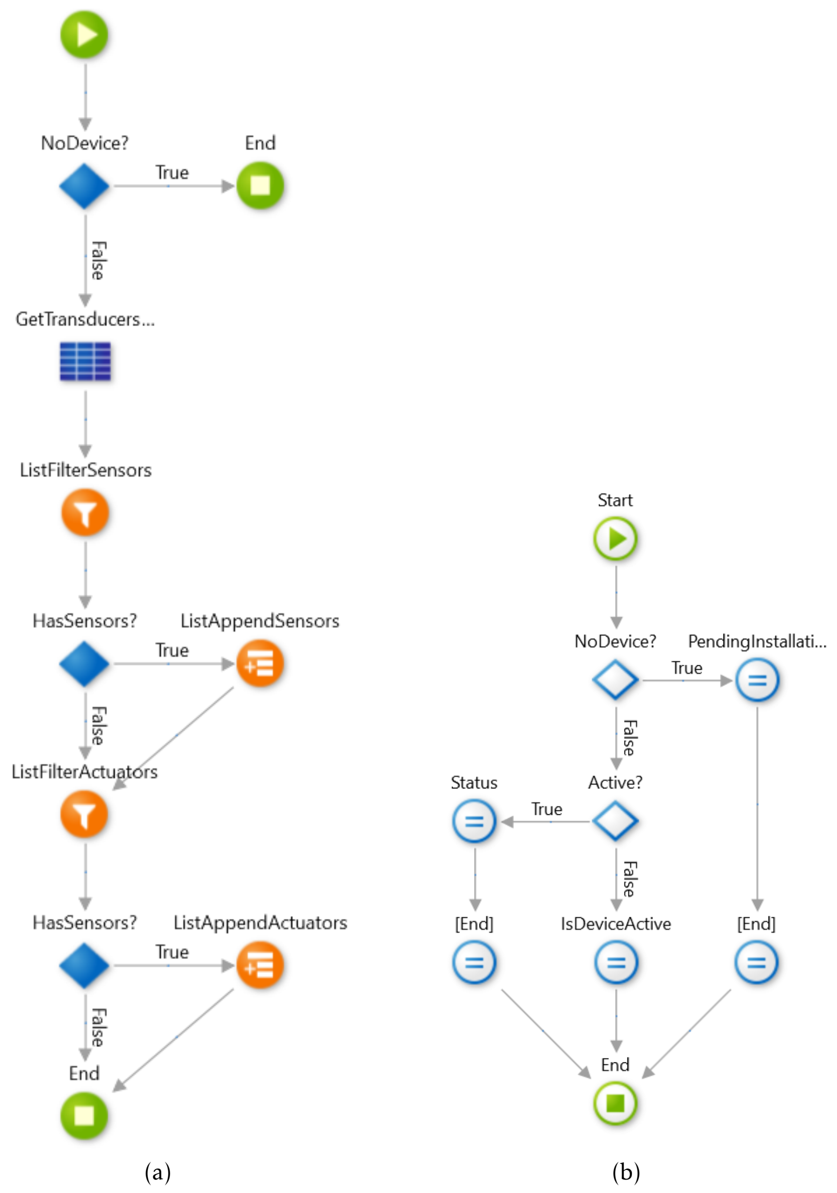
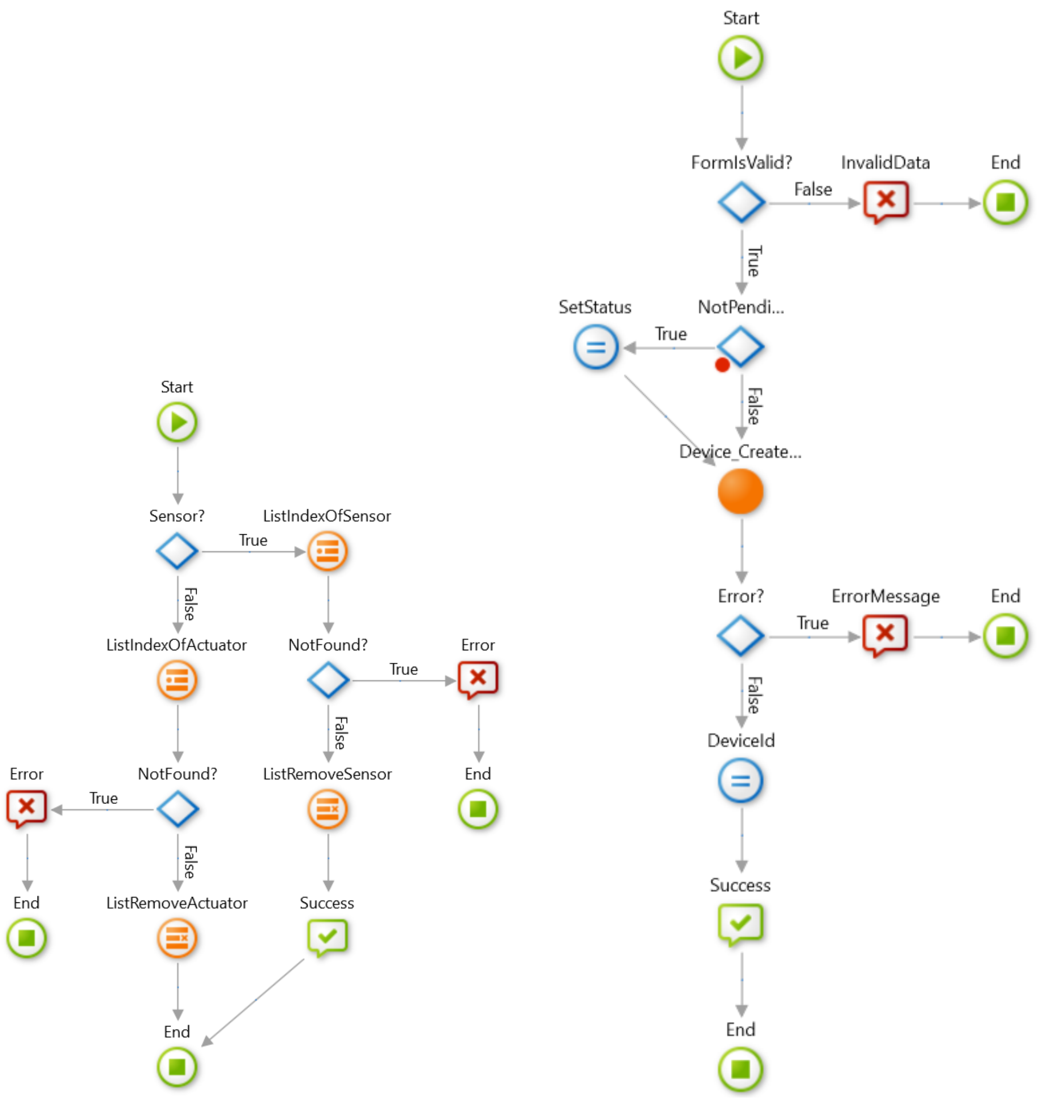


Figure 4.7: Action (a) to fetch the transducers and (b) after fetching the device.



(a) (b)

Figure 4.8: Actions with (a) to remove a transducer and (b) to add a device.

4.8 Transducers Actions

It is possible to associate `Transducers` to a `Device` by clicking on the "Add Transducer" button, which will open a popup. The actions to assess and create or update a transducer, with its inputs and outputs are shown in Table 4.4.

Regarding the action to validate a `Transducer`, seen in Figure 4.9b, it starts with an aggregate that will fetch a `Device` where the identifier will be the same as the one in the `Transducer` input record. The `Device` is left joined with a `Transducer` to see if the `Device` already has a different `Transducer` with the same `Name` and `JSONIdentifier` associated. The `Device` is also left joined with the `ValueType` and `TransducerValueType`. `If` conditions are then used to verify if these types are valid, if the device exists and if there is already another transducer with the same details as the one being added.

Table 4.4: Actions to add or remove devices and users to/from locations.

(a) Transducer_CreateOrUpdate Action

Inputs
• Transducer_In
• UserId
• Unit

Output
• Error

(b) Transducer_IsValid Action

Inputs
• Transducer_In
• Unit

Output
• Error

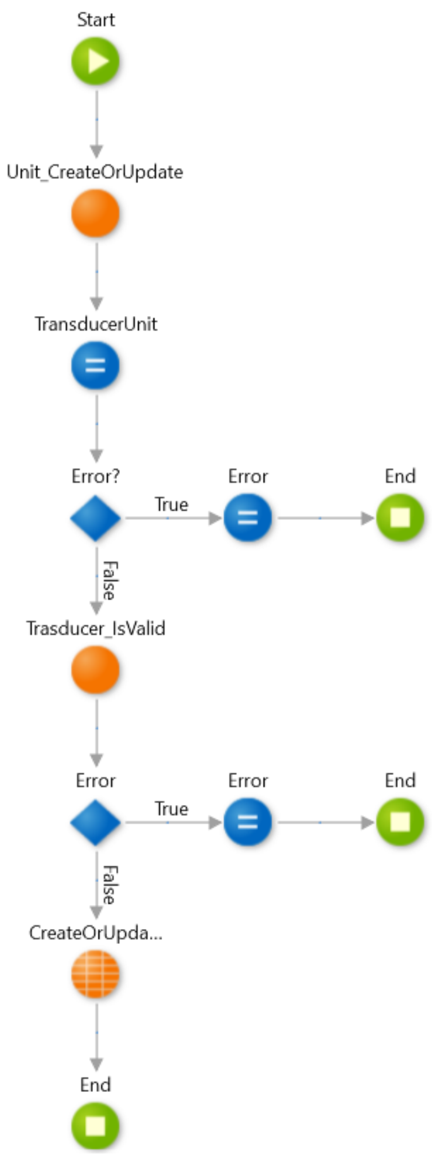
If none of these conditions fails, the chosen `Unit` is also assessed. Moving on to the create or update action, in Figure 4.9a, it uses the previous action to verify if the `Transducer` is valid. If there are no errors with the `Unit` and with the `Transducer`, the latter is created or updated.

4.9 Transducer Block

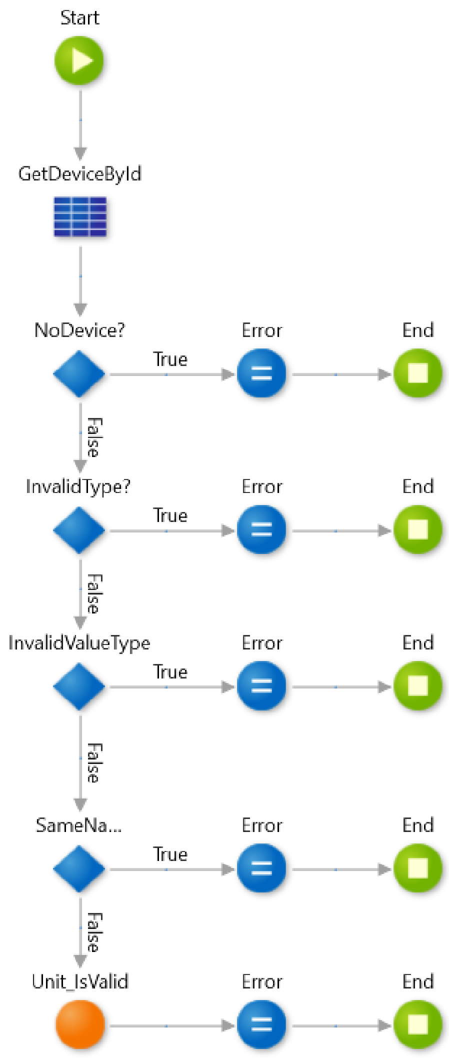
The popup that allows to add and edit transducers contains a block (for encapsulation and maintenance reasons). It receives the `DeviceId` and the `TransducerId` as inputs, as well as 3 aggregates. The first will fetch the `Transducer` given the block input. Finally, the units and the value types (boolean or decimal) will be retrieved by the remaining 2 aggregates. By filling in the details, the action shown in Figure 4.10a is triggered which, if the form details are valid will create the transducer and associate it with the device. An event is sent to the Device Details screen, which will refresh the data action shown previously in Figure 4.7a, on the event handler. The way to insert a unit is by starting to type a unit on the input, which will prompt a dropdown with already existing units that match the typed one. By selecting one of the units, the action in Figure 4.10b is called, which will assign the selected unit to the unit label input, as well as the unit name input. If the typed unit does not exist yet, a novel one is created. Finally, by clicking on the "x" icon, an event is sent to the parent, which will then toggle the boolean variable that controls the popup visibility.

4.10 Communications Data

The entity communications data was the way found to assure future extensibility of the IoT applications to other protocols (note that only MQTT will be tested in this thesis). This is because it contains an `In` and `Out` address for data received in Outsystems or in the device, respectively. These two parameters will most likely be a constant for every communication protocol implemented in the future. In MQTT these addresses will be the topic, but for TCP/IP, for instance, these will be IP:port.

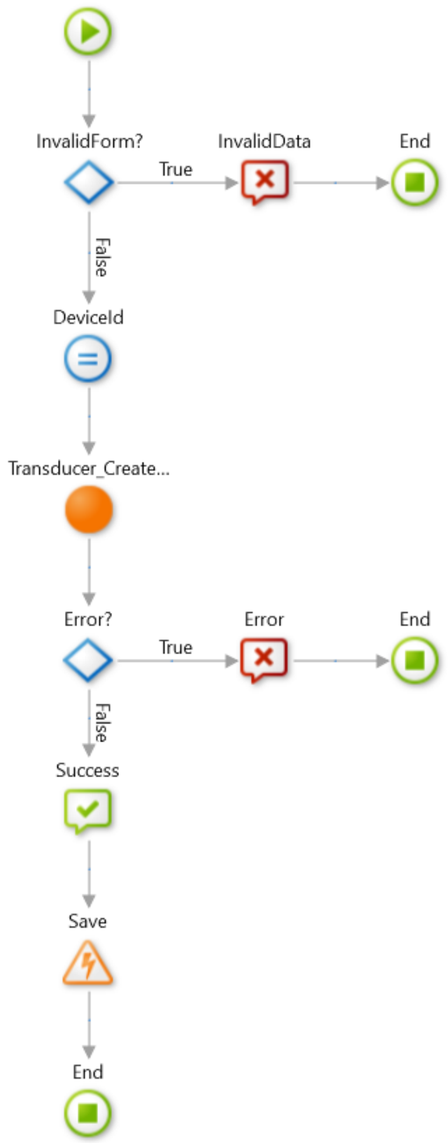


(a)

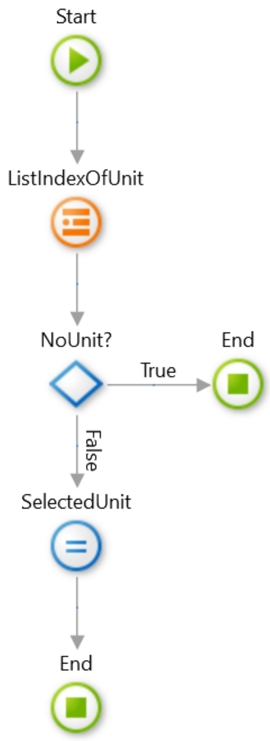


(b)

Figure 4.9: Actions to (a) add a transducer and (b), validate its details.



(a)



(b)

Figure 4.10: Actions to (a) add a transducer (b) to react to the user typing a unit label on the input.

The developed actions' inputs and outputs is shown in Table 4.5 and its implementation in Outsystems can be seen in Figures 4.11a and 4.11b. Starting with the action to validate the addresses, in Figure 4.11b, for MQTT, it will simply verify if both topics start with the character "/" and if they are not empty. If so, no errors are returned. The action to create or update `CommunicationsData` in Figure 4.11a starts by using the previous action to validate the addresses. If no errors are found, it verifies if the provided `Device` identifier is `NullIdentifier()`. If so, this means that a novel device with a novel topic exist and so, a new `PendingInstallation Device` is created calling the action shown in Figure 4.6a. Next, an aggregate is executed to fetch `CommunicationsData` left joined with a `Device`. The aggregate's filter condition assures that the addresses match the input ones and the join condition, that the `Device` is different than the one provided in the action's input, but the same as between both tables in the aggregate's join. Finally, if the topic already exists for a different device, an error is thrown, otherwise, the `CommunicationsData` is created or updated. The resulting identifier is returned by the action.

Table 4.5: Communications data actions.

(a) CommunicationsData_CreateOrUpdate

Inputs
• Topic_In
• UserId
• Topic_Out
• DeviceId

Outputs
• Error
• CommunicationsDataId

(b) CommunicationsData_IsValid

Inputs
• Topic_In
• Topic_Out

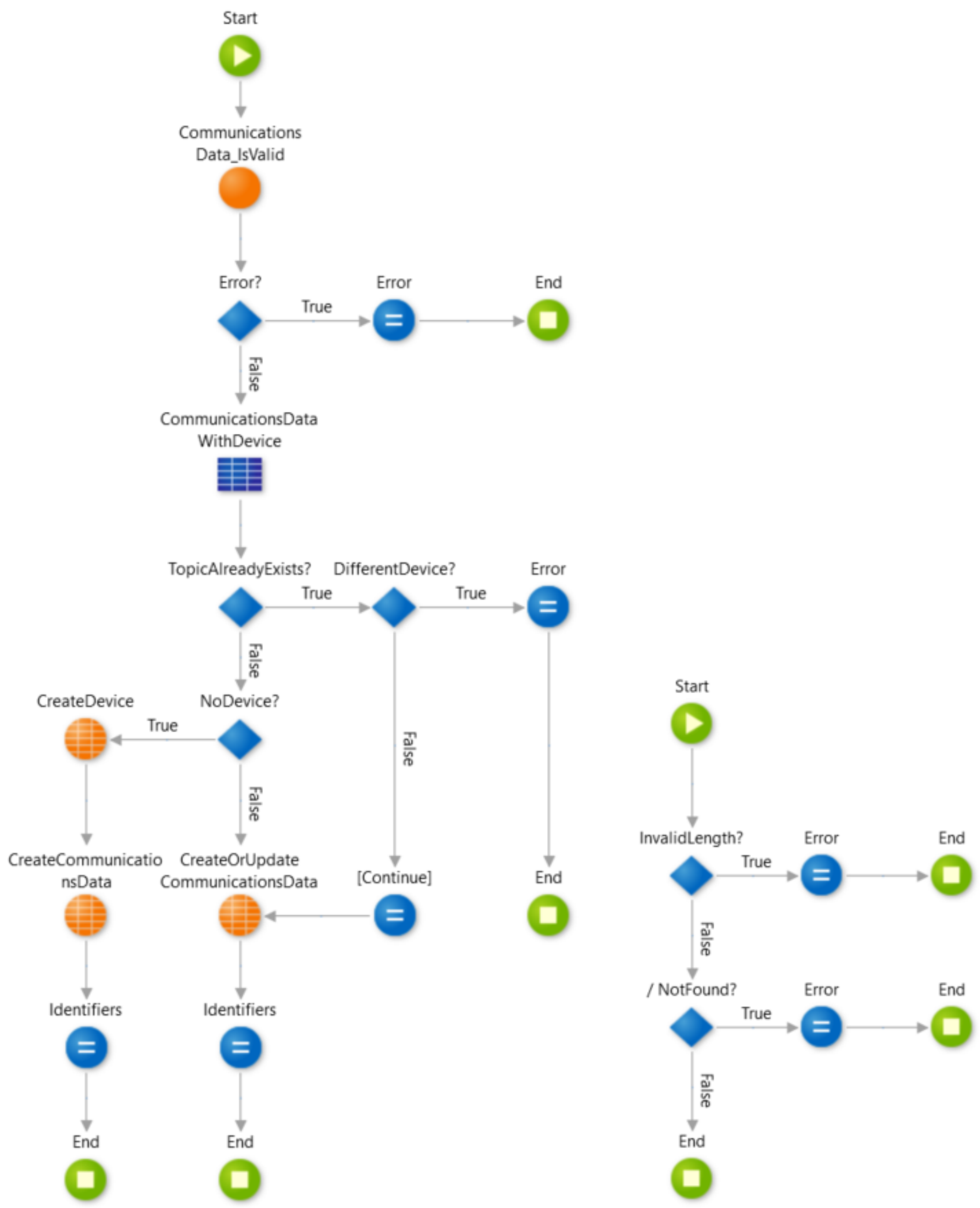
Output
• Error

4.11 Message Payload Registration

Now that both locations, devices, transducers and communications data actions are developed, its time to register the transducer's measurements. The developed actions' inputs and outputs are shown in Table 4.6 and its implementation is depicted in Figure 4.12.

Due to the fact that sometimes, data can be received with unknown topics, after which, a `Device` is created with the status "Pending Installation", this device has no associated transducers and so, the transducers data has to be temporarily stored until the device's information is completed and the status becomes `Active`. Since the information is stored in a temporary table, it can be handled later, asynchronously.

For that, a heavy timer was built and is shown in Figure 4.13a. A heavy timer differs from a regular timer in the sense that it can handle high amounts of data and because of that, the risks of it timing out or throwing errors are higher. Because of that, some considerations should be made on the code [42].



(a) (b)

Figure 4.11: Actions to (a) create or update and (b) assess communications data.

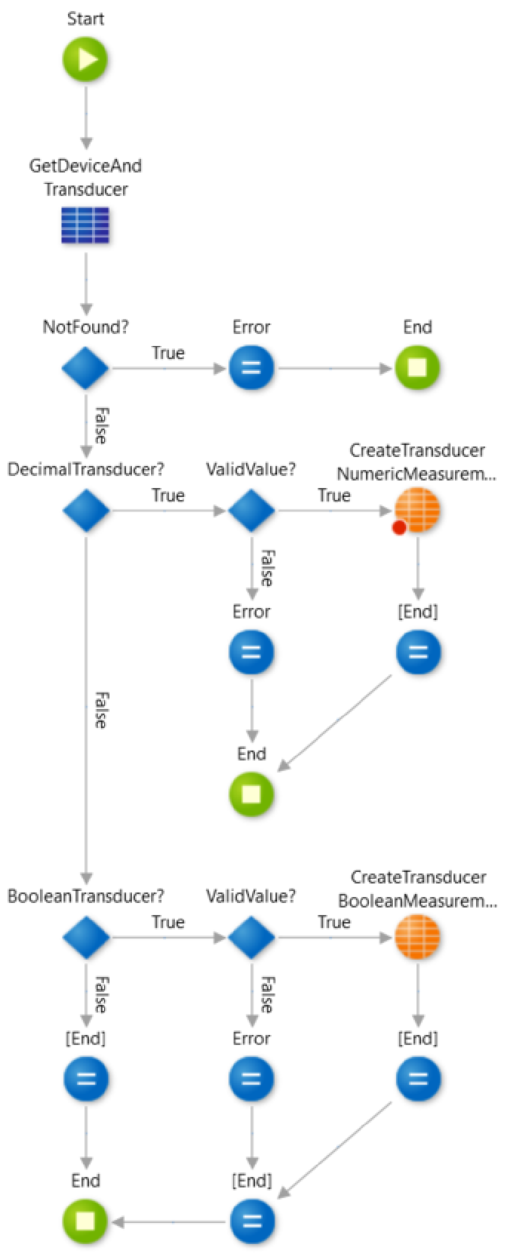
Table 4.6: Actions to register message payloads.

(a) TransducerPayload_Register Action

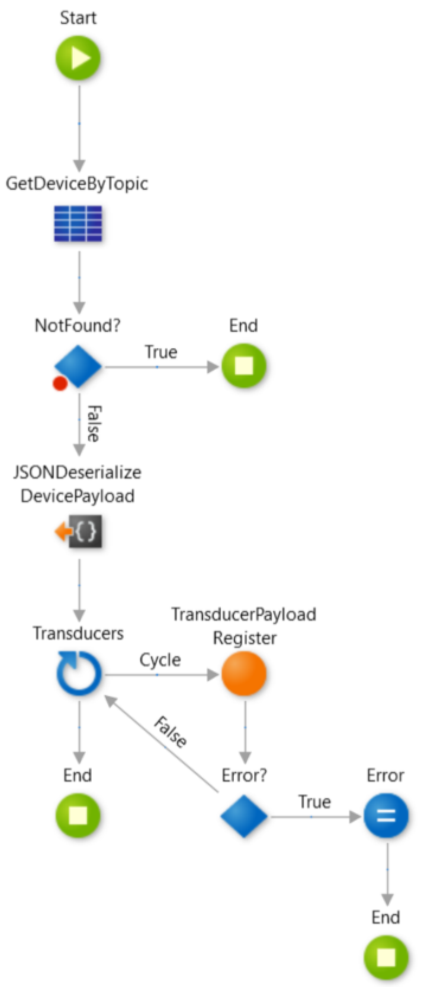
Inputs
• TransducerPayload
• DeviceId
Output
• Error

(b) DevicePayload_Register Action

Inputs
• JSON
• Topic_In
Output
• Error



(a)



(b)

Figure 4.12: Actions to (a) register the transducer payload and (b) register a device message payload.

Firstly, the time out of the timer is stored in a `DateTime` variable. Next, a `Boolean` site property is verifyed to assess if one can proceed. This site property was created because, if a timer keeps throwing errors, it never stops scheduling new executions to try again. This might cause the DB to fill up. Moreover, one verifies if the timer is close to the timeout stored in the beginning. If so, the timer is woke again and the timeout is reset. Otherwise, the code proceeds forward. In this case, all the measurements are retrieved from the temporary table, with its transducers and timestamp. The Aggregate will only fetch 50 records at a time, in order to process data in small batches. If there are records returned by the Aggregate, the measurements are registered and a flag is set on the temporary table indicating that the record was already processed. After processing the 50 records, the code returns to the `CloseToTimeout If` and the process repeats. When there are no measurements, another timer is waken up (see Figure 4.13b), which will execute an SQL query to bulk delete the processed records from the temporary table.

4.12 Dashboard Actions

The inputs and outputs of the actions to create or update and validate a dashboard can be seen in Table 4.7. Given that the validation action ends up being repetitive, only the create or update action will be shown. It is depicted in Figure 4.14a and is relatively simple to understand without further explanations. The action in Figure 4.14b sets a location as default which will make it be the one shown to a user when he opens the application.

Table 4.7: Actions related to dashboards.

(a) Dasboard_CreateOrUpdate

Inputs
• Dashboard
• UserId
Output
• Error
• DashboardId

(b) Dashboard_IsValid

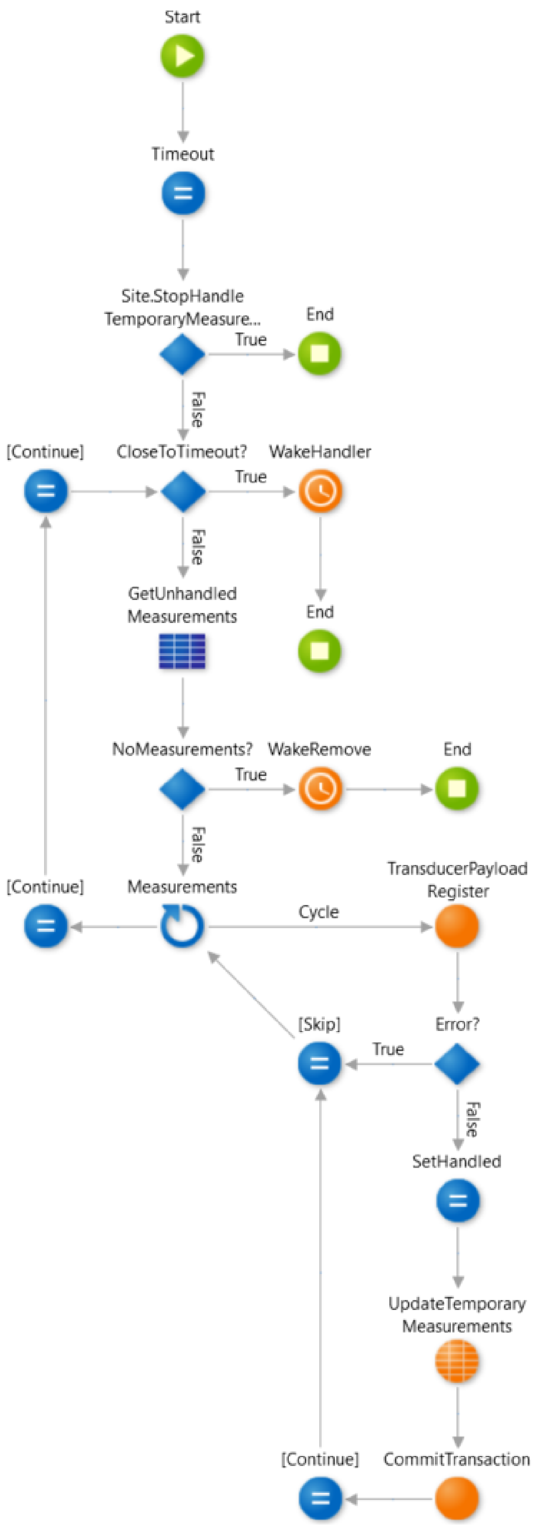
Inputs
• Dashboard
Output
• Error

(c) Dashboard_SetAsDefault

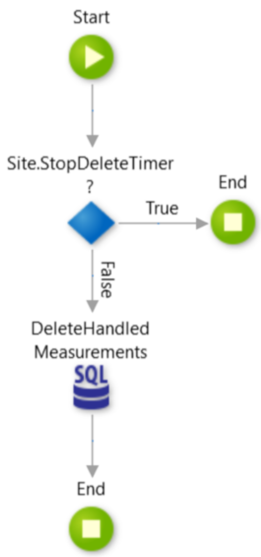
Inputs
• Dashboard
Output
• Error

4.13 Dashboard Detail Screen

This screen is relatively simple and contains an input with the `DashboardId` which will then used to obtain that dashboard. It also contains an aggregate to fetch all locations, which will populate the dropdown. The action to fetch a dashboard is shown in Figure 4.15 and will simple validate the `Form` and call the CRUD action.

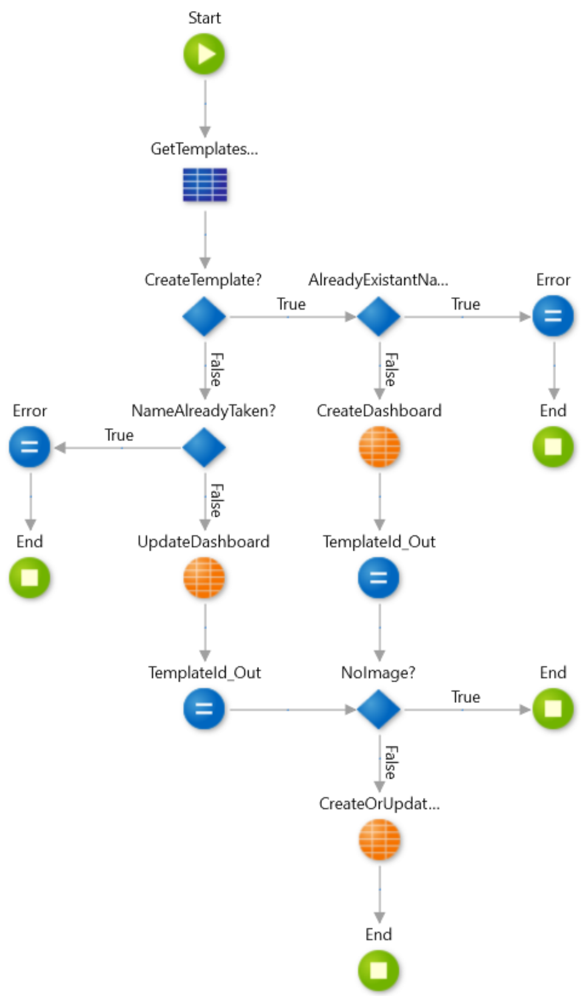


(a)

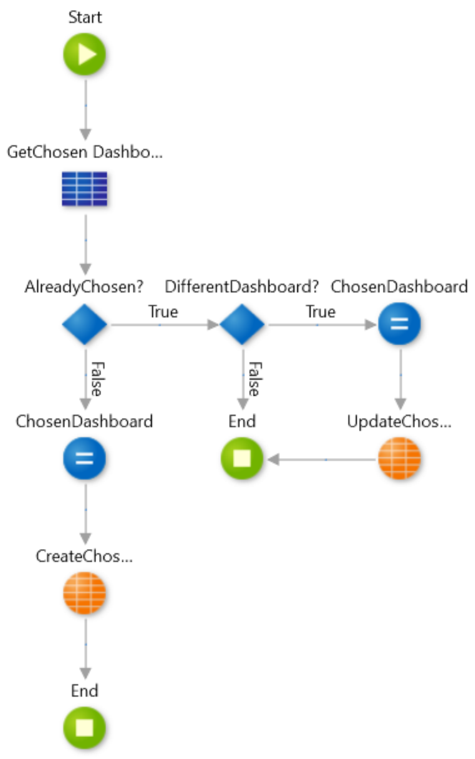


(b)

Figure 4.13: Timer actions to store transducer payloads temporarily, while its devices are on the “Pending Installation” status.



(a)



(b)

Figure 4.14: Action to (a) create or update dashboard and (b) to set dashboard as default.

4.14 Dashboard Main Screen

The main screen in this application contains 3 aggregates to fetch the cards in the dashboard (ordered by the order attribute); to retrieve the chosen dashboard where the user identifier equals `GetUserId()` and inner joined with a `Dashboard`, `Location` and left joined with a `DashboardImage`. Finally, the last aggregate will fetch the user details, such as the name (to present it on the screen).

The addition of a novel card consists of a simple append operation to the result of the `GetCards` aggregate. Moreover, an `OnAfterFetch` event is used for the `GetChosenDashboard` aggregate to refresh the `GetCards` aggregate to get the cards associated with the chosen dashboard.

To cope with the rearranging of cards, an event is associated with the `OnDragEnd` of the component shown in Chapter 5, so that the behaviour caused by the moving of cards gets saved on the server (by calling the `RearangeCards` action). This action is shown in Figure 4.16a, where the Javascript snippet is depicted in Listing 4.1.

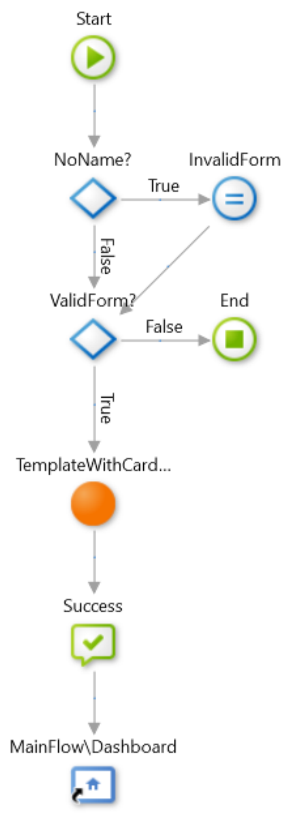


Figure 4.15: Action to save a dashboard.

Listing 4.1: Javascript to return the card identifiers.

```
var cardsList = "";
document.querySelectorAll("[data-card-id]").forEach(
   function(element) {
   cardsList += element.getAttribute("data-card-id") + ",";
   }
);
$parameters.CardIdsList = cardsList;
```

Given that the Dashboards are composed by a set of cards, it makes sense to divide their layout into a grid. Each card can then occupy a set number of cells in that grid, thus making the creation of templates as simple as distributing the cards, one after the other, through the grid.

To achieve this behaviour we made use of CSS Grid which allows developers to define a two dimensional grid on a web page and place all the HTML elements on the desired cells. Elements can span over multiple columns or rows, and we can even specify in which specific part of the grid a certain element should be, as well as leave cells empty. The functioning of the grid can be understood in Figure 4.17, where it can be seen how different card sizes occupy a different amount of cells on the grid.

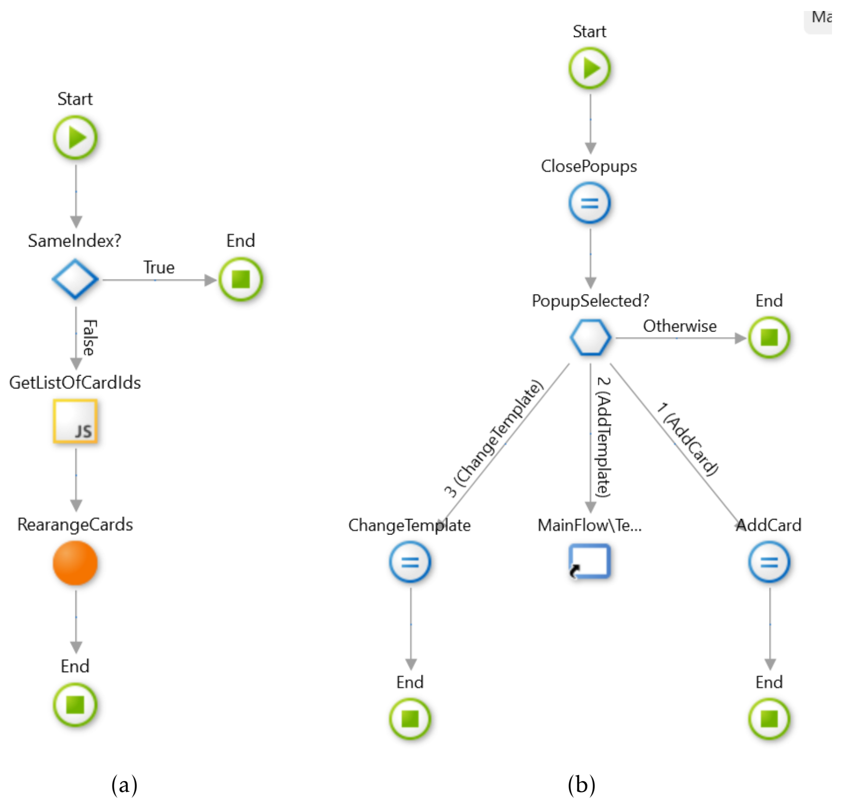
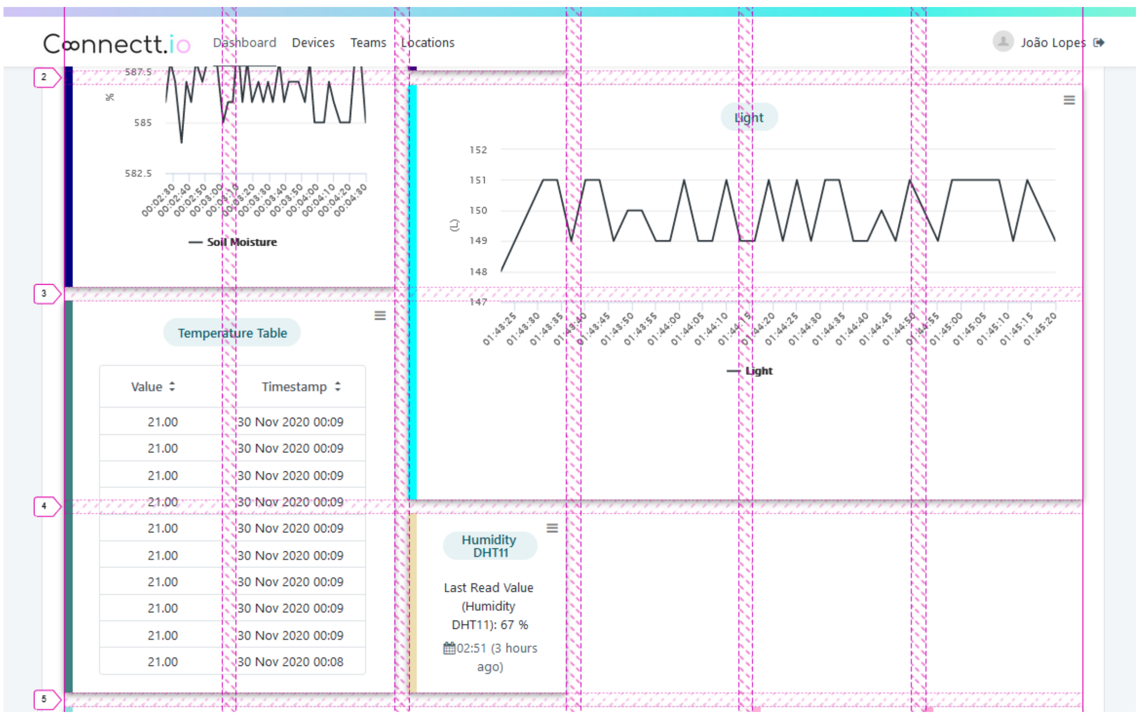


Figure 4.16: On Drag End event handler for the drag and drop component on (a) and action to manage popups on (b).



Value	Timestamp
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:08

Last Read Value (Humidity DHT11): 67 %
02:51 (3 hours ago)

Figure 4.17: CSS Grid to display dashboard cards.

In Outsystems there is already a block that makes use of CSS Grid, making it possible to display a list of Outsystems entities in a grid fashion, so it makes total sense to use this block to display out list of cards. However there is a limitation with the use of this block, as it doesn't allow the developers to specify that an element occupies more than one cell. Since this something the needed for our application, we needed to find a way to allow the user to specify cards with more columns or rows, and still be able to use the ready made grid Outsystems component. The solution for this problem was in creating some CSS classes, that would make use of the grid-row and grid-column attributes. These CSS Grid attributes allow the user to define the number of vertical or horizontal spaces in the grid that an element occupies. They also can be used to define in which cell the element should start. For our purposes the following classes where created:

Listing 4.2: CSS Grid classes to allow cards with different sizes.

```css
.span-2x2 {
   grid-column: span 2;
   grid-row: span 2;
}

.span-2x4 {
   grid-column: span 4;
   grid-row: span 2;
}
```

These two classes, when applied to the elements on our list (with Javascript injection, seen in Figure 4.18 and Listing 4.3) will make them occupy 2 rows and 2 or 4 columns respectively. Elements without any of these classes will then occupy a single cell.

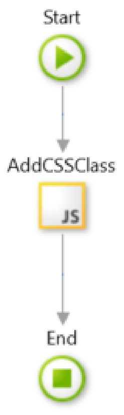


Figure 4.18: Action to add the CSS class to the card.

Listing 4.3: Javascript to add CSS classes to cards.

```javascript
var card = document.getElementById($parameters.WrapperId).parentElement;
var cssClass = "span-" + ($parameters.CardSize === 1 ? "1x1" : "2x" + (
    ↪ $parameters.CardSize === 2 ? "2" : "4"));
card.classList.add(cssClass);
```

With this solution, one can have the best of both worlds. On the one hand we can reuse an Outsystems component that will also have the best compatibility, and on the other, we can still keep the customization of the card sizes.

4.15 Card Actions

The inputs and outputs for the card-related actions are shown on Table 4.8. The create or update action, as well as the validation one are very similiar to the ones already explained in this document. For that reason, only the action to update the order of that card (order in which the card will be shown on a dashboard) will be shown: Figure 4.19. This action will simply get the card from the database and update its order attribute.

Table 4.8: Actions related to dashboard cards.

(a) Card_CreateOrUpdate

Inputs
• Card
• UserId
Output
• Error
• CardId

(b) Card_IsValid

Inputs
• Card
Output
• Error

(c) Card_UpdateOrder

Inputs
• CardId
• Order
Output
• Error

4.16 Card Blocks

To ensure encapsulation and avoid to repeat code, 6 different blocks were created. One generic, containing details such as the title and an if that will include the other 5 blocks depending on the card type (eg. Spline or Table). Then, each card will fetch the necessary data, accordingly. Also, an on parameters change event is used to react to novel data incoming via MQTT via the action `HiveSubscribe_Val`. For the cards containing actuator data, the function `HivePublish` from the MQTT component was used to send data to the broker upon clicking on a “Send” button.

There is another block to add or edit a card, but, since it is similar to the add/edit transducer, its logic will not be explained.

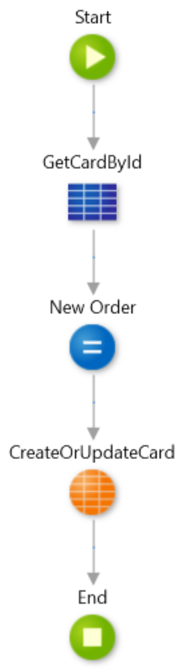


Figure 4.19: Action to change the order of a card in a dashboard.

4.17 Offline Synchronization

To showcase the native mobile application and its capabilities, data from the server was syncronized with the local storage. There are several patterns that can be chosen depending on the use case of the application (if internet failure may happen frequently or not, if read-only or read-write, the size of the data to be synced...).

In this thesis, large amounts of data may be needed to be synchronized at once and only read capabilities are required, since the user should not be able to create dashboards, add devices, etc, while offline. On the other hand, every time a device becomes online, it is important to synchronize the data on the server with the device, so that the user, although is not able to act upon it on the application, can see the transducers data in offline mode. The Read-Only Data Optimized [43], so that the server contains the data that may change over time (master-data), and synchronization will be used to retrieve the transducers data in a read-only and optimized way (as the name of the pattern suggests). This is because the transducers' data might be large in size and optimization is of most importance to avoid to drain the battery and consume mobile data. The way this pattern works is by storing the timestamp of the last synchronization and only retrieve data after it. As it is important not to occupy more storage than required, only data from transducers associated with dashboards should be on the device and if a transducer is deactivated or removed from a dashboard, its data should be deleted from the device.

The Offline Data Sync action is shown in Figure 4.20. This action is triggered by calling another action, Trigger Offline Data Sync, which will execute the former asynchronously.

This way, the application is not locked by this action and the user experience is better as the data is synchronized in the background. To know the stage of the synchronization, events are triggered on the layout block. They can be used to show a loading or to prompt the user to inform him that the synchronization is complete, had an error, or is in progress.

Firstly, on the action, the timestamp of the last synchronization is retrieved and is used as input for the Server Data Sync action, below. This action has as output the transducer's data to delete and to create; and the cards; dashboards; chosen dashboard and chosen dashboard image (only this one to not consume too much storage) associated with the user create them. This behaviour can be seen in the action on Figure 4.20. In the end, the Sync Properties timestamp is updated with the most current server timestamp.

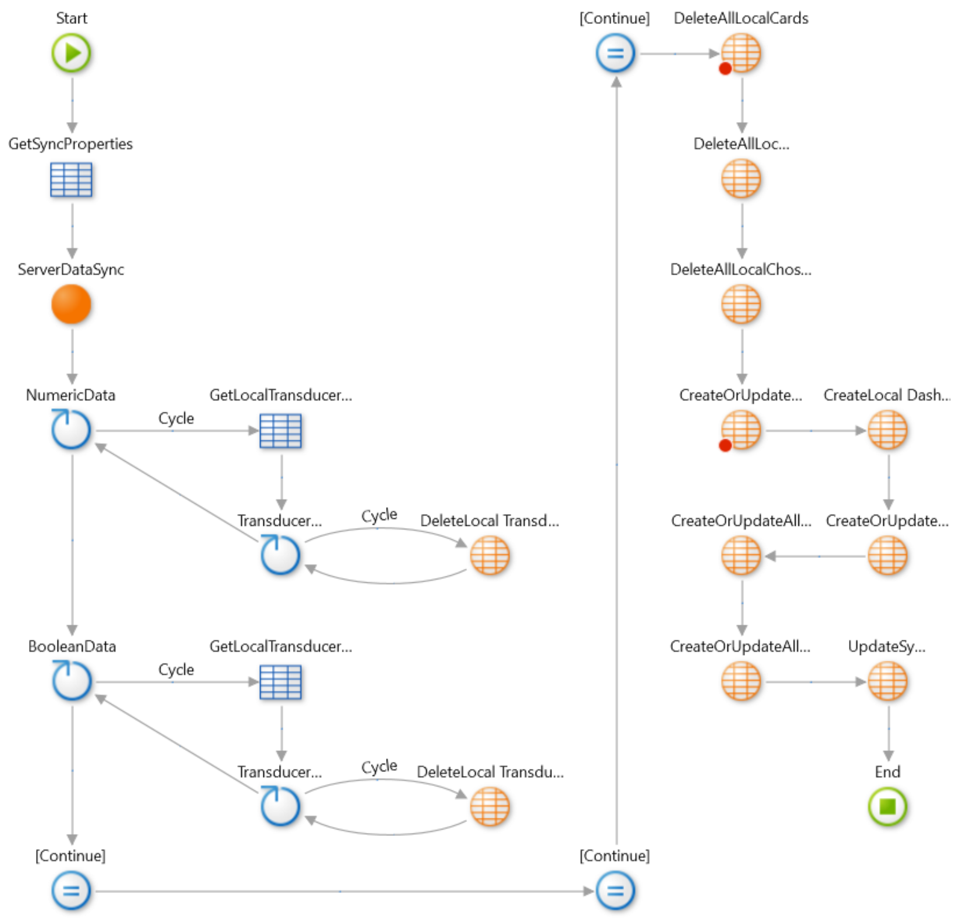


Figure 4.20: OfflineDataSync action, to perform the offline data synchronization.

Moving on to the Server Data Sync, this action, as the name suggests, is a server action that will fetch the data to be synchronized locally. For registered users only, this action will start by storing the current timestamp and fetch the `Dashboards` left joined with the `ChosenDashboard`, `DashboardImage`, `Location` and `Location_User`. After this, if there were results, they will be appended to a local dashboard list, where the Server Id will be the Id from the aggregate. Then, the identifiers will be appended to a string, comma separated, to be used as input in an SQL query, shown in Listing 4.4.

Listing 4.4: SQL query to fetch cards in given dashboards.

```sql
SELECT @NullIdentifier
     , {Card}.[Id]
     , {Card}.[CardTypeId]
     , {Card}.[CardSizeId]
     , {Card}.[Color]
     , {Card}.[Title]
     , {Card}.[IsSensor]
     , {Unit}.[UnitLabel]
     , {Unit}.[Name]
     , {Transducer}.[Id]
     , {Card}.[DashboardId]
     , {Card}.[Order]
     , {Device_Location}.[IsActive]
     , CASE WHEN {Transducer}.[TransducerValueTypeId] = @DecimalValueTypeId THEN 1
         ↪ ELSE 0 END

FROM {Card}
INNER JOIN {Transducer}
    ON {Transducer}.[Id] = {Card}.[TransducerId]
INNER JOIN {Device}
    ON {Transducer}.[DeviceId] = {Device}.[Id]
INNER JOIN {Device_Location}
    ON {Device_Location}.[DeviceId] = {Device}.[Id]
LEFT JOIN {Unit}
    ON {Transducer}.[UnitId] = {Unit}.[Id]

WHERE {Card}.[DashboardId] IN (@QueryIN)
```

The cards are then separated between active and inactive and between its data types and appended to the respective output variables, where the inactive ones will be removed in the `OfflineDataSync` action. Two similar SQL queries are used to retrieve the boolean and decimal measurements which are, again, appended to outputs to be be used in the client sync action.

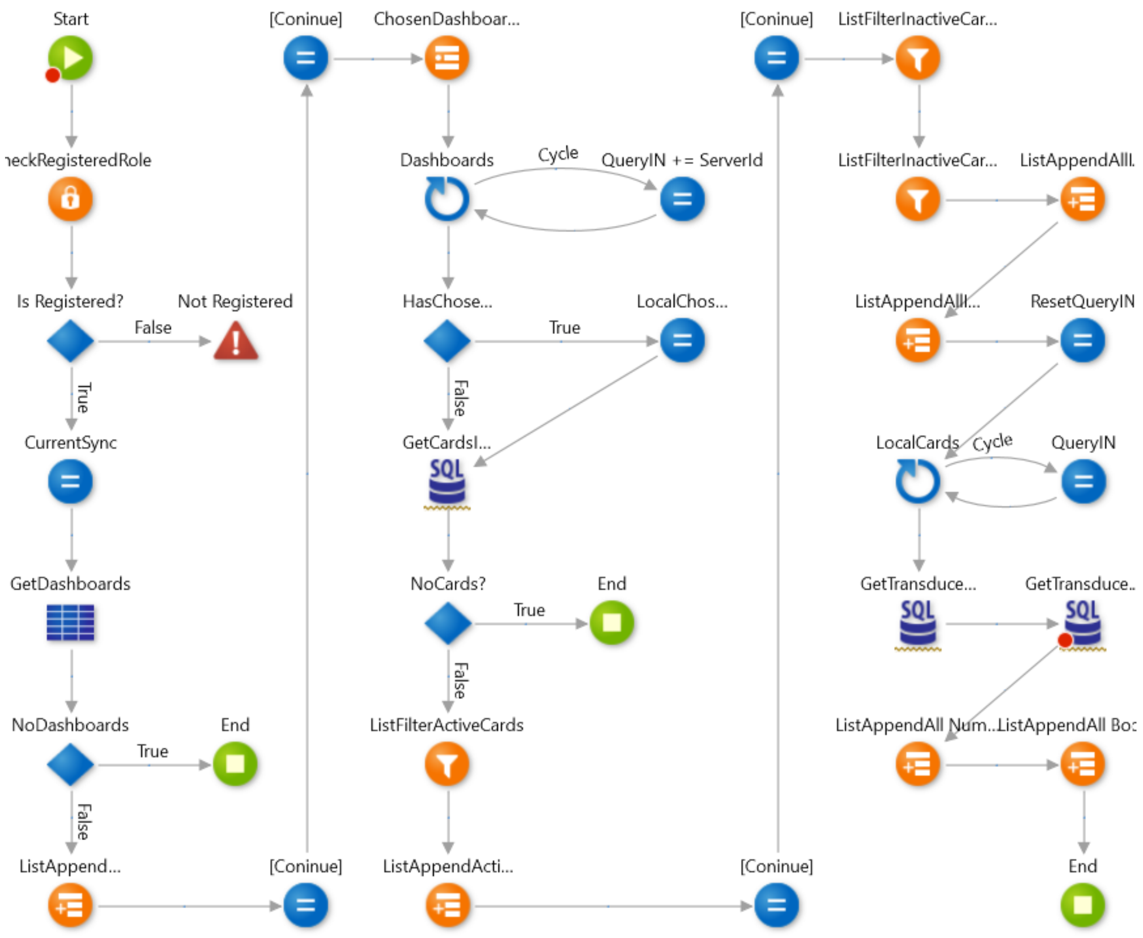


Figure 4.21: Server Data Sync Action

Chapter 5

SortableJS Component

One of the functionalities available on the IoT platform is the possibility for users to create different dashboards and personalize them. For instance, rearranging the order of the different cards on a template is useful, so that the most important information appears first. Essentially, cards, containing plots; tables; numerical values; among others, can be moved around to improve efficiency.

To achieve this, some sort of drag and drop behaviour to rearrange cards is the most viable option, UX-wise. Furthermore, as Low-Code is being used, there might already exist a Forge[1] component that provides drag and drop functionality. The requirements of this component are: capability with the React library, as this technology is being used, as well as the option to save the novel order of the cards and to, given a wrapper (dashboard), to change the order of its childs (cards). After some research, it was found that there was no React component that did this. Therefore, a novel drag and drop component was created and published on the Outsystems Forge. By creating a component, one will make a contribution to the developers' community, since other users will be able to download and reuse this component.

The Drag and Drop (DnD) API was taken into account, avoiding using complex JQuery code, but HTML5 native functionalities instead, with its 7 events that may occur when dragging and dropping an element [44]: **dragstart**, when the user starts dragging an object; **dragenter**, when the draggable object enters a target element (meaning that a drop is allowed); **dragover**, when a draggable element is over a drop area; **dragleave**, when a draggable object leaves a target element; **drag**, that fires constantly when a drag and drop operation is in place; **drop**, fired at the end of the drag and drop operation (this is where one receives the data sent by the draggable element); **dragend**, fired when the user releases the mouse after dropping an object.

[1]https://www.outsystems.com/forge/

5.1 SortableJS Library

After some research, a library called SortableJS[2] was found. There are several reasons to justify the choice for this library, such as the use of pure Javascript as well as the native HTML5 DnD API and does not uses JQuery at all, or other libraries [45] (being more lightweight). Other than this, whereas other libraries do not allow the use of dynamic lists, this library allows that [45], which is a requirement for this thesis, since there will be personalizable dashboards (lists), where cards (items) can be added/remove dynamically, according to the users' preferences and project requirements. Thirdly, whereas the HTML5 DnD does not support touch devices, the SortableJS overcame this problem by obtaining a link to an element by coordinates, after which, that element is cloned and moved after. The `setInterval` function is also used to check if that element is under the finger every 100 ms [45].

The summed-up functionalities of the SortableJS library are, according to [45], and later, to [46], as follows:

- The ability to sort both vertical and horizontal lists;
- The possibility to set the elements to be sorted (CSS selector);
- The ability to have different "drag and drop groups";
- The possibility to configure groups, *eg.*, to allow items from the list *A* and *B* to be moved to *C* and to deny items to be dragged between *A* and *B*.
- The possibility to set what element(s) can be dragged;
- The possibility to add a class to the moved element;
- The existence of `onAdd`; `onUpdate`; and `onRemove` events;
- The list can change dynamically (possibility to edit, add and delete elements);
- The possibility to add animations when moving items;
- Scroll improvements, were it is now possible to scroll a list first if it was in `overflow`. There is now a property to enable auto-scrolling; the how much close to the border one must be to activate scrolling; and the scrolling speed;
- It is possible to disable sorting.

[2]https://github.com/SortableJS/Sortable

5.2 Development

Now that the main functionalities of this library were mentioned, its time to implement this library in Outsystems and create a reusable component, which will then be published in the Outsystems Forge. First, its important to define what is a component. According to [47], it "is a reusable object that speeds up application creation and delivery (...) It should be easy to use and understand, and the focus should be main or common use cases". In this case, a library component will be created, containing encapsulated Javascript code (inside blocks and actions).

The first step is to install the library (`npm install sortablejs --save`) to access its minimized Javascript code. After that, one needs to create a new module on the Outsystems environment and initialize a Reactive web application. Forst, the theme and the default screens/WBs were removed, since they are not required. Next, a new WB was created and the MIT License of the library was included. Following, a new Javascript script was added to the module, called `SortableJS`, and inside it, the minimized library Javascript code mentioned *a-priori* was added. After studying the library and inspecting several of its demos ([45], [46], [48], [49], [50]), it was seen that the library received a "parent" to be initialized, after which, all of its children could be moved around - dragged and dropped. With this in mind, a new Web Block was created, also named `SortableJS`.

It is now required to enable this block to be reused in several applications, by several users, with different types of items: list items; table rows; containers, among others. For that, an Outsystems widget called `placeholder` was introduced on the WB. The objective of this widget is to allow different other widgets and components to be placed inside it, upon the initialization of the web block. For instance, if a `container` was inserted inside the placeholder, what one would see after using the developer tools to inspect the code would be a `<div></div>`. With this in mind, if a list with `<ul><li>(...)</li></ul>` were to be inserted inside the placeholder, the intended behaviour would be for `<ul></ul>` to be the parent, with its childs, `<li>(...)</li>`. The solution found is for the wrapper to be specified with a CSS-selector (see Listing 5.1) where a CSS query selector was required to select the list wrapper.

Listing 5.1: Drag and Drop Forge Component Javascript Code (initial version).

```javascript
var wrapper = document.querySelector($parameters.WrapperQuerySelector); // eg. "div >
    ↪ p"

if (wrapper)
    Sortable.create(wrapper, $parameters.OptionsJSON);
else
    console.log("There was an error retrieving wrapper!");
```

The problem with this solution was that the options JSON was not being recognized, as it was not being serialized. The final solution is depicted in Listing 5.3.

Listing 5.2: SortableJS Outsystems component JSON input example.

```
{
  group: "name", // or { name: "...", pull: [true, false, 'clone', array], put: [true
    ↪ , false, array] }
  sort: true, // sorting inside list
  delay: 0, // time in milliseconds to define when the sorting should start
  (...)
}
```

Listing 5.3: Drag and Drop Forge Component Javascript Code (final version).

```
function merge(dest, src) {
   for(var key in src) {
      dest[key] = src[key];
   }
   return dest;
}

function onDragEnd(e){
   var itemEl = {
      'from': e.from.getAttribute('data-list-id'), // previous list
   'to': e.to.getAttribute('data-list-id'), // target list
   'oldIndex': e.oldIndex, // element's old index within old parent
   'newIndex': e.newIndex, // element's new index within new parent
  }

   $actions.TriggerOnDragEnd(JSON.stringify(itemEl));
}

var list_wrapper = document.querySelector($parameters.WrapperQuerySelector), // eg.
    ↪ "div > p"
   options = JSON5.parse($parameters.OptionsJSON),
   event = {
      onEnd: function (e) {
      onDragEnd(e);
      }
   };
   options = merge(options, event);

if (list_wrapper){
   list_wrapper.setAttribute('data-list-id', $parameters.ListName);
   Sortable.create(list_wrapper, options);
}
else
   console.log("There was an error retrieving wrapper!");
```

Some improves were done in Listing 5.3. The first and most important can be seen in line 20 and that is the JSON parsing. Next, for the user to be able to see the changes that the drag and drop operations caused in an Outsystems variable, the JSON input can now also include an event function. This was what was done from rows 8 to 14 and 21 to 26. It can be seen that a new function, `merge(dest, src)` was created to append the event to the remaining JSON input. Furthermore, the solution in `onDragEnd(e)` is also quite elegant. Its different steps are explained below:

1. The `OnDragEndElement` structure was created. It contained two `Text` attributes for the target and previous drag and drop operation's list identifiers, `To` and `From`; and two `Long Integers` for their indexes, `OldIndex` and `NewIndex`, respectively;

2. An `OnDragEnd` event was created for the `SortableJSList` WB, containing variable of type of the aforementioned structure as input;

3. A client action was also created on the Outsystems WB that receives a JSON input of the type of `OnDragEndElement` and inside the action, this JSON is deserialized onto the structure and the event mentioned *a-priori* is triggered, sending the variable received as an input on the action.

What remains unknown is when is the client action called, to trigger the event. The answer relies in Listing 5.3 rows 8 to 14, where the client action JSON input is built, and the action is called. This action is shown in Figure 5.1, where the JSON data from the event in Listing 5.3 is deserialized to an Outsystems structure and sent via an event, so that it can be accessed on the screen/WB were the SortableJS placeholder is being referenced in. The structure and JSON contain information about the sender and receiver wrappers, `To` and `From`, as well as the old and new index, which can be associated with either list.

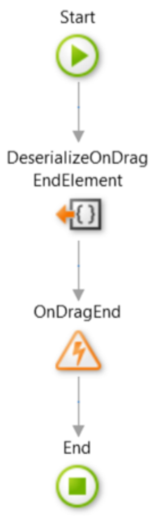


Figure 5.1: OnDragEnd SortableJS event handler Outsystems action.

Chapter 6

Prototype

After building the platform in both web and mobile versions, the applications need to be fed with data. As this is an IoT-related thesis, the data is provided from an Arduino, which will communicate with the platform through MQTT.

6.1 Arduino WiFi Module Configuration

For the Arduino to connect to a wireless network through the 802.11 protocol, the WiFi module ESP8266 ESP-01 will be used. It's range of operation is approximately 91 m; the operating voltage is 3.3 V (VCC); and it contains two serial communication pins: RX and TX. Given the aforementioned operating voltage, a voltage divider is required to cope with the 5 V of the Arduino, to lower it to the 3.3 V of the ESP module. So, a 2 kΩ resistor will connect to GND and 1 kΩ will connect the EN (enable) pin to VCC. The circuit ended up as depicted in Figure 6.1. In Figure 6.1, the ESP-01 pins are the following: the top-left one is the TX; below the latter, is the EN; at the bottom is VCC; at the right of VCC is RX; and at the top of the RX pin is the GND pin.

After connecting the Arduino to the module, it is required to assess if the later is working and at which baudrate. For that, a code snippet that included the SoftwareSerial library was uploaded to the Arduino. The ESP-01 comes pre-installed with AT firmware, which means that the module can be programmed using AT commands [51]. The Arduino RX and TX pins were set when creating the SoftwareSerial instance. The first step is to open the serial monitor on the Arduino IDE and send the AT command "AT" which "(...) will check if the module is connected properly and its functioning, the module will reply with an acknowledgment" [51]. However, this is not what happened, since no acknowledgement was received. Given this, the baudrate of the SoftwareSerial was changed from 9600 bps to 115 200 bps. The test was repeated and the aknowledgment was received.

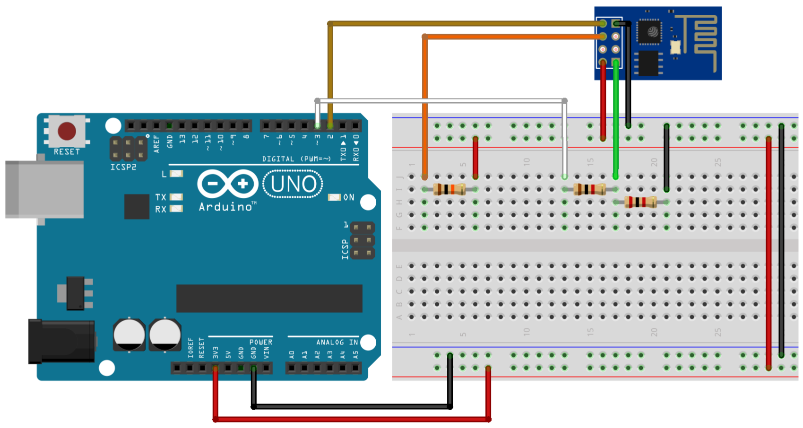


Figure 6.1: Arduino's ESP8266 ESP-01 schematic. (designed in `fritzing.org`)

The SoftwareSerial library that will further be used only works with the 9600 bps baudrate. Therefore, another AT command is required: "AT+UART_DEF=9600,8,1,0,0", which will change the default ESP-01 baudrate to 9600 bps. The baudrate of the SoftwareSerial was changed again to 9600 bps and the aknowledgment was received upon sending the command "AT". The module is ready to be used.

It is now time to implement the code that will be used to connect the Arduino to a network and send/receive data through the MQTT protocol. The developed code can be seen in the Listing C, in Appendix C. The most important things to notice at this moment in the aforementioned listing is that the SoftwareSerial, WifiEsp and PubSubClient libraries are being used. The first, was already used previously and it is required to allow the use of other Arduino's digital pins using software, in this case, the pins 2 and 3 as the Arduino's RX and TX, respectively. On the other hand, the WifiEsp library, as the name indicates, allows the Arduino to connect to the Internet. For this library, it is required to indicate the network Service Set Identifier (SSID), as well as the password. Next, the ESP module will be initialized and attempt to connect to the Access Point (AP).

When first testing the code, some problems arised, as the serial monitor showed a message saying "»> TIMEOUT »> (...) Warning: Unsupported firmware. Connecting to AP". What followed this message was the ESP attempting to connect to the AP unsuccessfully. After doing some research, it was found out that the firmware version was not supported by the WiFiEsp library. To flash the module, the solution relies on changing the connections as shown in Figure 6.2.

The only ESP-01 pin that is now being used, but was not yet mentioned *a-priori* is the GPIO0, which is a general input/output pin. Next, a software called ESP8266 Flasher was downloaded, as well as a new firmware *.bin*-file. On the downloaded software, the *.bin*-file was uploaded and the Arduino's COM port selected. After this, by clicking on the Download button, the novel firmware will be written on the ESP module. By repeating the test of trying to connect to the AP, it was now successful.

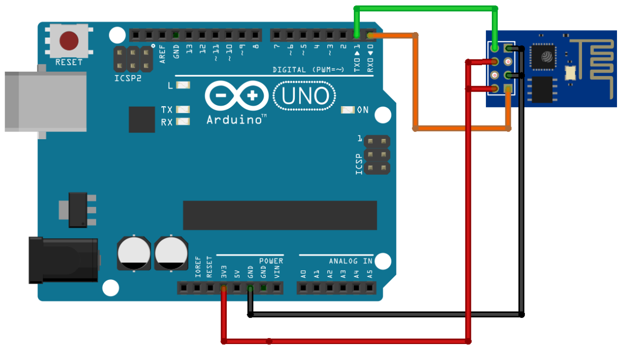


Figure 6.2: Arduino's ESP8266 ESP-01 schematic to flash new firmware. (designed in `fritzing.org`)

Finally, the PubSubClient library provides a client to publish/subscribe messages through the MQTT protocol. Therefore, the first required steps are to provide the MQTT server url; port; username; and password. Those were previously retrieved, upon the creation of the server instance. Next, the server parameters will be set on the client's instance and the module will attempt to connect to the broker. By uploading the code to the Arduino, one could confirm that the connection was successful, by verifying the output on the Serial Monitor.

The next step is to confirm if data can already be sent and received from/on the Arduino, respectively. For that, after creating the connection, one can subscribe a topic called "inTopic" and publish a topic called "outTopic". By using a chrome extension called MQTT Lens, it is possible to verify if the message are being well sent/received. After some minor bug corrections, this was indeed the case.

6.2 Integration with the MQTT Protocol

The first thing one needs is an MQTT broker. For this, CloudMQTT[1] was used, as it has a free plan. After creating the broker instance, one needs to save the server url; the username; the password; and the port. These are required for both the device and the IoT platform to connect with the broker.

Next, an MQTT client is required. The Forge (presented in Section 2.4.3) has an MQTT client component available with both a web[2] and mobile[3] MQTT client versions. These allow to connect/disconnect to a broker; publish to a topic; and subscribe a topic. The aforementioned component (its web version) can then be used in an Outsystems web application for it to be always working as an MQTT Client, being connected to the broker and subscribing to its topics, as well as store data in the Outsystems database. This client, however, will not be using the React technology, but the traditional web instead, as the forge component is not compatible with React.

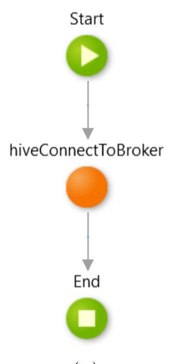
[1] `https://www.cloudmqtt.com/`
[2] `https://www.outsystems.com/forge/component-overview/1174/mqtt-web-client`
[3] `https://www.outsystems.com/forge/component-overview/1944/mqtt-mobile-client`

First, a web screen was created with a Preparation that calls the action `hiveConnect` to connect to the MQTT broker server upon giving following details: Host; Port; ClientId; Username; Password; Timeout; and if it is using the SSL certificate. Also, the WB `mqttJS/`
`↪ HiveMQ` was included to load the Javascript library of the MQTT client. By doing some debug on the browser's console, one could verify that the connection was not being successful - the broker was not being found. This was because a browser only supports MQTT over websockets. After doing some research, it was found that it was required to change the port, previously with the number 1*xxxx*, replacing it with the websockets' port, changing the first digit to 3, becoming 3*xxxx*. By repeating the tests on the Javascript console, one could see that the connection was now successful.

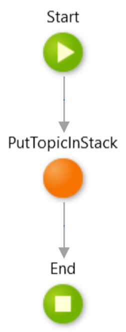
Moreover, another WB, `mqttJS/SubscribeEvent`, is required, as well as to provide the topic to subscribe, the Quality of Service (QOS), and the handler (Outsystems action) as inputs. Also, as seen in Listing 6.2, a list named `queue` was declared. This variable can be accessed in the scope of the screen and the need for it relies on the fact that `mqttJS/`
`↪ SubscribeEvent` works via synchronous callbacks, which means that data can be lost, as incoming data might not be processed if another message is being handled at the time. Therefore, a callback has to be as fast as possible and by using this variable, the data is appended to a stack, temporarily, and handled later. In Figure 6.3 it is possible to see the Preparation that will connect with the message broker (Figure 6.3a) and the action to add transducers' messages to the stack every time a novel MQTT message arrives (Figure 6.3b). Listing 6.1 contains the callback script that will be executed every time a new message arrives at the client. This snippet will append the message content (topic; timestamp and payload) to the `queue` declared on the screen. After this, a hidden link will be clicked, which will call the action seen in Figure 6.3a. The snippet in Listing 6.2 also clicks on this hidden link every 3 hours, assuring that every message is handled. The action that is called when the hidden link is clicked, named "WriteInDatabase", is shown in Figure 6.4. It starts by retrieving the last message from a screen input and split it into topic; timestamp and payload. An action from the Device_CS module is then called, which will assess the information and write it onto the database. All of the developments in this section were included in the MQTT_CS module.

Listing 6.1: Javascript snippet to write payload in database.

```
console.log('Topic Received');
queue.push('"+EncodeJavaScript(mqttMessage.topic)+"'+ ':' + '" + EncodeJavaScript
    ↪ (mqttMessage.timestamp) + "' + ':' + '" + EncodeJavaScript(
    ↪ mqttMessage.payload) + "');
console.log('Queue: ' + queue);
document.getElementById('" + WriteInDatabaseLink.Id + "').click();
```



(a)



(b) a

Figure 6.3: MQTT Client actions with (a) containing the screen Preparation that will connect with the MQTT message broker and (b), to insert transducer's data onto the stack "queue" via Javascript.

Listing 6.2: Javascript script included in client's web page.

```
<script>
    var queue = [];
    var intervalID = window.setInterval(writePendingData, 10800000);

    function writePendingData() {
        document.getElementById('" + WriteInDatabaseLink.Id + "').click();
    }
</script>
```

Regarding Figure 6.4, firstly, the message data is retrieved from the `queue` with the snippet depicted in Figure 6.3. In the same snippet, the date is inserted in an auxiliar input. After this, one can access the variable through the input. Next, the topic; timestamp and message payload are split and sent to an action in the Device_CS module, which allows this action to be reused in all protocols modules to be developed in the future. The action is shown in Figure 4.13, in Section 4.11. After this, the input variable is emptied and the hidden link is clicked again to assure that no message is left unprocessed.

Listing 6.3: Javascript snippet to retrieve message data from queue.

```
    if(queue.length != 0)
        document.getElementById('" + MessageInput.Id + "').value = queue.shift();
```

6.3 Prototype Schematics

In this section, sensors and actuators are connected to an Arduino to showcase the devices and transducers utility. Without further ado, Figure 6.5 shows the designed schematic.

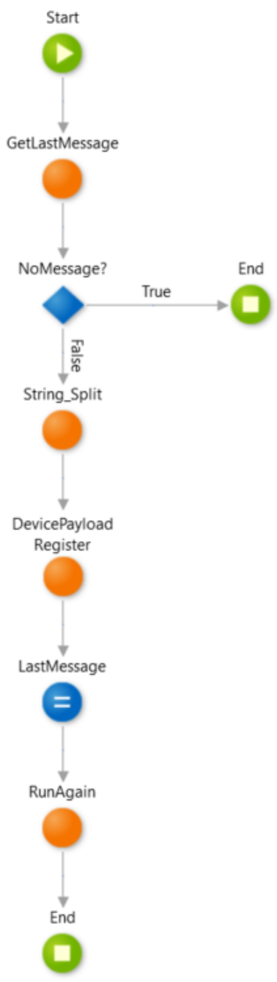


Figure 6.4: Action to write MQTT message content in the database.

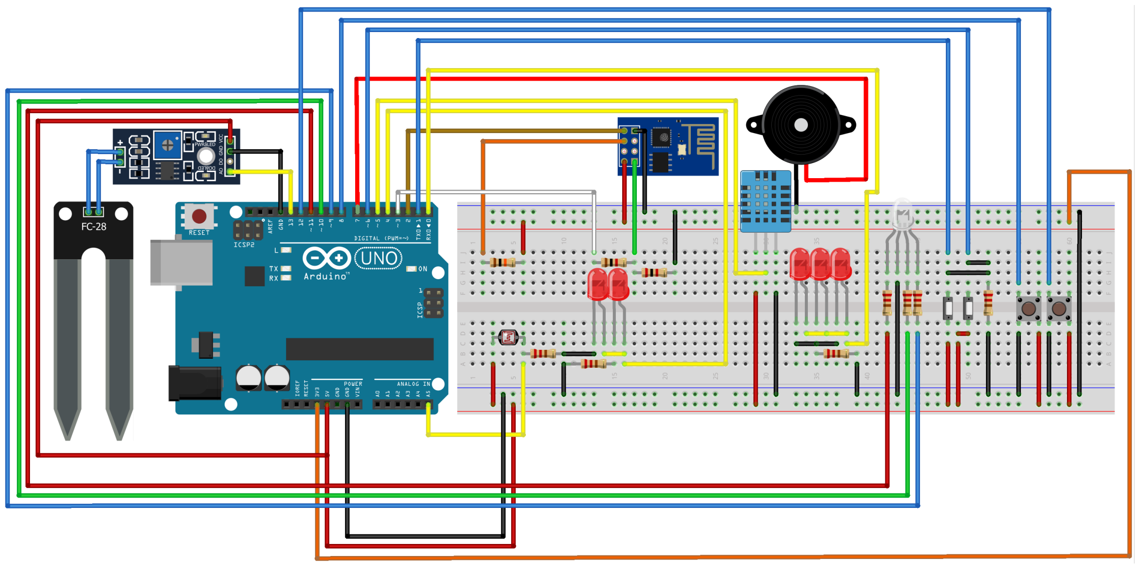


Figure 6.5: Hardware to showcase devices and transducers' platform usage. (designed in `fritzing.org`)

The transducers are separated into 3 virtual devices (in 2 different locations), as shown in Table 6.2 and Table 6.1. Each button sends a `boolean` simulating the activation/deactivation of something, such as a water pump. To measure the air humidity and temperature, the DHT11 sensor was used. Moreover, [52] the soil moisture sensor is connected to an amplifier/Analog-Digital (AD) circuit which is then connected to the Arduino. The VCC of the amplifier is connected to 3.3V and the analog data pin is connected to the $A0$ pin, on the Arduino. The amplifier pin that is not connected is the digital pin, which is not being used as one is only interested in getting analog data. The Light-Emitting Diode (LED)s are considered actuators, that can be turned on/off. With them, one can observe an actuator's status being changed from the platform. Moreover, a piezo speaker will function as an alarm and can be actuated from the platform as well. A Light Dependent Resistor (LDR) is used to retrieve luminosity values from the environment.

Table 6.1: Transducers in Location (T).

Device *A*

Transducer	**Type**	**MQTT Topic** (in/out)	
Piezo Speaker	Actuator	←	/outsystems/piezo
		→	/device/piezo
LDR (Photoresistor)	Sensor	←	/outsystems/ldr
		→	/device/ldr
Button (3)	Actuator	←	/outsystems/button3
		→	/device/button3
Button (4)	Actuator	←	/outsystems/button4
		→	/device/button4

Table 6.2: Transducers in Location (L).

Device *B*

Transducer	**Type**	**MQTT Topic** (in/out)	
Moisture Sensor	Sensor	←	/outsystems/moisture
		→	/device/moisture
Button (1)	Actuator	←	/outsystems/button1
		→	/device/button1
2 LEDs (1)	Actuator	←	/outsystems/leds1
		→	/device/leds1
RGB LEDs	Actuator	←	/outsystems/rgb
		→	/device/rgb

Device *C*

Transducer	**Type**	**MQTT Topic** (in/out)	
DHT11 (Temperature)	Sensor	←	/outsystems/dht11/temperature
		→	/device/moisture/dht11/temperature
DHT11 (Air Humidity)	Sensor	←	/outsystems/dht11/humidity
		→	/device/moisture/dht11/humidity
3 LEDs (2)	Actuator	←	/outsystems/leds2
		→	/device/leds2
Button (2)	Actuator	←	/outsystems/button2
		→	/device/button2

Chapter 7

Validations

The current chapter has the purpose of proving that this dissertation's thesis' IoT platform is indeed doable using a LCS to develop it. The multiple functionalities of the applications, both web and mobile, will be showcased and validated. The chapter is structured as follows: in Section 7.1, a use-case scenario to test the IoT applications, related to agriculture will be introduced and its usefulness, shown; in Section 7.2, the users' functionalities will be depicted; Section 7.3 shows the features related to locations; in Section 7.4, some devices will be added to those locations; finally, in Section 7.5, the dashboard funcionalitites will be demonstrated; in Section 7.6, the Forge component, SortableJS is also shown being used and further, the success of its implementation, discussed; Section 7.7 shows the success of the offline synchronization capabilities on the mobile application; final discussions regarding the overall success of this thesis will take place in Section 7.8.

7.1 Precision Agriculture Use Case

After developing the IoT platform, it is important to showcase a real life scenario, solving today's problems. In this sense, as the world's population has been increasing over the past few years, with no signs of stopping, better solutions for the use and sharing of natural resources are required, so that sustainability can be achieved [53]. One of these solutions relies on optimizing agriculture (one of the world's most important industries as it is present in roughly 34% of the total land area of the planet [54]), since the aforementioned resource-demands are causing the growth of industrial farming and consequently, scarcity of natural resources, negatively impacting the nature (habitat loss and degradation) and the societies' well-being. Optimizing agriculture leads to the concept of PA, also known as smart farming.

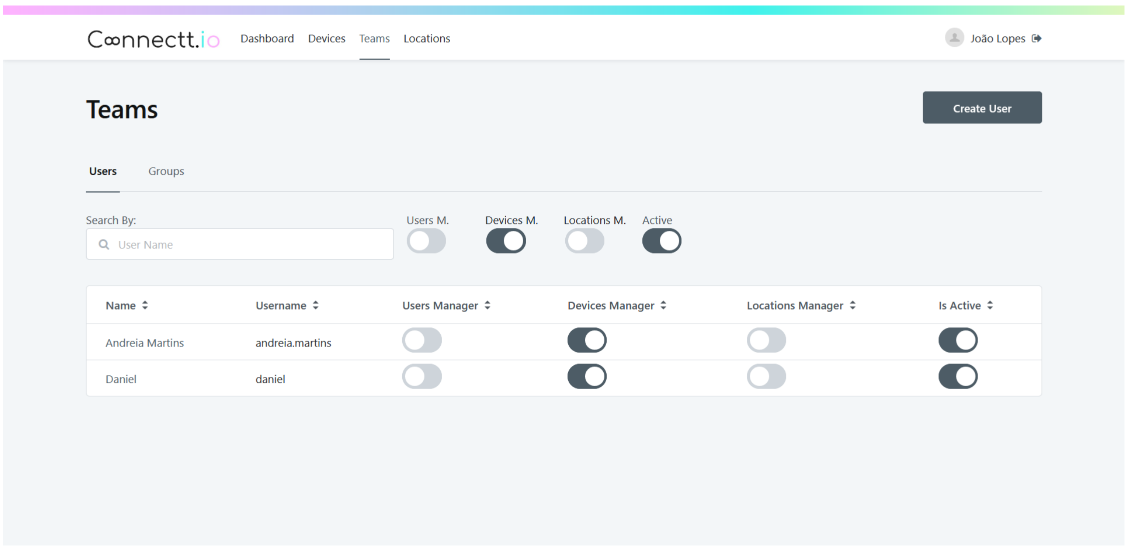
PA is defined by Plumb [55] as “(...) a management strategy that uses information technologies to bring data from multiple sources to bear on decisions associated with crop production (...)”. In other words, PA is a method in which farmers optimize the amount of fertilizers, pesticides, water [56], and others, in targeted areas to maximize the crop yield and quality, and to minimize pests, diseases and waste of resources. This means that different regions of a field may be dealt with in different ways since they present different characteristics with respect to the soil, sun exposure, irrigation, among others.

One of the ways to increase the production and, simultaneously, decrease the man-power relies in IoT, so that farmers can keep track of the state of their fields, create dashboards, optimized to different fields and users and manage the field remotely.

For the following sections, it may be assumed that the users of the IoT application, such as farmers or supervisors, can have little to no experience with the use of web or mobile (IoT) applications. An administrator will be responsible to manage teams and associate those with roles and locations and device administrators can create devices.

7.2 Teams

To test the users’ USs seen in Section 3.2.1, the administrator will create some users on the web platform. In Figure 7.1, a screen allows to visualize users (if the Users tab is selected), search them by name and filter them by group, as well as to change the roles of users within the screen. By clicking on a user, a popup will open, allowing to edit the user and add him to groups. Another tab in the same screen, in Figure 7.2, allows to list groups and change the roles of those groups. By clicking on the group, a popup will open to edit the group and add users to this group.



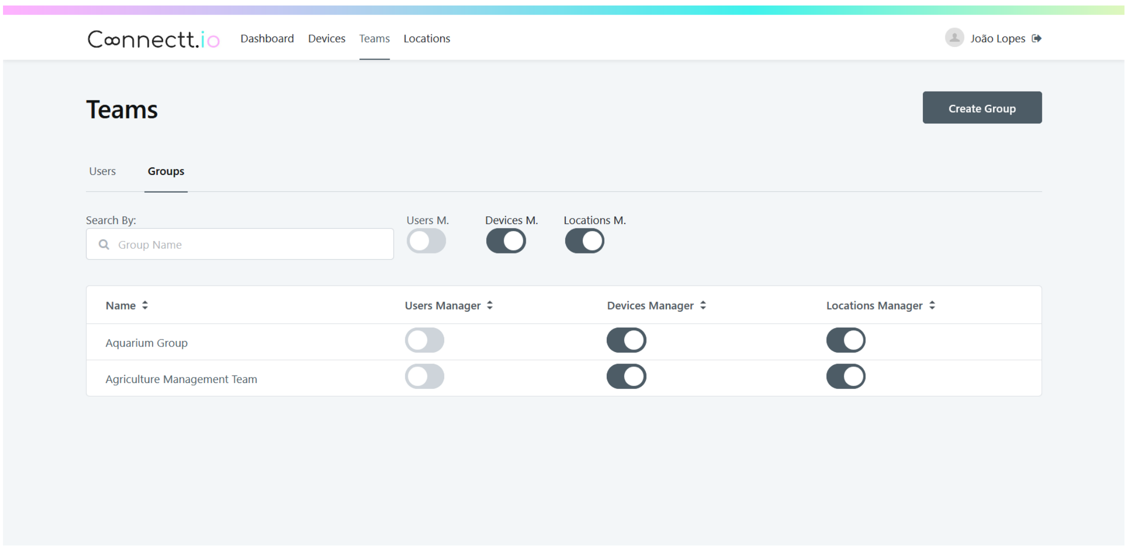
Teams

Search By: User Name

Users M. ○ Devices M. ● Locations M. ○ Active ●

Name	Username	Users Manager	Devices Manager	Locations Manager	Is Active
Andreia Martins	andreia.martins	○	●	○	●
Daniel	daniel	○	●	○	●

Figure 7.1: Users list screen to manage users and its roles.



Teams

Search By: Group Name

Users M. ○ Devices M. ● Locations M. ●

Name	Users Manager	Devices Manager	Locations Manager
Aquarium Group	○	●	●
Agriculture Management Team	○	●	●

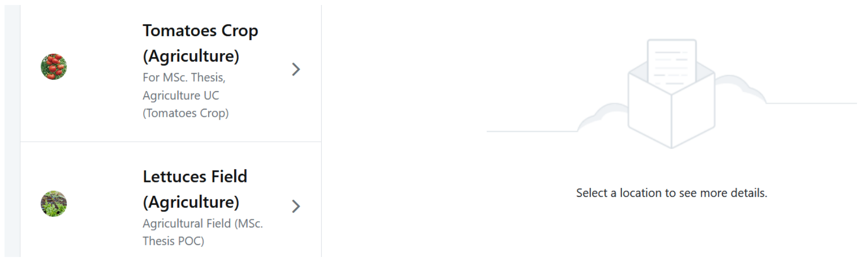
Figure 7.2: Groups list screen to manage groups and its roles.

7.3 Locations

Regarding locations, Figure 7.3 depicts the list screen, where a list containing the several locations associated with a user can be seen. If the user selects a location, on the left, a summary of that location is shown on the right. An example of this behaviour is shown for the "Tomatoes Crop Test". The reader can see that the user "Daniel" is also assigned to this location. Also, by clicking on the "Edit" button, the user can edit a selected location. The list of locations, as in Figure 7.3a, Figure 7.3b and Figure 7.3c, contains the title, a photo (if exists) and a short description. If the user clicks on a location, more information is shown, such as the sensors, actuators and users assigned to the location.

Either by clicking on the "Add Location" or the "Edit" button, the user will be lead to the screen shown in Figure 7.3d: the location detail screen. This screen receives as input the Location identifier which will be `NullIdentifier()` in the case of the first button. In this screen, it is possible to add, both devices and users to the location. In this case, the device "Heater" was added and it can be removed by clicking on the "x" icon. Furthermore, as no device is currently selected on the `Dropdown`, the icon to add novel devices is disabled. In contrast, this icon is enabled below, to add an user, "Daniel". You can see that the user "Andreia Martins" is already assigned to this location. Finally, for the color associated with a location for personalization and UX purposes, a component for the color picker, seen in Figure 7.3e, was included[1]. It is possible to see this information saved and shown on the list in Figure 7.4a, where an icon with this color is shown to ease the identification of devices by its location.

[1]https://www.outsystems.com/forge/component-overview/9116/hi-input-color

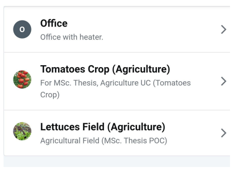


Tomatoes Crop (Agriculture)
For MSc. Thesis, Agriculture UC (Tomatoes Crop)

Lettuces Field (Agriculture)
Agricultural Field (MSc. Thesis POC)

Select a location to see more details.

(a)

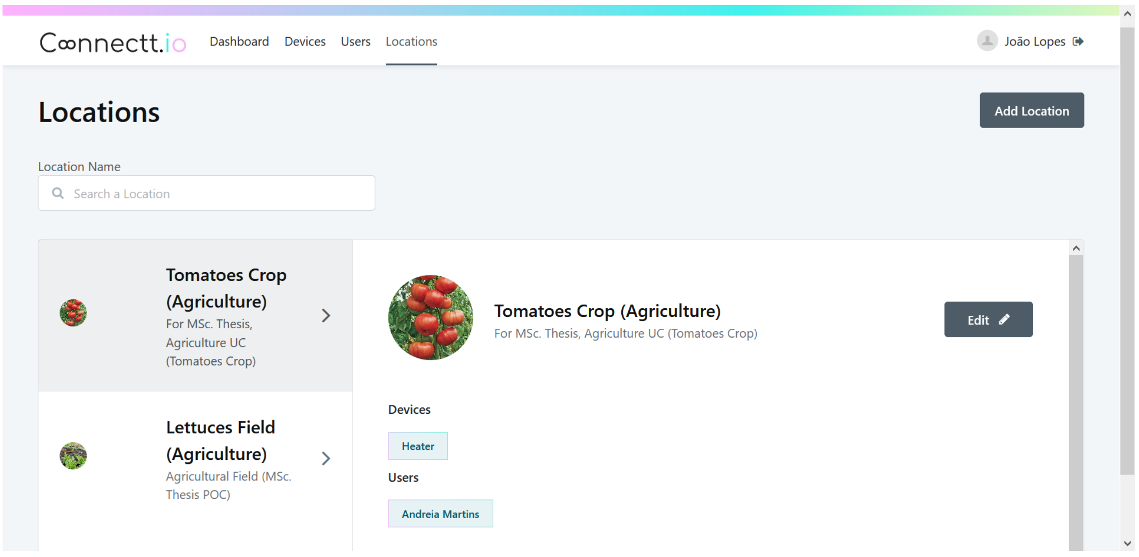


Office
Office with heater.

Tomatoes Crop (Agriculture)
For MSc. Thesis, Agriculture UC (Tomatoes Crop)

Lettuces Field (Agriculture)
Agricultural Field (MSc. Thesis POC)

(b)



Locations

Add Location

Location Name

Search a Location

Tomatoes Crop (Agriculture)
For MSc. Thesis, Agriculture UC (Tomatoes Crop)

Lettuces Field (Agriculture)
Agricultural Field (MSc. Thesis POC)

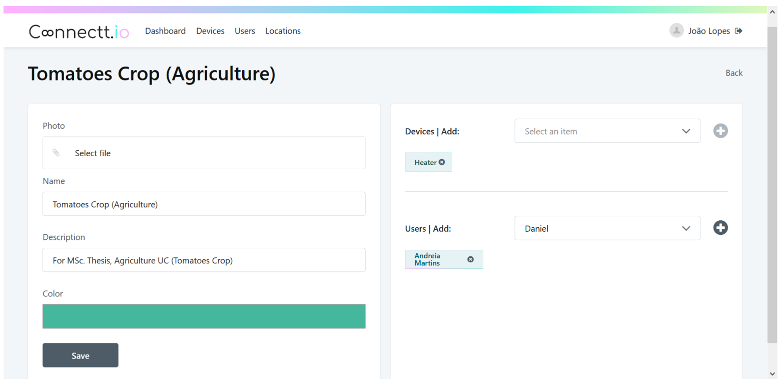
Tomatoes Crop (Agriculture)
For MSc. Thesis, Agriculture UC (Tomatoes Crop)

Edit

Devices
Heater

Users
Andreia Martins

(c)



Tomatoes Crop (Agriculture)

Photo: Select file

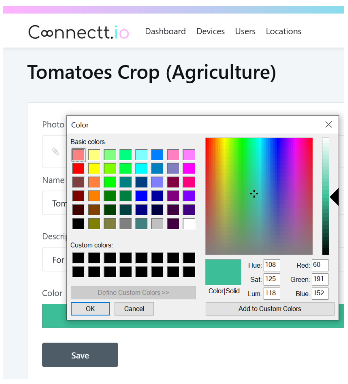
Name: Tomatoes Crop (Agriculture)

Description: For MSc. Thesis, Agriculture UC (Tomatoes Crop)

Color

Devices | Add: Select an item
Heater

Users | Add: Daniel
Andreia Martins



Tomatoes Crop (Agriculture)

Hue: 108 Sat: 125 Lum: 118
Red: 60 Green: 191 Blue: 152

(d) (e)

Figure 7.3: Locations' list with (a) no location selected, (b) in mobile, (c) showing a location detail upon selection on the list and the locations detail screen with (d), the screen and (e), to add a color to the location.

7.4 Devices and Transducers

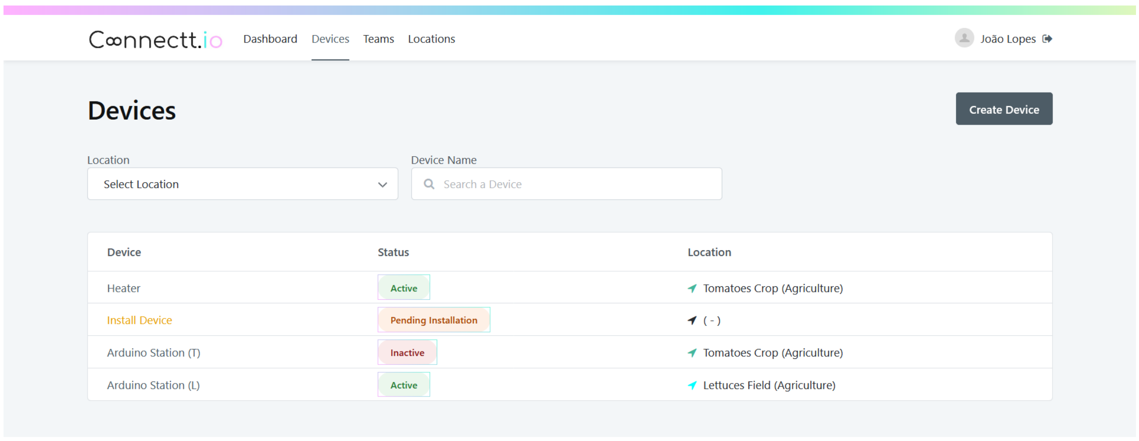
Upon receiving data on the MQTT client, the client will try to register those measurements. In simple terms, there are three possible outcomes: the first, where the device and transducer exist, where the measurements are registered on the database, associated with the transducer; the second, where the device exists and the transducer is not found, for the provided MQTT topic and payload and the latter, where neither the device, nor the transducer are found for a given topic and `JSONIdentifier` on the payload. For these two cases, the measurements are stored in a temporary table and only upon the transducer registration, these measurements are registered, *per-se*, which means that only then, they could be seen in the platform. Also, if the device associated with a topic and message does not exist, the system will automatically create those.

These 3 cases where tested and so, 2 devices were created (one of them with missing transducers). Also a third device not registered on the platform was used for tests, but its messages started being received. The behaviour, as expected, is seen in Figure 7.4a, where a device received data, but given that it was not on the system, it was created with the "Pending Installation" status. By opening it, one can see its details, complete its details, activate the device and add its transducers. After doing this, it was seen that the measurements on the temporary table were transfered and are now registered. The same happened to the devices with missing transducers. Data corresponding to Inactive devices was discarded, as expected as well. It is also possible to see that, on the list in Figure 7.4a, the locations' icons have the same color as the one selected when creating the location. This eases the user to find devices more easily. The filters were also tested and are working as expected. It is possible to filter by location by selecting one on the `Dropdown` and to filter by name.

Finally, clicking on the "Create Device" button, as well as clicking on the device name, on the list, will redirect the user to the device's detail screen (Figure 7.4b), to create and edit devices, respectively, as well as add or remove transducers to it. In Figure 7.4b, it is possible to see a portion of this detail screen. In Figure 7.4c, the modal that allows to add a transducer to a device can be seen and was also tested and prooved to be functioning correctly.

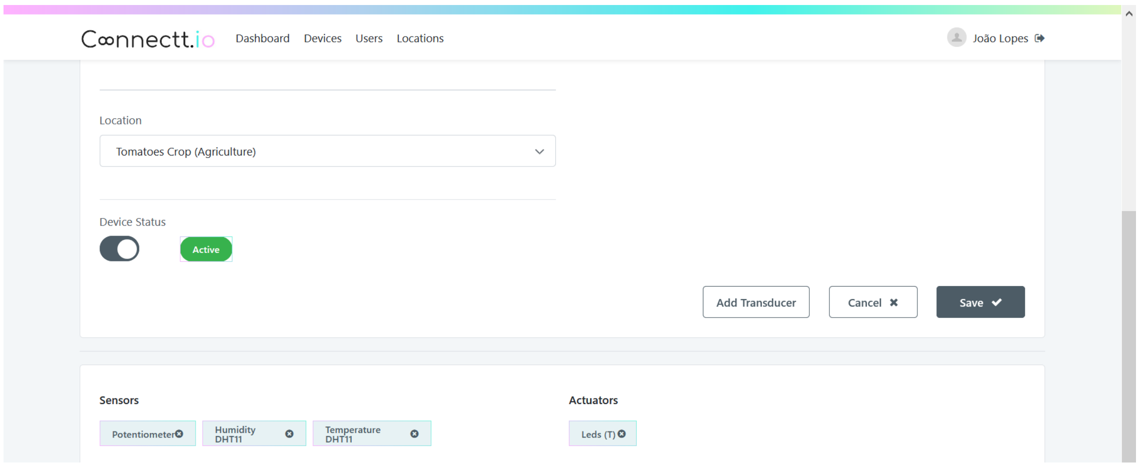
7.5 Dashboard

One of the main contributions of this thesis is the fact that the platform allows the user to create several dashboards. This is done by clicking on the plus sign on the bottom right corner of the screen, that will provide the option to add a novel dashboard or change a dashboard, in case no dashboard exists yet (Figure 7.5a). In case there is no dashboard selected/created by the user, the user can take a shortcut and click on the button in the center of the screen (Figure 7.5b).



Device	Status	Location
Heater	Active	Tomatoes Crop (Agriculture)
Install Device	Pending Installation	(-)
Arduino Station (T)	Inactive	Tomatoes Crop (Agriculture)
Arduino Station (L)	Active	Lettuces Field (Agriculture)

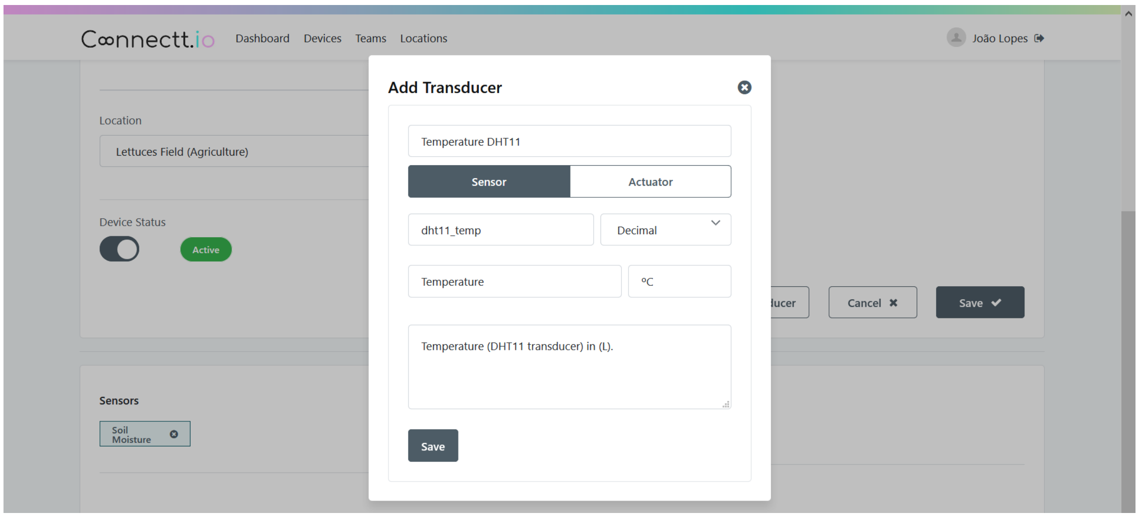
(a)



Location: Tomatoes Crop (Agriculture)

Device Status: Active

(b)



Location: Lettuces Field (Agriculture)

Device Status: Active

Temperature DHT11

dht11_temp

Decimal

Temperature

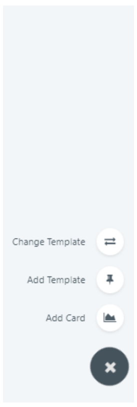
ºC

Temperature (DHT11 transducer) in (L).

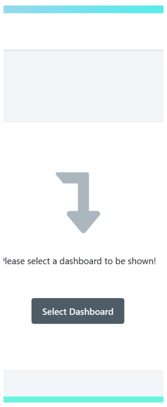
(c)

Figure 7.4: Devices screens with (a) the list of devices and (b), its details and (c), with the modal to add a transducer to a device.

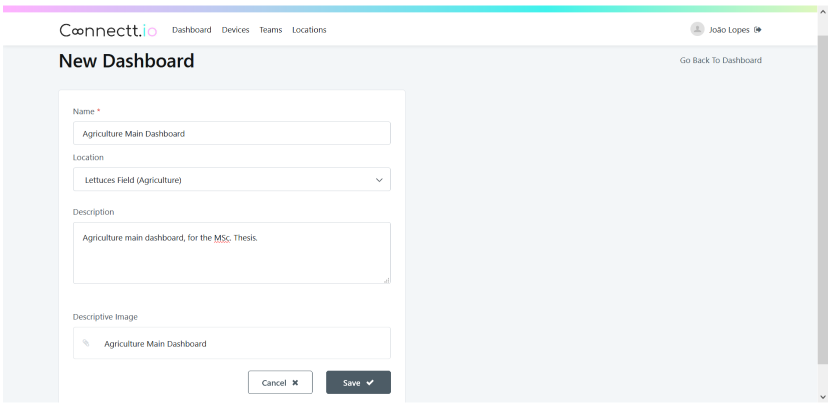
Both buttons, in case no dashboard exists yet, will open the screen visible in Figure 7.5c, which allows to create a dashboard by providing its name, a location, a description and, if available, a photography that may ease the identification of the dashboard. After doing this, one can choose dashboards by choosing the corresponding option on the plus sing in the bottom right corner, or by clicking on the "change" icon in the top right corner, alongside the title and location `Tag`. This will open the modal depicted in Figure 7.5d, which presents a gallery with the different dashboard options. By clicking on one, that dashboard becomes the predefined by that user. All of this was tested and is working as expected.



(a)



(b)



New Dashboard

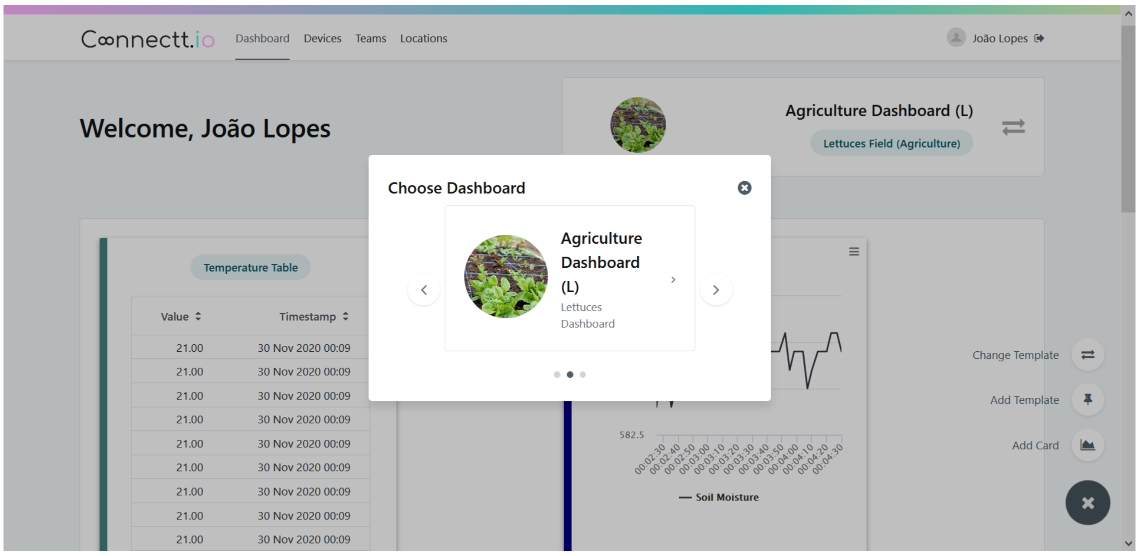
Name *
Agriculture Main Dashboard

Location
Lettuces Field (Agriculture)

Description
Agriculture main dashboard, for the MSc. Thesis.

Descriptive Image
Agriculture Main Dashboard

(c)



Welcome, João Lopes

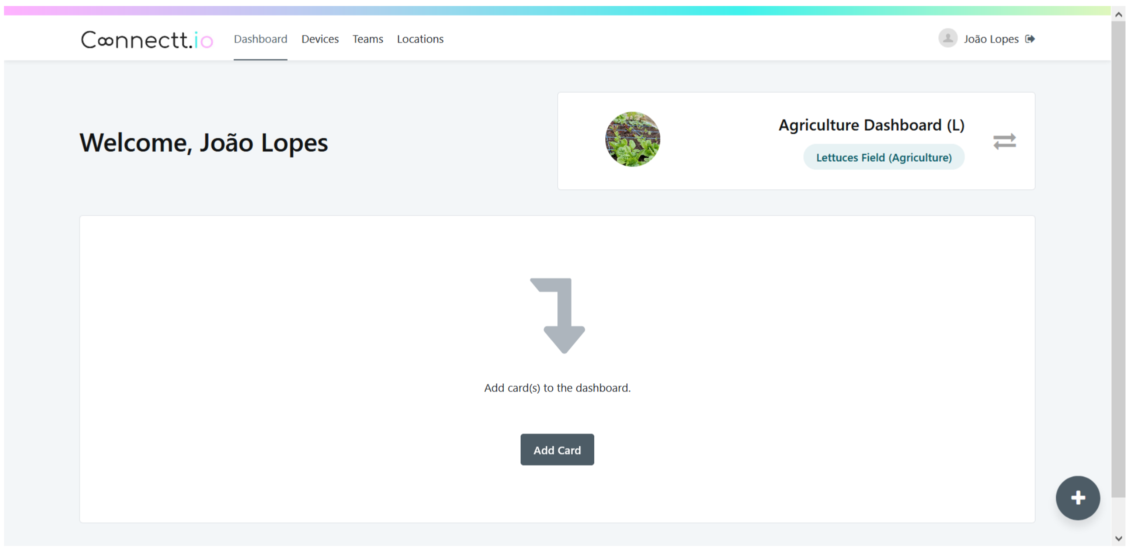
Temperature Table

Value	Timestamp
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09

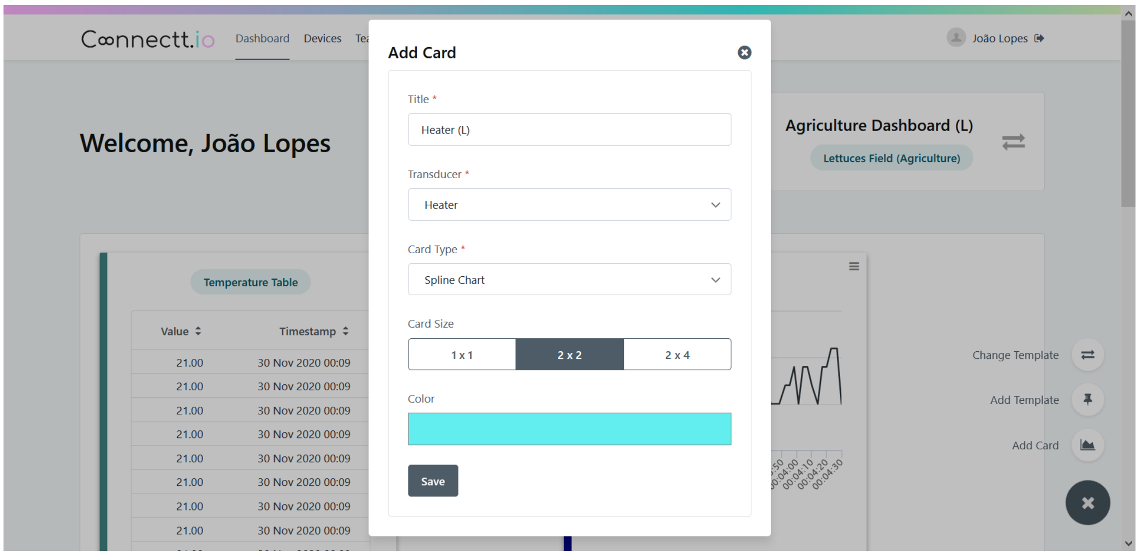
(d)

Figure 7.5: Dashboard initial screens with (a), no dashboard selected, (b), the buttons with dashboard options, (c), with the dashboard detail screen to edit and create dashboards and (d), the screen to change dashboards.

Upon selecting a dashboard, if there are no cards associated with that dashboard yet, the screen in Figure 7.6a will be shown and, either by clicking on the "Add Card" in the center or via the plus icon on the bottom, the user will be able to open a modal to add cards: Figure 7.6b. In this modal, it is possible to select the card title, add the card transducer and select the type of card, through a `Dropdown`. The size of the card is chosen via a `ButtonGroup` and the color uses the same component as in when creating locations. This color will be used on the border of the cards, to easily identify them.


Connectt.io
Dashboard
Devices
Teams
Locations
João Lopes
Welcome, João Lopes
Agriculture Dashboard (L)
Lettuces Field (Agriculture)
Add card(s) to the dashboard.
Add Card

(a)



Add Card

Title *
Heater (L)

Transducer *
Heater

Card Type *
Spline Chart

Card Size
1 x 1 | [✓ 2 x 2] | 2 x 4

Color

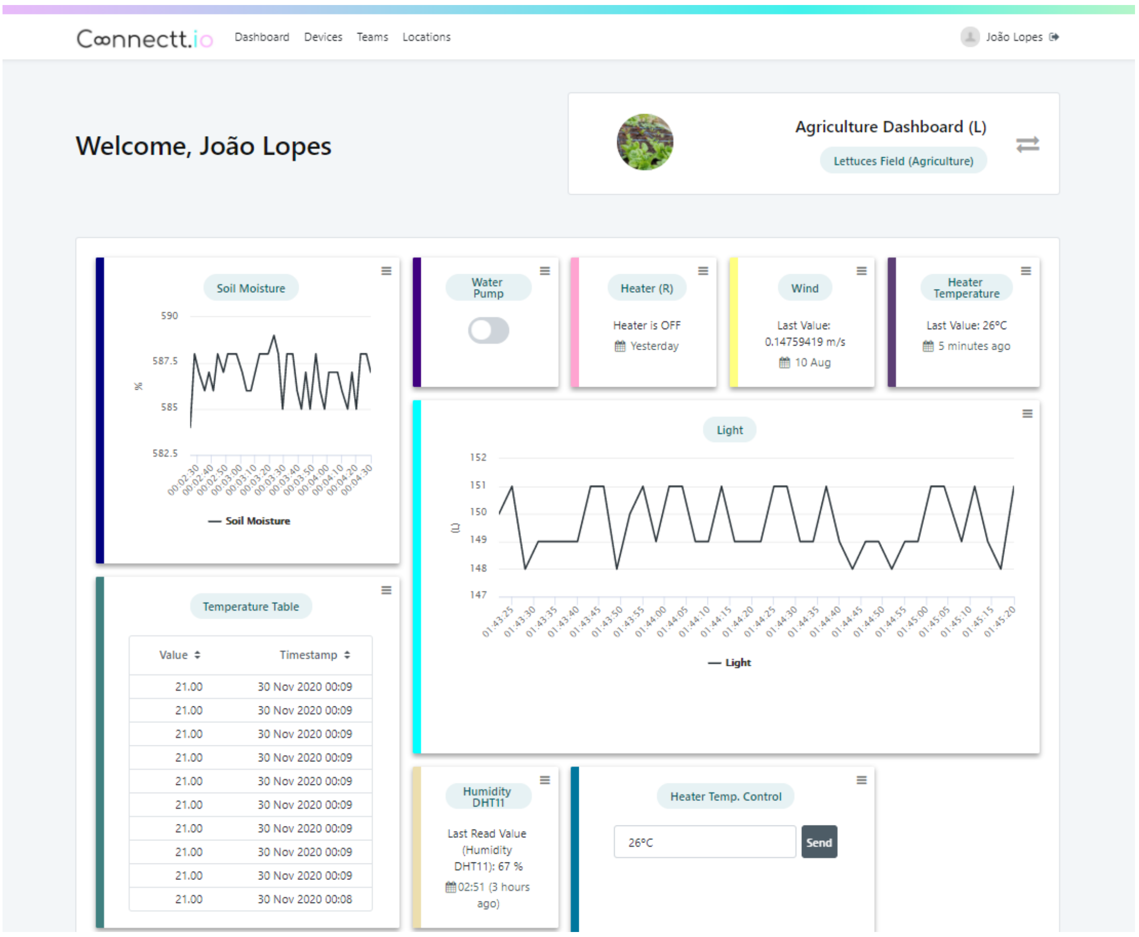
Save

Value	Timestamp
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09

(b)

Figure 7.6: Screens to add a card to dashboard in web application with (a), an empty dashboard, where the central button provides easy access to (b), the modal to add and edit a dashboard card.

Finally, a dashboard can be seen in the Figure 7.7, where several cards, with different types and sizes were added. First, an spline plot can be seen in both a 2×2 and 2×4 size (when hovering above the plot with the mouse, it is possible to see its values in that point: the border of the balloon also has the same color as the card); next, a switch, that allows to change the status of the water pump; next, 3 cards displaying the last recorded value of 3 different transducers; in addition, a table in a 2×2 card, containing some temperature values (from the dht_11 transducer); finally, a card to send values via MQTT. In this case, a temperature is being sent. By clicking on the title of the card, it is possible to edit the card. Also, it is possible to remove the card in the same place. Furthermore, it is possible to rearrange the cards order.



Welcome, João Lopes

Agriculture Dashboard (L)
Lettuces Field (Agriculture)

Heater (R)
Heater is OFF
Yesterday

Wind
Last Value: 0.14759419 m/s
10 Aug

Heater Temperature
Last Value: 26°C
5 minutes ago

Temperature Table

Value	Timestamp
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:08

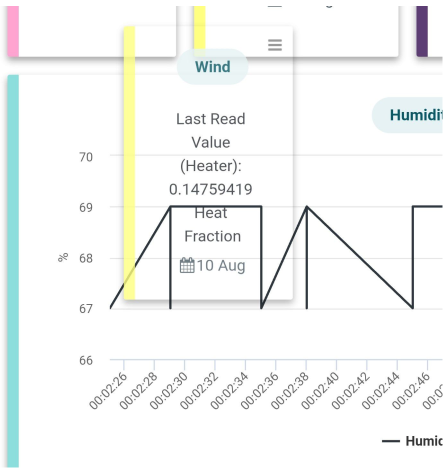
Humidity DHT11
Last Read Value (Humidity DHT11): 67 %
02:51 (3 hours ago)

Heater Temp. Control
26°C

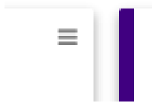
Figure 7.7: Dashboard with cards in the Lettuces Field location. The cards can be added, removed and rearranged.

7.6 Components

As said several times in this document, the SortableJS was an Outysystems component that can be used anywhere and in any application, web or mobile. This thesis allowed to test it, both in web and mobile, showcasing that, although it is based on the HTML DnD API, it supports touch, as seen in Figure 7.8a, where a card is being moved around in a mobile device, by using touch. Both in the web and mobile applications, it is only possible to move the cards when clicking on the icon in the top right corner of the cards, depicted in Figure 7.8b. The mouse cursor changes from pointer to a grabbing hand, indicating that it is now possible to grab and move the card. It was tested if the configuration upon rearranging the cards was saved by refreshing the page. This was indeed the case.



(a)



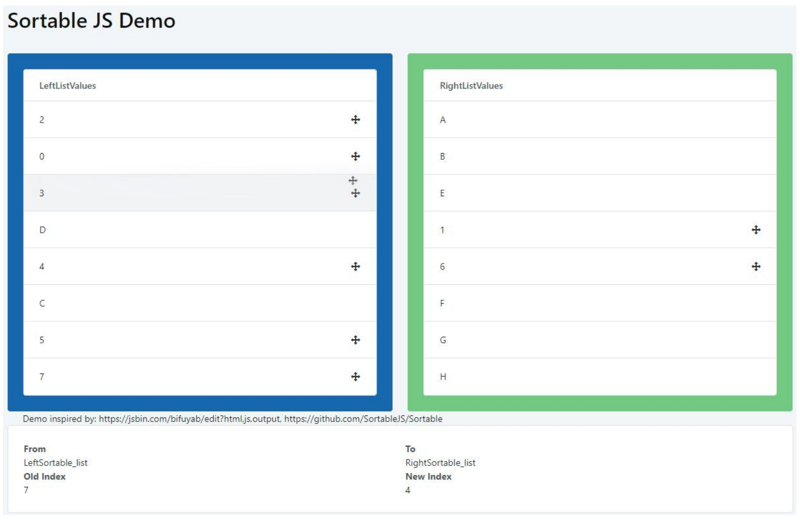
(b)

Figure 7.8: Drag and drop with the SortableJS component with (a), a card being dragged in a mobile/touch device and (b) with the icon containing the CSS class indicated in the SortableJS JSON options to allow the icon to be the grabbable area.

This component is already published[2], currently having a total of 91 downloads and 13 people following the component, meaning that they will be notified every time the component receives updates. I have also received messages from users reporting bugs and requesting some guidance on how to use the component, which confirms that the component is being used by the Outsystems community. Based on the community feedback, I have managed to correct some errors, namely where the JSON was not being parsed in the Javascript.

Also, the demonstration[3] can be tried on a browser. A screenshot of the developed demonstration is shown (Figure 7.9), where one can drag and drop elements from one list to another and inside the same list and also, see the output of the events being shown below.

[2]https://www.outsystems.com/forge/component-overview/8045/sortablejs
[3]https://joao-p-de-lopes.outsystemscloud.com/SortableJS_Demo/Demo?_ts=637296747936438250



Sortable JS Demo

LeftListValues
2
0
3
D
4
C
5
7

RightListValues
A
B
E
1
6
F
G
H

Demo inspired by: https://jsbin.com/bifuyab/edit?html,js,output, https://github.com/SortableJS/Sortable

From
LeftSortable_list
Old Index
7

To
RightSortable_list
New Index
4

Figure 7.9: DraggableJS Forge Component Demonstration.

It is important to keep in mind that this component is not specific to IoT, as it can be used in any aplication, from commercial large projects, to the smallest ones. The published demonstration is a great example, as it has no IoT associations. It is simply made of two generic lists. Furthermore, a key aspect and contribution of this component is that it was, at least at the time and as of my knowledge, the first React DnD component in the Outsystems Forge and it was simple to use. The user only needs to drop the placeholder on a screen/WB and its childs inside it. Moreover, this component is having an impact on the community, as it is already being used by other developers worldwide and change requests have been made by them. Finally, an article related to this component was written and published[4].

Also, as it was seen in Chapter 4, it was important for this thesis to showcase the mobile capabilities of a mobile device, compared to a web browser. In this sense, one type of component that can be built for mobile devices is a plugin, which would wrap up a Cordova plugin and make use of the mobile device's hardware, such as its location, gyroscope, bluetooth, camera, among others.

So it was thought of an IoT plugin need, yet not developed in Outsystems, such as barcodes and QR-codes, that can be used in IoT for process and control optimization in Business-to-Business (B2B) [57], as these codes can provide information about a product or its manufacturer [57]. For instance, they can be used for tracking reasons [57] - to control a delivery chain. As there are already some Forge plugins that can read these codes using the phone camera, it was decided to develop a plugin to interact with the scanners in Zebra[5] devices. These devices are widely used to track products and its embedded scanners are quicker than the traditional camera.

[4]`https://medium.com/noesis-low-code-solutions/drag-n-drop-in-outsystems-the-recipe-4ac8c6104f46`

[5]`https://www.zebra.com/us/en.html`

The component that was developed is already available for download in the Outsystems Forge[6]. It communicates with the device hardware via Android intents and makes use of the code presented in [58]. More details about the implementation of this plugin wrapper are shown in Appendix A. This component is also already being used in different developers, worldwide. The component developed in this thesis is also being used by one of the biggest retail chains in the Iberian Peninsula. The component has had 55 downloads and 7 people following it and I have also been contacted by people using this component. A demonstration with this component is also available in the component download page.

Just to reinforce, although these were developed specifically for IoT applications, these components are generic and may be used in any application.

7.7 Offline Synchronization

To validate the offline synchronization on the mobile application, one just had to simply open the application and verify that data was being displayed. This is confirmation since the data on the dashboard screen is being shown from the device's data storage and if this data is being shown, it means it was stored successfully.

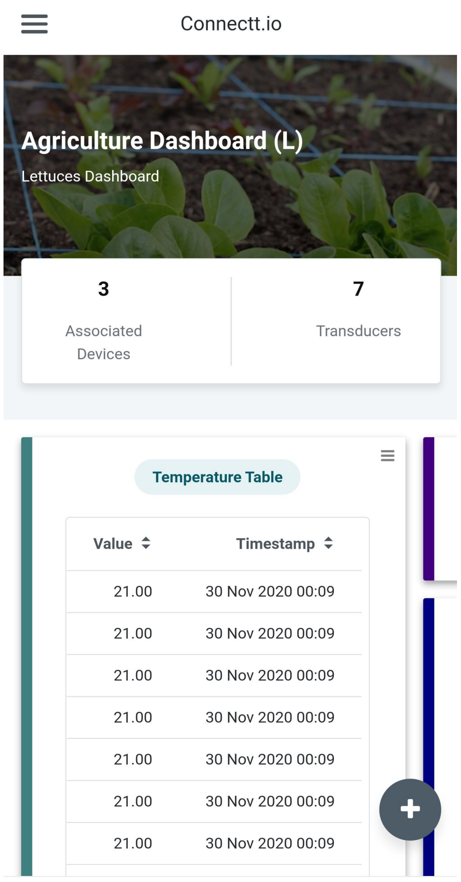
Nonetheless, debug was done to verify, step-by-step, if this data is being stored as intended, as well as every time de device becomes online, a login is done and on the application resume (which happens when the user opens the application from the application's drawer). The dashboard as seen in a mobile device is shown in Figure 7.10. It looks similar to the web application but this is a completely different application in a completely different module. Although it is reusing most of the logic used in the web application.

7.8 Discussion

At the end of this chapter, it is possible to say that Low-Code allowed to increase the development speed and, at the same time, to decrease the probability of errors and in the future, to ease the addition of new features and maintenance. Some of the contributions were: two components (drag and drop and barcode reader); MQTT integration; mechanisms to maintain the database clean; devices, locations, users and dashboards maintenance; the possibilities to customize the dashboard.

Both components are already being used by the community and the published article will only contribute to widen the usage. The simplicity of the drag and drop component, its wide versatility and the fact that it is touch-compatible make it a useful component in DnD applications.

[6]https://www.outsystems.com/forge/component-overview/6328/zebra-datawedge-connector--plugin



Value	Timestamp
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09
21.00	30 Nov 2020 00:09

Figure 7.10: Mobile application dashboard.

In addition, the IoT platform is stable and all of the predicted functionalities were developed with success. An estimated amount of two weeks was given to tune and bug fixing, which allowed to conclude the platform developments with success. Also, comparing the development times between the web application (built first) and the mobile application, it was possible to confirm that the modular architecture decreased development times when reusing the logic in the mobile application. Also, when a bug was found, most of the times, it was only required to replace it in one place.

Finally, and in my personal opinion as someone whom has been working with Outsystems for almost 2 years, the traditional web development approach takes considerably longer. Also, the fact that the mobile developments are so similar to the web developments when using React, the development effort is lower. The Outsytems platform optimizes the developments and so, not only are applications built faster, they are also built better. In case there are some limitations on the platform, it is still possible to combine Outsystems with Javascript, CSS and .Net extensions. Other than the component article[7], a scientific article is also under development.

[7]https://medium.com/noesis-low-code-solutions/drag-n-drop-in-outsystems-the-recipe-4ac8c6104f46

Chapter 8

Conclusions and Future Work

At the end of this thesis, it is possible to say that the main objectives were achieved. An IoT platform was built using a Low-Code platform, showcasing the versatility of these kind of platforms and its promising future.

One of the goals was to show that the modular architecture could, in addition to the use of a low-code development platform, accelerate the developments in great extent. Although it was not possible to measure developments at all times, the mobile application was built in a week, whereas the web application took approximately 8 weeks. Not only the reuse of code due to the architecture accelerated developments, but also, in my opinion, developing using low-code platforms is unequivocally faster comparing with more "traditional" approaches. The web developments took longer simply because the CS-modules' actions were developed simultaneously with the application *per-se*. In the future, the development times when using Outsystems and similar platforms might be even faster given that novel components that intend to help the developer solve problems are published every day. Also, Outsystems optimizes the code in the background, for instance, an Aggregate resulting SQL-query only fetches the data being used. Like this, there are plenty other improvements. The same happens with security, with Outsystems handling some common security threats automatically.

One of the main objectives of this thesis was to build a customizable dashboard and that objective was fulfilled, as well as a component was published and can now be downloaded and used by the Outsystems community. The ease of use of this component is one of the main advantages. A demonstration of this component is available online, where it is possible to drag and drop elements from one list to another, as well as the component was tested on the mobile application and it was confirmed that it can be used in touch devices as well.

Also, a barcode/qrcode reader was developed and is compatible with the Zebra devices. This component was tested, both in my personal device (with a common camera), as well as with a Zebra TC20, which contains an integrated reader, with a physical button on the device, which can be used with the developed plugin to trigger the "scanning". Moreover, the developed mobile application reproduces all of the web application's functionalities, as well as it adds local storage so that, for instance, a farmer, can use the application and visualize data on the dashboards without a connection to the internet, which is likely to happen in agriculture environments, as an example. Also, sometimes, heavy data had to be processed and this was handled elegantly, using heavy timers and light BPTs and BPTs. Mechanisms were also used to avoid errors, data loss and timeouts.

There is always space for improvements. First and foremost, the MQTT topics should include some authentication properties on the payload message JSON. Security is of most importance in IoT applications and, although Outsytems takes care of some of the threats, there is always more that can be done. Furthermore, more types of cards can be built, such as dials, gauge meters, radio buttons, custom HTML, as well as more types of plots can be added and more than one data series can be added to the plots. There should also be possible to range the measurements, both in plots and in tables. Also, a feature that should be added is the possibility to schedule events and tasks, for instance, turn on the water pump when the humidity is below a certain threshold, or send a notification/email whenever this threshold is hit. Continuing, a component named Data Grid[1] could be used to visualize the transducers' data in a different way. More components could also be developed or used in order to implement artificial intelligence and machine learning on the platform, as these concepts are often related with IoT. Additionally, an MQTT React component should be developed in Outsystems to replace the used one (in Traditional Web). This would allow asynchronous calls and, generally, a better handling of parallel incoming MQTT data. This component would also make possible to show data changing in real time on the developed dashboard. Again, this component could be used by the community in the future. In terms of database, the concept of company could be included, where each company has it's own users and teams. More protocols should also be added in the future, as well as more widgets and more personalization.

Nowadays, Outsystems is relatively expensive and the trial version was used to develop this work. Nevertheless, as Hendriks states, "This improvement in efficiency enables the idea of continuous integration by being able to produce a business application in matter of weeks instead of months." [29], regarding model-based platforms and so, one can only hope that eventually, low-code becomes more accessible and affordable, so that progress in several fields, such as IoT could be achieved faster than before.

A scientific article is currently under development and another one was already published[2].

[1]https://www.outsystems.com/forge/component-overview/5554/data-grid-web

[2]https://medium.com/noesis-low-code-solutions/drag-n-drop-in-outsystems-the-recipe-4ac8c6104f46

Bibliography

[1] *Technology is moving faster than ever before — here's how businesses are keeping up.* https://www.businessinsider.com/sc/how-businesses-keep-up-with-technology?sr_source=lift_facebook. Accessed: 31-10-2020.

[2] D. Michels. *Measuring Your Organization's Ability To Change, July, 2020.* https://www.forbes.com/sites/davidmichels/2020/07/27/measuring-your-organizations-ability-to-change/?sh=554c26f85466. Accessed: 31-10-2020.

[3] S. Madakam, R Ramaswamy, and S. Tripathi. "Internet of Things (IoT): A literature review." In: *Journal of Computer and Communications* 3.05 (2015), p. 164.

[4] *The future of IoT: 10 predictions about the Internet of Things.* https://us.norton.com/internetsecurity-iot-5-predictions-for-the-future-of-iot.html. Accessed: 08-09-2020.

[5] I. Yaniv Iarovici Marketing Director of Industrial and W. D. Edge. *Top 10 IoT Use Cases | July, 2020.* https://www.iotevolutionworld.com/iot/articles/446032-top-10-iot-use-cases.htm. Accessed: 08-09-2020.

[6] *Low-Code Is the Future - OutSystems Named a Leader in the 2019 Gartner Magic Quadrant for Enterprise Low-Code Application Platforms.* https://www.prnewswire.com/in/news-releases/low-code-is-the-future-outsystems-named-a-leader-in-the-2019-gartner-magic-quadrant-for-enterprise-low-code-application-platforms-868871260.html. Accessed: 26-09-2020.

[7] *Dashboards.* https://www.gartner.com/it-glossary/dashboard. Accessed: 22-07-2019.

[8] *IoT Dashboards – Attributes, Advantages, & Examples.* https://ubidots.com/blog/iot-dashboards/. Accessed: 25-07-2019.

[9] *ThingsBoard Open-source IoT Platform.* https://thingsboard.io. Accessed: 26-07-2019.

[10] E. N. Bitencourt and W. P. dos Anjos. *IoT Centralization and Management Applying ThingsBoard Platform.* Häme University of Applied Sciences - HAMK, University Centre-Hämeenlinna, Smart Services Research Unit, 2018.

[11] *IoT Standards and Protocols.* https://www.postscapes.com/internet-of-things-protocols/. Accessed: 17-08-2019.

[12] *Examining 5 IEEE Protocols – ZigBee, WiFi, Bluetooth, BLE, and WiMax.* https://www.iotforall.com/ieee-protocols-zigbee-wifi-bluetooth-ble-wimax/. Accessed: 10-08-2019.

[13] *How Does WiFi Work? – 8 Things You Didn't Know About Wifi.* https://www.iotforall.com/how-does-wifi-work/. Accessed: 10-08-2019.

[14] *How Mesh Networking Will Make IoT Real.* https://hackernoon.com/how-mesh-networking-will-make-iot-real-b5b88baab63b. Accessed: 15-08-2019.

[15] *How Do Cellular Networks Work for IoT?* https://www.iotforall.com/cellular-connectivity-iot-overview/. Accessed: 11-09-2019.

[16] *What Is LoRa? A Technical Breakdown.* https://www.link-labs.com/blog/what-is-lora. Accessed: 05-09-2019.

[17] *Sigfox: A cellular network, just for Things.* https://www.nickdecooman.com/blog/sigfox-a-cellular-network-just-for-things. Accessed: 11-09-2019.

[18] *SigFox for M2M & IoT.* https://www.electronics-notes.com/articles/connectivity/sigfox/what-is-sigfox-basics-m2m-iot.php. Accessed: 11-09-2019.

[19] *MQTT and HTTP : Comparison between two IoT Protocols.* https://iotdunia.com/mqtt-and-http/. Accessed: 12-09-2019.

[20] *Internet of Things: Battle of The Protocols (HTTP vs. Websockets vs. MQTT).* https://www.linkedin.com/pulse/internet-things-http-vs-websockets-mqtt-ronak-singh-cspo/. Accessed: 12-09-2019.

[21] *MQTT and DDS for M2M: Disparate Approaches to the Internet of Things.* https://www.rti.com/blog/mqtt-dds-m2m-protocol-internet-of-things/. Accessed: 26-08-2019.

[22] *Beginners Guide To The MQTT Protocol.* http://www.steves-internet-guide.com/mqtt/. Accessed: 12-09-2019.

[23] J. Vermillard. *Introduction to CoAP, the REST protocol for M2M.* 2013.

[24] *Wireless Sensor Network Node with REST Advantages: CoAP Protocol.* http://www.wsnmagazine.com/wsn-coap-protocol/. Accessed: 31-08-2019.

[25] *CoAP Protocol: Step-by-Step Guide.* https://dzone.com/articles/coap-protocol-step-by-step-guide. Accessed: 12-09-2019.

[26] *JSON vs XML.* https://www.educba.com/json-vs-xml/. Accessed: 14-09-2019.

[27] *Low-Code Development Platforms.* https://www.outsystems.com/low-code-platforms/?utm_source=twitter&utm_medium=agorapulse&utm_campaign=lowcodeplatforms&utm_content=lowcodelp-meta&utm_term=internal-out.

[28] Outsystems. *What Is Low-Code?* https://www.outsystems.com/blog/what-is-low-code.html. Accessed: 20-05-2019.

[29] D. Hendriks. *The selection process of model based platforms*. Tech. rep. Radboud Universiteit, 2017.

[30] *Application Platform as a Service: What Is It Really?* https://www.outsystems.com/blog/posts/application-platform-as-a-service/. Accessed: 14-09-2019.

[31] F. J. G. Pereira. "The DS-Pnet modeling formalism for cyber-physical system development." In: (2017).

[32] Outsystems. *Outsystems Overview - Introduction to the Course*.

[33] R. Alves. *Reactive Web Applications - The Next Generation of Web Apps*. https://www.outsystems.com/blog/posts/reactive-web-applications/. Accessed: 27-09-2020.

[34] T. Simões. *All You Need to Know About Reactive Web Applications*. https://www.outsystems.com/blog/posts/all-you-need-to-know-about-reactive-web/. Accessed: 27-09-2020.

[35] Outsystems Documentation. *Actions in Web Applications*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Implement_Application_Logic/Actions_in_Web_Applications. Accessed: 28-07-2019.

[36] Outsystems Documentation. *Processes*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Use_Processes_(BPT)/Processes. Accessed: 10-09-2019.

[37] Outsystems Documentation. *Extensions*. https://success.outsystems.com/Documentation/11/Extensibi.

[38] Outsystems Documentation. *From architecture to development*. https://success.outsystems.com/Documentation/Best_Practices/Architecture/From_architecture_to_development. Accessed: 21-03-2021.

[39] *User Stories with Examples and Template*. https://www.atlassian.com/agile/project-management/user-stories. Accessed: 02-06-2020.

[40] D. Arnott. *Using Personas for Writing User Stories*. https://www.frontrowagile.com/blog/posts/29-using-personas-for-writing-user-stories. Accessed: 05-10-2020.

[41] Outsystems Documentation. *The 4 Layer Canvas*. https://success.outsystems.com/Support/Enterprise_Customers/Maintenance_and_Operations/Designing_the_architecture_of_your_OutSystems_applications/01_The_4_Layer_Canvas. Accessed: 04-06-2020.

[42] Lara Silva, OSQUAY, May 19. *How to make sure a heavy timer doesn't misbehave in the Outsystems platform*. https://www.osquay.com/knowledge-center/how-to-make-sure-a-heavy-timer-doesnt-misbehave-in-the-outsystems-platform. Accessed: 15-11-2020.

[43] Outsystems Documentation. *Read-Only Data Optimized | December, 2020*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Use_Data/Offline/Offline_Data_Sync_Patterns/Read-Only_Data_Optimized. Accessed: 20-02-2021.

[44] *HTML5 - Drag drop*. https://www.tutorialspoint.com/html5/html5_drag_drop.htm. Accessed: 18-01-2019.

[45] *Sorting with the help of HTML5 Drag'n'Drop API*. https://github.com/SortableJS/Sortable/wiki/Sorting-with-the-help-of-HTML5-Drag'n'Drop-API. Accessed: 21-04-2020.

[46] *Sortable v1.0 — New capabilities*. https://github.com/SortableJS/Sortable/wiki/Sortable-v1.0-%E2%80%94-New-capabilities. Accessed: 21-03-2020.

[47] Outsystems Documentation. *The Complete Guide to Creating Components*. https://success.outsystems.com/Documentation/Best_Practices/The_Complete_Guide_to_Creating_Components. Accessed: 22-03-2020.

[48] *SortableJS - JavaScript library for reorderable drag-and-drop lists*. http://sortablejs.github.io/Sortable/. Accessed: 22-03-2020.

[49] *Swap Thresholds and Direction*. https://github.com/SortableJS/Sortable/wiki/Swap-Thresholds-and-Direction. Accessed: 22-03-2020.

[50] *Dragging Multiple Items in Sortable*. https://github.com/SortableJS/Sortable/wiki/Dragging-Multiple-Items-in-Sortable. Accessed: 22-03-2020.

[51] *Get Started With ESP8266 Using "AT Commands"Via Arduino*. https://www.instructables.com/id/Get-Started-With-ESP8266-Using-AT-Commands-Via-Ard/. Accessed: 31-12-2019.

[52] *Arduino Soil Moisture Sensor*. https://www.instructables.com/id/Arduino-Soil-Moisture-Sensor/. Accessed: 21-07-2020.

[53] W. W. F. for Nature: Viera Ukropcova and S. Halevy. "Living Planet Report 2017 - Appetite for Destruction." In: (October, 2017).

[54] W. W. F. for Nature. "Living Planet Report 2016." In: (2016).

[55] R. Plumb. "Precision agriculture in the 21st century: geospatial and information technologies in crop management, Committee on Assessing Crop Yield: Site-Specific Farming, Information Systems and Research Opportunities, Board on Agriculture, National Research Council, National Academy Press, Washington DC, USA 1997, xii+ 149 pp, price£ 32.95. ISBN 0-309-05893-7." In: *Pest Management Science* 56.8 (2000), pp. 723–723.

[56] *Precision Crop Management - Putting plants on a tailor-made diet*. http://www.cema-agri.org/page/3-precision-crop-management. Accessed: 13-08-2018.

[57] *QR-Codes for Process and Control Optimization, November, 2019*. https://blog.quanos-service-solutions.com/en/expert-blog/qr-codes-for-process-and-control-optimization. Accessed: 01-11-2020.

[58] *Github repository from Darryn Campbell - DataWedge-Cordova-Sample - Under MIT License*. https://github.com/darryncampbell/DataWedge-Cordova-Sample. Accessed: 02-11-2020.

[59] Outsystems Documentation. *Using Cordova Plugins | October, 2020*. https://success.outsystems.com/Documentation/11/Extensibility_and_Integration/Mobile_Plugins/Using_Cordova_Plugins. Accessed: 01-11-2020.

[60] Outsystems Documentation. *Heavy Timers*. https://www.outsystems.com/learn/lesson/1749/heavy-timers. Accessed: 07-05-2020.

[61] Outsystems Documentation. *Design Scalable Database Queueing Using Light Processes*. https://success.outsystems.com/Documentation/10/Developing_an_Application/Use_Processes_(BPT)/Design_Scalable_Database_Queueing_Using_Light_Processes. Accessed: 07-05-2020.

[62] *Light BPT & Parallel Processing - Lightweight Multitasking - nextstep*. https://pt.slideshare.net/OutSystems/what-is-light-bpt-and-how-can-you-use-it-for-parallel-processing. Accessed: 07-05-2020.

[63] Outsystems Documentation. *Light Processes Under The Hood*. https://www.outsystems.com/forums/discussion/41063/light-processes-under-the-hood/. Accessed: 07-05-2020.

Appendix A

Zebra DataWedge Connector Plugin

As mentioned in Chapter 5, to showcase the mobile capabilities in Oustystems, a plugin component wrapping the cordova plugin available in [58] was developed. The first thing to do was to read the github repository thoroughly and ask the owner for permission to use the code, as there was no available license at the time.

Outsystems-wise, a module was created. Then, the Common Plugin was also downloaded and used, in order to follow best practices. The following step was to reference the cordova plugin by including the JSON identifier in the component's Extensibility Configuration, as seen in Listing A.1.

Listing A.1: Referencing the Cordova plugin.

```
{
    "plugin":
    {
        "identifier": "com-darryncampbell-cordova-plugin-intent@1.3.0"
    }
}
```

Before diving into the developments, it is important to mention that the cordova plugin available in the repository in [58] interfaces with the scanner hardware via android intents, communicating with an application named `DataWedge`, which is available in all Zebra devices. With this plugin, it will be possible to interact with the scanner, both via the physical buttons on the mobile device, or via buttons and links in an application.

According to the Outsystems best practices, its important to have an action to verify if the plugin is available in the and return an `IsAvailable Boolean` with the result [59]. Listing A.2 verifies if the plugin is available. Moreover, since this component is only to be used in Zebra devices, it is important to verify if that is indeed the case, by checking if the application that will communicate with the component via intents is also available.

This is done through another plugin, `AppAvailabilityPlugin`, which will check for the scheme `"com.symbol.datawedge"`. Notice that for this component, it is also a good practice to verify if it exists. All of this is shown in Figure A.1a.

The remaining actions will all make use of this first action, as seen in Figure A.1b, since one does not want perform actions if the plugin is not available. When using the plugin, it is required to register the broadcast receiver, allowing the DataWedge to communicate with the application where the component is being used, by providing its schema via input. Other configurations also took place, such as the profile name previously defined in DataWedge, as well as a boolean that indicates if a sound is to be played upon scanning a code.

A WB was created, containing 2 hidden text inputs and 2 hidden links. One of the inputs was used to store the scanned barcode/QR-code, whereas the second stored the number of available scanners detected in DataWedge, since external scanners can also be connected. One of the hidden links will call an action, `OnScanReceived`, which will retrieve the scanned barcode from the aforementioned input and send it through an Outsystems event. The same is done with the other link, returning a boolean indicating if the scanner is available (at least 1 detected scanner). Note that the identifiers of both the links and the inputs are passed to the `StartBroadcastReceiver` Javascript node, as the inputs will be populated from rows 109 to 124 in Listing A.3, as well as the links will be clicked inhere. The `StartBroadcastReceiver` snippet is called upon the rendering of the WB, on the `OnReady` action.

Furthermore, 4 more actions were included in the module. These are: `DisableScanner` to avoid the scanner from being used accidentaly; `ToggleSoftTrigger` which will toggle the scanner when this action is called; `TriggerPress`, which will keep scanning for as much time as the user is pressing a button and `UnregisterBroadcastReceiver`, which should be called to stop listening from intents, for example, on the `OnDestroy` event. These actions will not be shown as they were also adapted from [58] and are simpler.

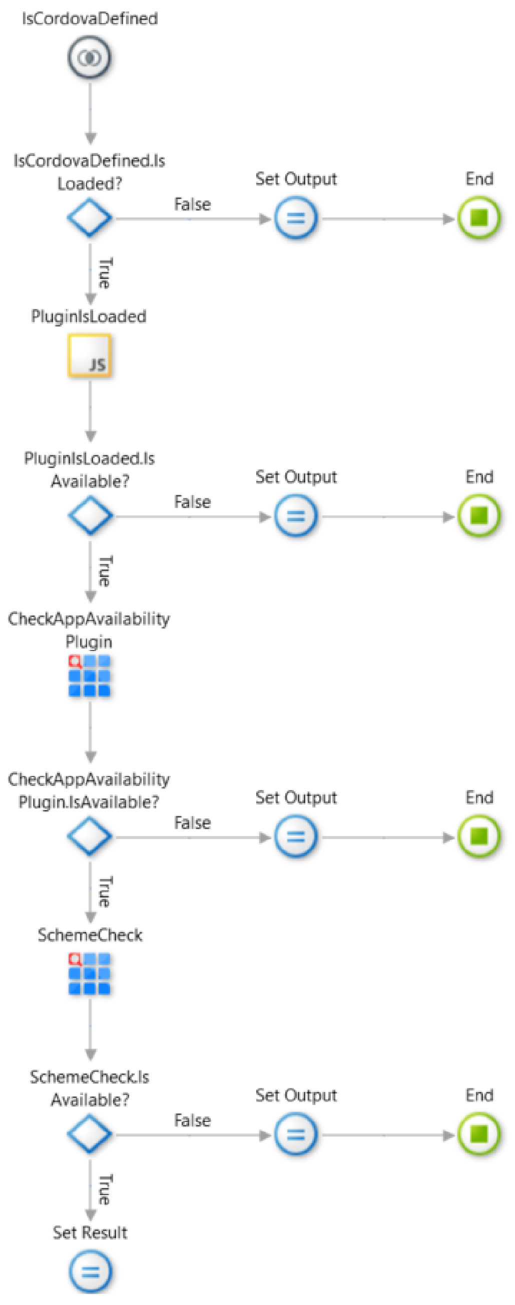
Listing A.2: Assessing if cordova plugin is available in the target module.

```
$parameters.IsAvailable = false;

if (typeof(window.plugins) !== 'undefined') {
   $parameters.IsAvailable = typeof(window.plugins.intentShim) !== 'undefined';
}
```

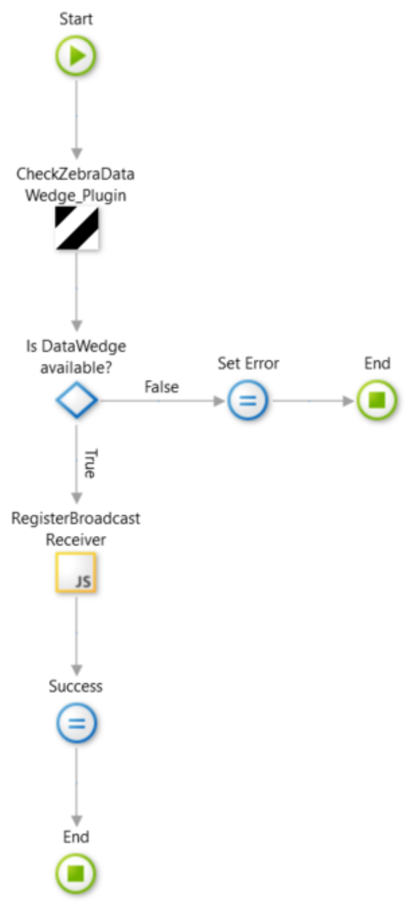
.,

Listing A.3: Javascript snippet to register a broadcast receiver

```
function sendCommand(extraName, extraValue) {
    var sendCommandResults = "false";
    //console.log("Sending Command: " + extraName + ", " + JSON.stringify(extraValue)
        ↪ );
    var broadcastExtras = {};
    broadcastExtras[extraName] = extraValue;
    broadcastExtras["SEND_RESULT"] = sendCommandResults;

    window.plugins.intentShim.sendBroadcast({
        action: "com.symbol.datawedge.api.ACTION",
        extras: broadcastExtras
    };
}

function datawedge63() {
    /* (...) */
}

function datawedge64() {

    /* (...) */
    var profileConfig2 = {
        "PROFILE_NAME": $parameters.ProfileName,
        /* (...) */
    };
    sendCommand("com.symbol.datawedge.api.SET_CONFIG", profileConfig2);

    var profileConfig3 = {
        "PROFILE_NAME": $parameters.ProfileName,
        /* (...) */
        "decode_audio_feedback_uri": $parameters.BeepSound

    };
    sendCommand("com.symbol.datawedge.api.SET_CONFIG", profileConfig3);

    setTimeout(function () {
        sendCommand("com.symbol.datawedge.api.GET_ACTIVE_PROFILE", "");
    }, 1000);
}

function registerBroadcastReceiver() {
    window.plugins.intentShim.registerBroadcastReceiver({
        filterActions: [
            /* (...) */
        ]
    },
    function (intent) {
        /* (...) */

        if (intent.extras.hasOwnProperty('
            ↪ com.symbol.datawedge.api.RESULT_GET_VERSION_INFO'))
        {
            /* (...) */

        }
        else if (intent.extras.hasOwnProperty('
            ↪ com.symbol.datawedge.api.RESULT_ENUMERATE_SCANNERS'))
        {
            // Return from our request to enumerate the available scanners
            var enumeratedScannersObj = /* (...) */
            var numberActiveScanners = enumeratedScannersObj.length;

            document.getElementById($parameters.IsScannerActiveId).value =
                ↪ numberActiveScanners;
            document.getElementById($parameters.HiddenLinkHasScannersId).click();

        }
        else if (!intent.extras.hasOwnProperty('RESULT_INFO') && intent.extras["
            ↪ com.symbol.datawedge.data_string"] != null)
        {
            document.getElementById($parameters.WidgetId).value = intent.extras["
                ↪ com.symbol.datawedge.data_string"];
            document.getElementById($parameters.HiddenLinkUponScanId).click();
            //console.log("Scan: " + intent.extras["com.symbol.datawedge.data_string
                ↪ "]);
        }
    }
    );
}

registerBroadcastReceiver();
// To determine version
sendCommand("com.symbol.datawedge.api.GET_VERSION_INFO", "");

/*
Adapted from 2019 Darryn Campbell https://github.com/darryncampbell/
    ↪ DataWedge-Cordova-Sample/blob/master/LICENSE
MIT license available in: https://github.com/darryncampbell/DataWedge-Cordova-Sample/
    ↪ blob/master/LICENSE
*/
```


IsCordovaDefined
IsCordovaDefined.Is Loaded?
False
Set Output
End
True
PluginIsLoaded
JS
PluginIsLoaded.Is Available?
False
Set Output
End
True
CheckAppAvailability Plugin
CheckAppAvailability Plugin.IsAvailable?
False
Set Output
End
True
SchemeCheck
SchemeCheck.Is Available?
False
Set Output
End
True
Set Result

(a)



(b)

Figure A.1: Zebra DataWedge Connector Plugin Outsystems actions with (a) to check for the cordova plugin availability in the target module and (b) to register the broadcast receiver.

Appendix B

Bootstraping Data

Due to the SARS-CoV-2 outbreak and resulting confinement, the laboratory material became inaccessible for quite some time. For this reason, the strategy of uploading data from an Excel file to the Outsystems platform took place. Those developments, although were not planned, allow to show the reader some more functionalities of the Outsystems' technology, namely, the use of timers, integrated with a Light Business Process Technology (LBPT).

Before diving into the timers and LBPTs, first, one needs to generate data. For that, a simple python script was written that would generate random sensor values, build the JSON that would be received from an Arduino and save that data on an Excel file. This script is depicted in Listing B.1.

Listing B.1: Python Script to generate devices and transducers' data to an Excel file.

```python
import xlsxwriter

from random import seed
from random import random

import datetime

def generate_transducer_value (transducer, is_sensor):
    random_value = random()
    if(is_sensor):
        return "'" + str(random_value) + "'"
    return "'" + str(random_value > 0.5) + "'"

def write_transceiver_JSON(transducer, value, timestamp):
    return "{'transducer':'"+transducer+"','value':"+str(value)+", 'timestamp': '#"+
        ↪ timestamp.strftime("%Y-%m-%d %H:%M:%S")+"#'},"

```

```
17  def bootstrap_device(device, timestamp, device_id, index):
18      JSON = "{'device':["
19      for transducer in device:
20          JSON+=write_transceiver_JSON(transducer, generate_transducer_value(transducer,
                ↪  device[transducer]), timestamp)
21      # print(JSON)
22      JSON = JSON[:len(JSON)-1] + "]}"
23      transducers_data.write('B'+str(index), JSON)
24      transducers_data.write('C'+str(index), device_id)
25
26  # seed random number generator
27  seed(1)
28
29  # key: transducer; value: is_sensor
30  deviceA = {'moisture': True, 'button1': False, 'rgb': False, 'leds1': False}
31  deviceB = {'dht11_temp': True, 'dht11_humidity': True, 'leds2': False, 'button2':
        ↪ False, 'potentiometer': True}
32  deviceC = {'piezo': False, 'leds3': False, 'ldr': True, 'button3': False, 'button4':
        ↪  False}
33
34  devices = {'A': deviceA, 'B': deviceB, 'C': deviceC}
35
36  workbook = xlsxwriter.Workbook('python/transducers_data.xlsx')
37  transducers_data = workbook.add_worksheet('Transducers Data')
38
39  transducers_data.write('A1', 'Id')
40  transducers_data.write('B1', 'JSON')
41  transducers_data.write('C1', 'Device')
42
43  current_date = datetime.datetime.now()
44
45  # generate device data
46  for i in range(1,1000, 3):
47      index = i
48      for device in devices:
49          index+=1
50          bootstrap_device(devices[device], current_date, device, index)
51      current_date = current_date + datetime.timedelta(minutes = 15)
52
53  workbook.close()
```

Dividing the script into smaller snippets, in rows 8 – 12, a function called `generate_` ↪ `transducer_value` will return a random decimal number in case the transducer is a sensor, or a boolean value, otherwise. This value is used as input to a function called `write_transceiver_JSON` to create a JSON object with the format in Listing B.2. The function receives the transducer name, its value and a timestamp (current date time incremented 15 minutes within a loop (row 52)). Next, a function named `bootstrap_device` joins all of the transducers (JSON objects into a JSON list, as shown in Listing B.3.

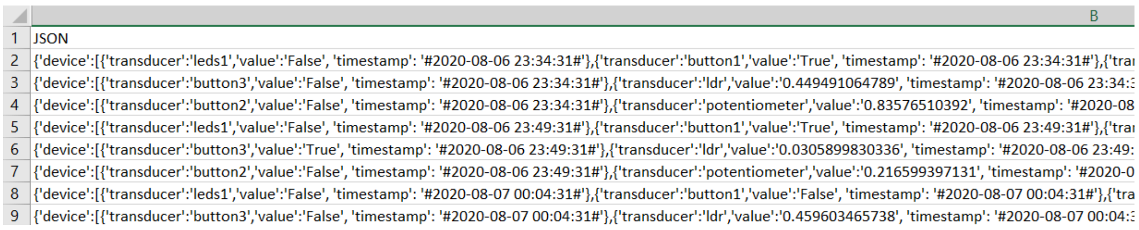
Listing B.2: Transducer JSON Object.

```
{
    transducer : "dht11_humidity",
    value : "54",
    timestamp : "#2020-08-01 13:01:55#"
},
```

Listing B.3: Device JSON Object

```
{
device: [
    {
        transducer : "leds1",
        value : "False",
        timestamp : "#2020-08-01 02:01:55#"
    },
    {
        transducer : "dht11_humidity",
        value : "54",
        timestamp : "#2020-08-01 13:01:55#"
    },
    (...)
]}
```

The devices were defined from row 30 to 32 of Listing B.1 as key value lists, with the key being the transducer identifier the value indicating if the device is a sensor. As this device is received as input in the function, it can then be iterated. For each transducer on the device, the `write_transducer_JSON` is called and its result appended to the JSON list. This is then written on an Excel file, as shown in Figure B.1.



	B
1	JSON
2	{'device':[{'transducer':'leds1','value':'False', 'timestamp': '#2020-08-06 23:34:31#'},{'transducer':'button1','value':'True', 'timestamp': '#2020-08-06 23:34:31#'},{'tra
3	{'device':[{'transducer':'button3','value':'False', 'timestamp': '#2020-08-06 23:34:31#'},{'transducer':'ldr','value':'0.449491064789', 'timestamp': '#2020-08-06 23:34:
4	{'device':[{'transducer':'button2','value':'False', 'timestamp': '#2020-08-06 23:34:31#'},{'transducer':'potentiometer','value':'0.83576510392', 'timestamp': '#2020-08
5	{'device':[{'transducer':'leds1','value':'False', 'timestamp': '#2020-08-06 23:49:31#'},{'transducer':'button1','value':'True', 'timestamp': '#2020-08-06 23:49:31#'},{'tra
6	{'device':[{'transducer':'button3','value':'True', 'timestamp': '#2020-08-06 23:49:31#'},{'transducer':'ldr','value':'0.0305899830336', 'timestamp': '#2020-08-06 23:49:
7	{'device':[{'transducer':'button2','value':'False', 'timestamp': '#2020-08-06 23:49:31#'},{'transducer':'potentiometer','value':'0.216599397131', 'timestamp': '#2020-0
8	{'device':[{'transducer':'leds1','value':'False', 'timestamp': '#2020-08-07 00:04:31#'},{'transducer':'button1','value':'False', 'timestamp': '#2020-08-07 00:04:31#'},{'tra
9	{'device':[{'transducer':'button3','value':'False', 'timestamp': '#2020-08-07 00:04:31#'},{'transducer':'ldr','value':'0.459603465738', 'timestamp': '#2020-08-07 00:04:

Figure B.1: Screenshot with bootstraped device data.

Once this is done, it is time to upload this data to the Outsystems platform. The first step is to add the `xlsx`-file to the IoTDevices_CS module resources. After this, one needs to have a way of storing this data on the Outsystems' platform. For that, a novel entity was created to store the temporary data retrieved from the resources' file. This entity simply has an `Id` and the JSON text field.

Next, a timer will be used to occasionally load the data from the `xlsx`-file to the temporary table. The reason why a temporary table is used is because it is not a good practice to process the data on a timer, as timeouts and errors may occur.

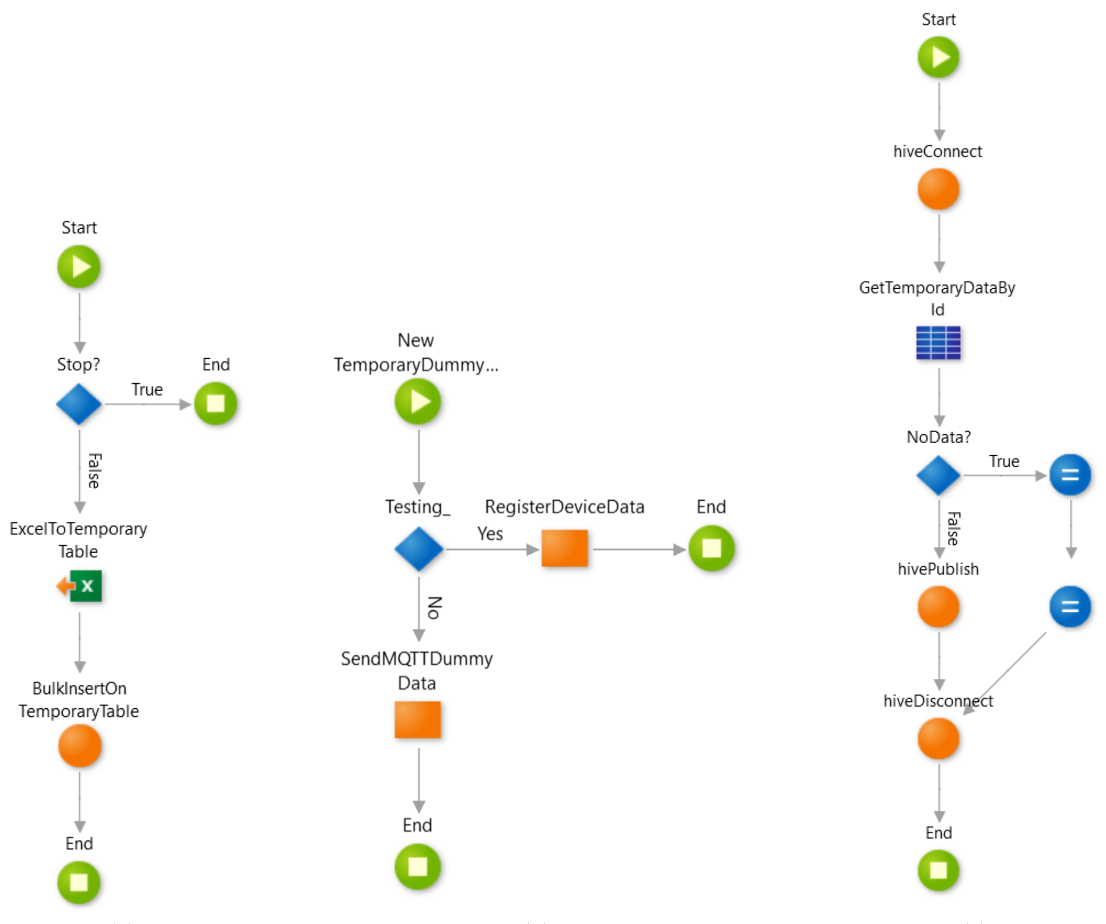
One way to cope with this is to use Heavy Timers [60], where a batch of data is processed at a time, cached and, at the same time, the timer's timeout is controlled and when close to the timeout, the timer is waked up again to avoid unprocessed data. A novel way to deal with the timeout issue is to use LBPTs. For performance reasons, it is preferable to insert all of the `xlsx`-data at once on the Outsystems DB. For that reason, the Forge (see Section 2.4.3) Component BulkInsert [1] was used.

The way LBPTs differ from the traditional BPTs is that the former do not require tracking and, therefore, they run much faster [61]. Another difference is that LBPTs do not allow Output parameters and Callback actions [62][63]. Other than event handling; large scale batch processing; and replacing timers without schedule; LBPTs also are suited for background processing [62][63] (the functionality that will be used in this case, to get the data from the temporary table and do some background processing, namely send data via MQTT). To transform a BPT into a light BPT, one needs to do the following [61]: create an event triggered by a DB event, namely, when creating an entity record (the property Expose Process Entity must be set to "No"); the process must be simple, only containing one Automatic Activity and finally, on the Service Center (see Section 2.4.3), enable the Light Process Execution on the module, after which, the module must be republished, so that the changes take effect (a compilation message will confirm that the process is now lightweight).

In this sense, an Automatic Activity was included in the Process flow, connecting to the MQTT broker, fetching the temporary entity record associated with the process and after, sending the retrieved JSON to the aforementioned broker. It is important to note that, on the Automatic Activity (Figure B.2c), there is an action to connect to the MQTT broker, after which the temporary table record is fetched, based on its Id, available on the process scope; and, if there is no record on the table, one simply disconnects from the broker; otherwise, the JSON is sent before disconnecting.

Also, on the Figure B.2a, where the timer logic is depicted, it is possible to see that an Outsystems' action called `ExcelToRecordList` is used to ease the convertion of an Excel file to the temporary table's record. For the Bulk Insert action, however, it was required to convert the record list to an object, by using the Outsystems' function `ToObject`.

[1] https://www.outsystems.com/forge/component-overview/1117/bulkinsert



(a) (b) (c)

Figure B.2: Logic to load and send "sensors' data" generated *a-priori* with (a) being the timer logic to convert the excel data into the temporary entity record and further, bulk insert it into the temporary table; (b) is the LBPT, with an Automatic Activity inside and finally; (c) with the Automatic Activity that gets the record from the temporary table and sends its JSON via MQTT. Note that the two assigns in c) are simply a trick to join the if True branch with the hiveDisconnect action.

APPENDIX C

PROTOTYPE EMBEDDED CODE

The code depicted in Listing C was uploaded to the Arduino. This contains, both the code to connect to the MQTT broker (review Section 6.2), but also, to retrieve sensor data and send that data via MQTT and to retrieve messages to change the status of actuators. Only the code for some transducers is depicted in Listing C.

Listing C: Arduino Embedded Code

```
#include <string.h>
#include <TimeLib.h>
#include <SoftwareSerial.h>
#include <PubSubClient.h>
#include <WiFiEspClient.h>
#include <WiFiEsp.h>
#include <dht11.h>

/*WiFi Connection*/
const char* ssid = "example_ssid";
const char* password = "example_password"; // WIFI network PASSWORD

/*MQTT*/
const char* mqttServer = "<SRVR>.<HOST>.com";
const int mqttPort = "<PORT>";
const char* mqttUser = "<USER>";
const char* mqttPassword = "<PWD>";

/****Transducers****/
String deviceA_decimal[1] = { "moisture" };
String deviceA_boolean[3] = { "button1", "rgb", "leds1" };
String deviceB_decimal[3] = { "dht11_temp", "dht11_humidity", "potentiometer" };
String deviceB_boolean[2] = { "led2", "button2" };
String deviceC_decimal[1] = { "ldr" };
String deviceC_boolean[5] = { "piezo", "leds3", "button3", "button4" };

/*Time*/
#define TIME_HEADER "T" // Header tag for serial time sync message
#define TIME_REQUEST 7 // ASCII bell character requests a time sync message

void processSyncMessage() {
  unsigned long pctime;
  const unsigned long DEFAULT_TIME = 1357041600; // Jan 1 2013

  if(Serial.find(TIME_HEADER)) {
     pctime = Serial.parseInt();
     if( pctime >= DEFAULT_TIME) { // check the integer is a valid time (greater than
         ↪ Jan 1 2013)
       setTime(pctime); // Sync Arduino clock to the time received on the serial port
     }
  }
}

time_t requestSync() {
  Serial.write(TIME_REQUEST);
  return 0; // the time will be sent later in response to serial mesg
}

String printDigits(int digits){
  // utility function for digital clock display: prints preceding colon and leading 0
  if(digits < 10)
    return "0" + String(digits);
  return String(digits);
}

String returnTimestamp() {
  // digital clock display of the time
  String timestamp = "#";
  timestamp += year();
  timestamp += "-";
  timestamp += printDigits(month());
  timestamp += "-";
  timestamp += printDigits(day());
  timestamp += " ";
  timestamp += printDigits(hour());
  timestamp += ":";
  timestamp += printDigits(minute());
  timestamp += ":";
  timestamp += printDigits(second());
  timestamp += "#";

  return timestamp;
}

```

```
String returnJSON(String transducerName, String value) {

  String json = "";
  json += "{'transducer':'"+transducerName+"','value':'";
  json += value;
  json += "','timestamp':'"+returnTimestamp()+"'},";
  delay(100);

  return json;
}

/*ESP-01*/
WiFiEspClient espClient;
PubSubClient client(espClient);
SoftwareSerial soft(2, 3);
int status = WL_IDLE_STATUS;

/*Messages*/
unsigned long lastMsg = 0;
#define MSG_BUFFER_SIZE (400)
char msg[MSG_BUFFER_SIZE];
int value = 0;

/****Transducer Pins****/
int moisture_pin = A0;
int ldr_pin = A5;
int dht_pin = 5;
int led_pin = 4;
int led_pin_2 = 0;
int led_pin_3 = 1;
int rgb_pin = 10;
int buttons_pin = A3;
int potentiometer_pin = A1;
int piezo_speaker_pin = 6;
dht11 DHT11;

void InitWiFi(){

  delay(10);

  soft.begin(9600); // Initialize serial for ESP module
  WiFi.init(&soft); // Initialize ESP module

   Check for the presence of the shield
  if (WiFi.status() == WL_NO_SHIELD) {
    Serial.println("WiFi shield not present");
    while (true);
  }

  Serial.println(F("Connecting to AP "));
  while ( status != WL_CONNECTED) {
    Serial.print(F("Attempting to connect to WPA SSID: "));
    Serial.println(ssid);

    // Connect to WPA/WPA2 network
    status = WiFi.begin(ssid, password);
    delay(500);
  }
  Serial.println(F("Connected to AP"));
}

/*********************************/
/******* Receive Topics *********/
/*********************************/

/*
 * /outsystems
 * /device
*/

void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.println("] ");
  boolean IsValidTopic = false;

  for(auto x : deviceA_decimal) {
    Serial.println(x);
    x.toCharArray(msg, MSG_BUFFER_SIZE);
    IsValidTopic = IsValidTopic or strcmp(strcat("/device/", msg), topic) == 0;
    if(IsValidTopic)
        break;
  }

  /* A portion of the code was commented. */

}

void reconnect() {
  Serial.println("Connecting to MQTT...");
  if (client.connect("ESP8266Client", mqttUser, mqttPassword )) {
    Serial.println("connected");
    client.subscribe("inTopic");
  } else {
    Serial.print("failed with state ");
    Serial.print(client.state());
    delay(2000);
  }
}

```

```
void setup() {
   Serial.begin(9600);

   setSyncProvider(requestSync); //set function to call when sync required
   Serial.println("Waiting for sync message");
   delay(1000);

   /*********************************/
   /******* Initialize Pins *********/
   /*********************************/
   pinMode(ldr_pin, INPUT);
   /* A portion of the code was commented. */
   pinMode(led_pin, OUTPUT);
   /* A portion of the code was commented. */

    InitWiFi();
    client.setServer(mqttServer, mqttPort);
    client.setCallback(callback);

    for(unsigned int i = 0; i<4294967295; i++) {
       snprintf (msg, MSG_BUFFER_SIZE, "%u", i);
       client.publish("outTopic", msg);
    }
}

void loop() {

   if (Serial.available()) {
      processSyncMessage();
   }
   if (timeStatus()!= timeNotSet) {
      returnTimestamp();
   }
   delay(150);

   if (!client.connected()) {
      reconnect();
   }
   client.loop(); // This should be called regularly to allow the client to process
       ↪ incoming messages and maintain its connection to the server.

   /* A portion of the code was commented. */

   /***************/
   /*** Device 3 ***/
   /***************/
   /* (...) */
   json = "{'device':[";

   /*** Potentiometer (Sensor) ***/
```

```
    json += returnJSON("potentiometer", (String)analogRead(potentiometer_pin));

    json += returnJSON("buttons1", (String)analogRead(buttons_pin));

    json += returnJSON("ldr", (String)analogRead(ldr_pin));

    json += returnJSON("leds1", "1");
    json += returnJSON("leds2", "1");
    json += returnJSON("leds3", "1");

    digitalWrite(led_pin, HIGH);
    digitalWrite(led_pin_2, HIGH);
    digitalWrite(led_pin_3, HIGH);
    delay(200);
    digitalWrite(led_pin, LOW);
    digitalWrite(led_pin_2, LOW);
    digitalWrite(led_pin_3, LOW);

    json += returnJSON("leds1", "0");
    json += returnJSON("leds2", "0");
    json += returnJSON("leds3", "0");

    json += returnJSON("piezo", "1");
    tone(piezo_speaker_pin, 1000); // Send 1KHz sound signal...
    analogWrite(rgb_pin, 150);
    delay(150); // ...for 1 sec
    noTone(piezo_speaker_pin); // Stop sound...
    analogWrite(rgb_pin, 0);
    delay(150); // ...for 1sec
    json += returnJSON("piezo", "0");
    json += "]}";
    //Serial.println(json);

    json.toCharArray(msg,MSG_BUFFER_SIZE);
    client.publish("device/deviceC", msg);
}
```