Smart Behaviours for Smart Homes

José Augusto de Góis Rodrigues de Sá

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Renato Jorge Caleira Nunes

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha

Supervisor: Prof. Renato Jorge Caleira Nunes

Member of the Committee: Prof. Alberto Manuel Ramos da Cunha

October 2019

Acknowledgements

First, I would like to thank my family. My father for always pushing me to do more and to be the best I can be. My mother for all the unconditional love and support and for helping me never give up on my dreams. My sister for all the patient, love and support. Finally, my twin brother, for all the strength, knowledge and support that you gave me since we were born.

Next I would like to thank my supervisor, Professor Renato Nunes for all the help and knowledge during the Thesis and for all the hours and patience spent.

Next I would like to thank my friends, Diogo and Catarina. Thank you for supporting me and it makes me happy to see all of us ready to start professional lives that will be full of success for sure!

I would also like to thank my friends and roommates in Lisbon, Yuri, Rui and Daniel. Thank you for welcoming me with open arms and for all the fun times we had during this journey that is University.

Finally, and most importantly I want to thank my girlfriend Madalena. Thank you for all the hours you spent by my side helping me and always pushing me to do more and better. Thank you for all the hours spent at Técnico where I wanted to just give up and go home and you were always there to put my mind back in its place. Thank you for your advices, knowledge and strength. Thank you for making me a better person, for being my best friend, the love of my life and for loving me like I love you.

Abstract

Nowadays home automation systems represent an important market. Everyone wants to have the power to control their home since these systems offer comfort, security and good support for energy management. There are a variety of home automation systems available in the market and, although there has been evolution related to the interaction between the user and the system, most of them do not offer good solutions regarding flexible, generic, and powerful automation mechanisms that can be applied in a simple way.

In this work a solution will be presented, in the context of the DomoBus system, which uses the concept of "Automation Block". Automation Blocks are software entities that have various inputs and outputs and perform some predefined function. Automation Blocks can be very powerful and, because of that, quite complex. So, a web application was developed that fully supports the development of Automation Blocks, which is expected to be done by knowledgeable users. Those Automation Blocks can then be shared and applied to real systems. This last task can be performed by common users, which just have to select which Automation Blocks to apply and associate real devices to their inputs and outputs. This is also supported by the developed application, allowing users to easily customize the behavior of their homes and benefit the most from their home automation system.

Keywords: DomoBus, DomoBus Automation Block, Home Automation system, Automation mechanisms, Home Behaviour, Ruby On Rails, Vue.js.

Resumo

Atualmente, os sistemas de automação para casas representam um mercado importante. Toda a gente quer ter o poder de controlar a sua casa, pois esses sistemas oferecem conforto, segurança e uma forma simples de gerir os gastos de energia. Existem diversos sistemas disponíveis no mercado e, embora tenha havido evolução relacionada à interação entre o utilizador e o sistema, a maioria deles não oferece boas soluções nem mecanismos de automação flexíveis, genéricos e poderosos que podem ser aplicados de maneira simples.

Neste trabalho, será apresentada uma solução, no contexto do sistema DomoBus, que utiliza o conceito de "Bloco de Automação". Os blocos de automação são entidades de software que possuem várias entradas e saídas e executam alguma função predefinida. Os blocos de automação podem ser muito poderosos e, por isso, bastante complexos. Assim, foi desenvolvida uma aplicação Web que suporta totalmente o desenvolvimento de Blocos de Automação, que se espera que seja feito por utilizadores que tenham conhecimento dos mesmos. Os blocos de automação podem ser aplicados a sistemas reais. Esta última tarefa pode ser executada por utilizadores comuns, que precisam apenas de selecionar quais blocos de automação aplicar e associar dispositivos reais às entradas e saídas. Esta tarefa também é suportada pela aplicação desenvolvida, permitindo que os utilizadores personalizem facilmente o comportamento das suas casas e beneficiem ao máximo do seu sistema de automação.

Keywords: DomoBus, Bloco de automação DomoBus, Sistema de automação para casas, Mecanismos de automação, Ruby On Rails, Vue.js.

Table of contents

Table of Figures viii

Table of Tables x

List of Acronyms xi

1 Introduction 1

2 Related Work 3

2.1 Insteon 3

2.1.1 Protocol and Modulation 3

2.1.2 Products 5

2.1.3 Control 6

2.2 IFTTT 7

2.2.1 Protocol and Modulation 7

2.2.2 Control 7

2.3 Amazon Echo 11

2.3.1 Protocol and Modulation 12

2.3.2 Control 12

2.4 Conclusion 13

3 DomoBus 14

4 Proposed Solution 18

4.1 Input aggregation 19

4.2 Default inputs 19

4.3 Input type 19

4.4 Input condition 20

4.5 Input definition 20

4.6 Output Definition 21
4.7 Timer Definition 21
4.8 State Components Definition 21
4.8.1 Condition Definition 21
4.8.2 Action Definition 22
4.8.3 State Definition 23
4.9 Instantiation Process Definition 23
4.10 Configuration 24
4.11 Solution 25
5 Implementation 27
5.1 Architecture 27
5.2 Users 28
5.3 Authentication 28
5.4 XML File 30
5.5 Adding a house 31
5.6 Creating a Domobus Automation Block 32
5.7 Write the block to a file 38
5.8 Editing a Domobus Automation Block 39
5.9 Connecting house's devices to a Domobus Automation Block 43
6 Evaluation 47
7 Conclusion 48
8 References 49

Table of Figures

Fig. 1: Insteon Architecture 4
Fig. 2: Applet "Welcome Home" from IFTTT platform.[12]....8
Fig. 3: Stringify setup before setting the connections. [19]....10
Fig. 4: The final setup. [19] 11
Fig. 5: Smart Home Skill API....12
Fig. 6: DomoBus Control Level [13]....14
Fig. 7: DomoBus Supervision Level [13]....15
Fig. 8: Example of different properties values....16
Fig. 9: Example of devices definition....17
Fig. 10: Representation of a DomoBus Automation Block [9] 18
Fig. 11: Simple architecture for proposed solution....25
Fig. 12: Simple Vue.Js component. 26
Fig. 13: System Architecture. 27
Fig. 14: JWT diagram. [14]....29
Fig. 15: authorize_request method....30
Fig. 16: XML example file 30
Fig. 17: Home page of the system. 31
Fig. 18: Small example of the processing of a house XML....32
Fig. 19: State machine diagram. 33
Fig. 20: Second step of the create algorithm - Inputs 34
Fig. 21: First State - “Off” 35
Fig. 22: Second State - “On”....36
Fig. 23: Third state "Waiting"....37
Fig. 24: State object organized hierarchically.....38
Fig. 25: DomoBus Automation Block Entity-Relationship Diagram.....39
Fig. 26: Pencil symbol to edit a block and the button to create a block. 40
Fig. 27: Input object processed in the front-end. 42
Fig. 28: View of a house.....44
Fig. 29: View of a division. 45

Fig. 30: Input Presence with two properties connected. 46

Table of Tables

Table 1: Modulation properties used by Insteon.....5

List of Acronyms

DAB Domobus Automation Block

JWT Json Web Token

PDA Personal digital assistant

SDM SmartLabs Device Manager

XML E**X**tensible **M**arkup **L**anguage.

1 Introduction

The desire to control devices in our home, or even remotely program them to execute a specific task at a given time, is a growing interest nowadays. People are constantly searching for ways to ease everyday tasks, like having a front gate that opens by itself and lights that turn on automatically when the front door is opened. The possibility to control and manage resources is something that many people are searching for. The capability to manage energy consumption or even to control a living space from distance is growing every day in today's market and will only continue to grow, not only because of user demand, but also because of the various home automation systems that are appearing every day that offer these possibilities.

Home automation systems allow users to take control of their homes and perform several tasks. With this type of system, the user can easily control his house using, for example, a smartphone. There is a big variety of systems in the market, such as Insteon, Amazon Echo, IFTTT, but most of them do not offer good solutions regarding flexible, generic, and powerful automation mechanisms that can be applied in a simple way. When we look to what the market has to offer, various systems allow the user to create scenarios and execute commands via smartphone, or through voice, among others. Some systems offer fully automated tasks but only when integrated between themselves. Domobus Automation Blocks (DAB) are blocks that are part of the DomoBus system. This block works like a state machine that accepts inputs that give the user the possibility of connecting several devices to a single device by using combinatory logic. This feature prevents the user from using several identical algorithms for devices that will run equal tasks. Even though DomoBus allows this feature, and much more, it presents a problem. The way the blocks are created, defined and used is very complex and hard for the common user to do by himself. This is where Smart Homes comes in.

Smart Homes is a web application focused on encapsulating all the complexity associated with the definition of DomoBus Automation Blocks. It is targeted for two types of users. One type of user represents someone acquainted with the system that has the required programming skills needed to create a DomoBus Automation Block. The other is the common user that will use the system to fully automate his home.

An Automation Block is defined in a very generic way and has one or more inputs that, combined or not between them, generate one or more outputs. Those inputs can take the form of, for example, a switch, a thermostat, a movement detector, etc. An output is an action over those properties, for example, “turning on” of the switch if a movement is detected. The interface should allow the second user to map his home based on an XML file and to choose a block, from a list of already defined blocks, to apply in a division, specifying which real devices connect to the inputs and outputs of the block.

Further in this document, it will be analysed what the market has to offer and how DomoBus differentiates itself from the competition.

2 Related Work

Home automation technologies and Internet of Things are growing areas in today's market. In the last couple of years, several systems appeared, all with different kinds of possibilities, but not all offering the user a realistic opportunity to totally control an action using devices present in his home. Further into this work, some of the most known home automation systems will have their pros and cons analysed when it comes to allow the users to create their own actions.

2.1 Insteon

Insteon is a home automation technology created in 2007 by SmartLabs, a company based in USA, that provides the user with the possibility to control a wide variety of electronic powered devices such as thermostats, lights, remote controllers and others. Insteon devices use Radio Frequency, Powerlines, or both, to communicate with each other making it a very reliable system. Each device acts like a peer which let them operate independently from one another, being able to receive and send messages.

2.1.1 Protocol and Modulation

Insteon uses is own protocol, with the same name, to communicate between devices using a powerline, radio frequency or both, as described before. In Fig. 1 we can see the devices that communicate over powerline and over radio frequency [1]. If the device PL-1 wants to communicate with RF-1, then the message will be sent through DB-1, since they are not directly connected.

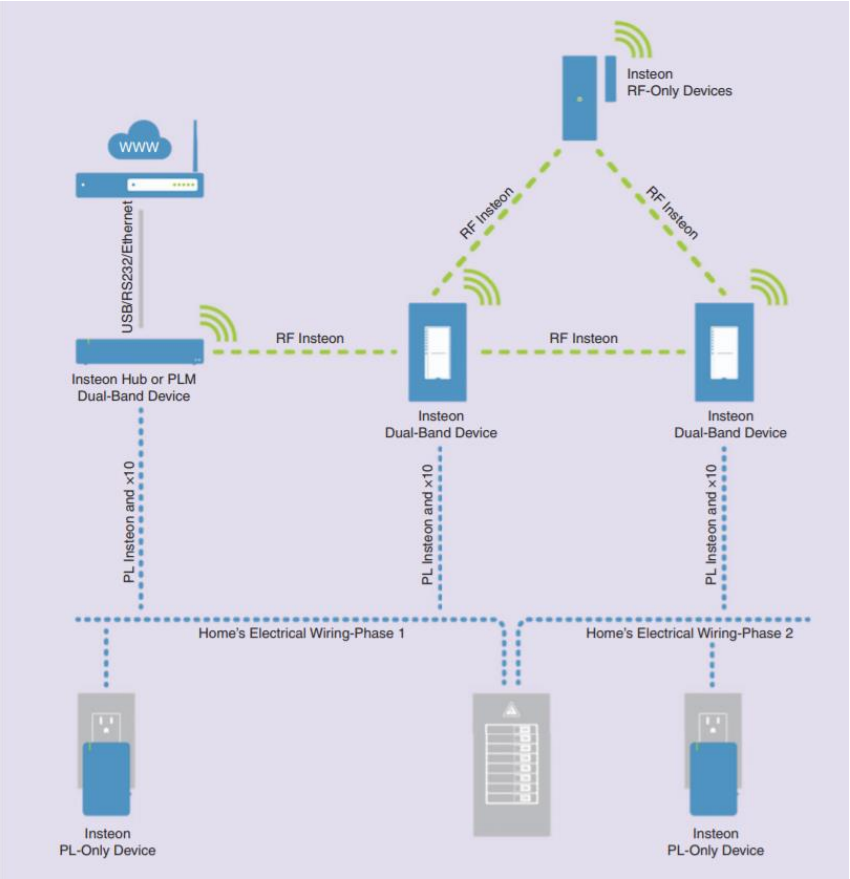

Fig. 1: Insteon Architecture

Insteon is compatible with X10 devices. Regarding the modulation, it uses FSK for Radio Frequency and BPSK for Powerline (see Table 1) [2].

		RF	PL
Data Rate	**Instantaneous**	38,400 bps	13,165 bps
	Sustained		2,880 bps
Modulation		FSK	BPSK
Frequency		915 MHz (US) 869.85 MHz (Europe) 921 MHz (Australia)	131.65 KHz

Table 1: Modulation properties used by Insteon

2.1.2 Products

Insteon offers a variety of products to the users. They are included in 5 categories: Insteon Hub, Wall Switches, Wall Keypads, Plug-In Devices and Wireless sensors. Each one of these categories offer different products with different applications [4].

The Insteon Hub is a small box that connects the user smartphone to the house. Using the Insteon application, the user sends requests to the hub that processes them and triggers responses in the devices. It costs 80 dollars.

Wall switches are devices that allow the user to turn on/off or control the brightness of a light. Their cost varies between 50 to 55 dollars each.

Wall keypads are devices that have a group of small buttons that the user can configure to perform different actions. A wall keypad with 4 buttons can activate a different scene for each one. A scene is an action that the user creates using his smartphone, that allows the possibility of dispatching several responses with only one action. This is useful if a user wants to, for example, turn on all the lights in a room with only one action. Wall Keypads have a price of 80 dollars.

Plugin-In devices are able to transform the table lamps presents in the user's home into an Insteon device that can be controlled using a smartphone. They have a cost of 50 dollars.

Wireless Sensors are devices that can be used to trigger automated actions previously programmed. Some sensors available are open/close sensors, hidden door sensors, motion sensors, among others. The prices vary between 30 and 40 dollars.

2.1.3 Control

In order to successfully use Insteon, the user needs to buy an Insteon Hub and, for example, two plug-in devices to connect two table lamps present in the user's home. Insteon offers a diversity of commands that include scenes that let the user manage one or more devices with a single command, the creation of schedules to, for example, turn on or off a light in the house at certain hours and push up notifications to let the user know when a device changes state.

If the user wants both lights to be on at the same time, it is possible to create a scenario where both lamps are added, and it is chosen if they will act has a controller or a responder. Controllers activate the scene while responders are activated when the first happens. In this case, both lamps will be responders because what will trigger the action is the users click in the application (controller).

An advantage of Insteon is the fact that, with a couple of modules, it is possible to create an automated action that allows, for example, to turn a light ON when a door is opened. Despite its strong aspects, Insteon modules can't be reused, meaning if a user adds a new switch or a new lamp to a room, it will be necessary to buy new modules and define a new logic to them.

Since the actions that Insteon presents are limited, Insteon Application Development Overview was developed. Insteon Application Development Overview allows developers to build and create applications that can be internal or external. Internal applications can be programmed directly on the device while external applications are programmed on a PC such as a Raspberry Py or a PDA. These connection between the hardware and the Insteon devices is made using an Insteon Bridge [18].

The Insteon Manager App is an external application that connects to the Insteon network via an Insteon bridge. An example of an Insteon Manage App is the SmartLabs Device Manager (SDM) that encapsulates all the complexity behind the messages that are sent to devices, so the

user only needs to focus on creating their application layers. SDM offers an interface that allows developers to connect their custom applications by following a series of commands to communicate with them using simple messages via HTTP calls. Some of the available commands care:

- **port=(COM#|USB4|SIM28|?)** - Sets the port on the device. It can be COM# (# equals to a number. Ex: COM1, COM2, …), USB4, SIM28. The question mark is used to search for ports.

- **isResponding** - SDM returns if a port is responding or not.

2.2 IFTTT

IFTTT is a software that connects mostly two services or apps through the programming idea of "If This Then That". The automation between them is made with actions called "Applets" that a user can create. Nowadays IFTTT works with hundreds of services like Alexa from Amazon, Facebook and NASA.

2.2.1 Protocol and Modulation

It is possible to communicate with any applet available in IFTTT using, for example, an Arduino through MQTT protocol. MQTT is a small but powerful publish/subscribe message protocol for small devices that works on top of the TCP/IP Protocol. The way it works seems very simple despite the lack of information on this matter. Using MQTT, the user publishes the first service and subscribes to the second one [3].

2.2.2 Control

IFTTT is free of use and all the user needs to do is to register in their platform or their mobile application. After that the user gains access to thousands of different applets that can be subscribed and used. In order to subscribe to an applet, the user is obligated to connect the services that the applet calls to his IFTTT account. It is possible to create applets by simply combining various app services and setting parameters that will trigger the action.

In the platform there are several applets available related to home automation. Although some of them lack useful applications, others offer a variety of different actions for the user. For example, an applet called “Welcome Home” turns on the heater and the hue lights when the user arrives home. For this it uses 3 services, the GPS location of the user, the MiGo service that let you control the heat in your home and the Philips Hue a service that lets you control the lights in your house. This is a good automated action but involves, at least, two different parties that the user is obligated to be involved in: Philips and MiGo [12] (Fig. 2).

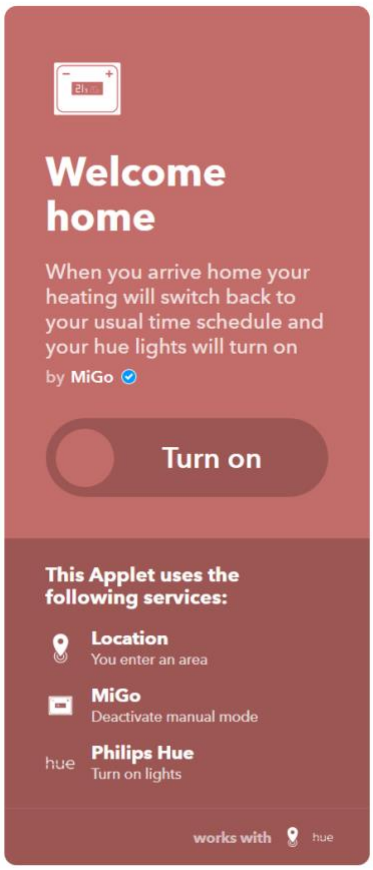

Fig. 2: Applet "Welcome Home" from IFTTT platform.[12]

IFTTT also allows the user to control their API to communicate with them. Each trigger and action will have a 1-1 endpoint with IFTTT and that is how the communication occurs. The endpoint must be exposed if the application is running on a local machine. The user can create triggers and actions in a really simple way, for example *{{user_api_prefix}}/ifttt/v1/triggers/receive_email*, is called when the systems detect that an email has been received, and *{{user_api_prefix}}/ifttt/v1/actions/save_to_google_drive*, is the action that is called and saves on the google drive the attachments that are present in the email [17]. Now all the user needs to do is expose the endpoints, by using ngrok for example, to communicate with IFTTT. This IFTTT Platform gives the user some nice features and possibilities such as using a movement sensor as a trigger and a lamp turning on as an action. The downside is that one action only accepts one trigger so, for example, it is impossible for a user to connect a movement sensor and a light sensor to the same lamp (and action) because the applet only accepts one trigger. Nevertheless, there are some ways the user can overcome this issue. One of the possibilities is using Stringify to merge all the triggers in one.

Stringify is an application that allows the user to link a number of triggers or actions that will run in a flow system, one after the other [19]. Works the same way as IFTTT but as other limitations. This app gives the user brand new possibilities to integrate with IFTTT. Next it will be explained how it is possible to have two triggers in an IFTTT applet using Stringify. In this case, we will illustrate how to create the automation required to switch a light on when a motion sensor detects movement in the dark [22].

First Abode is added as a service in the IFTTT app. Abode is a system that allows the creation of complex automation tasks. Next IFTTT is added as a “thing” in the Stringify app. A “thing” is like a service. The next step involves installing all the components needed to create the service. The components are the Date&Time thing, the Mode thing and the IFTTT thing. The IFTTT thing is what allows Stringify to execute an applet if the triggers conditions are met. It is also required that the devices needed to perform the action are also installed and correctly configured. In this case the motion sensor and a lamp. To detect night time (dark), it could be used a light sensor but, in this case, will only be used date and time. After all is configured, the user creates a flow through the Stringify app and adds the things that will be used to perform the action. In Fig. 3 is possible to see what the user should end up at this point.

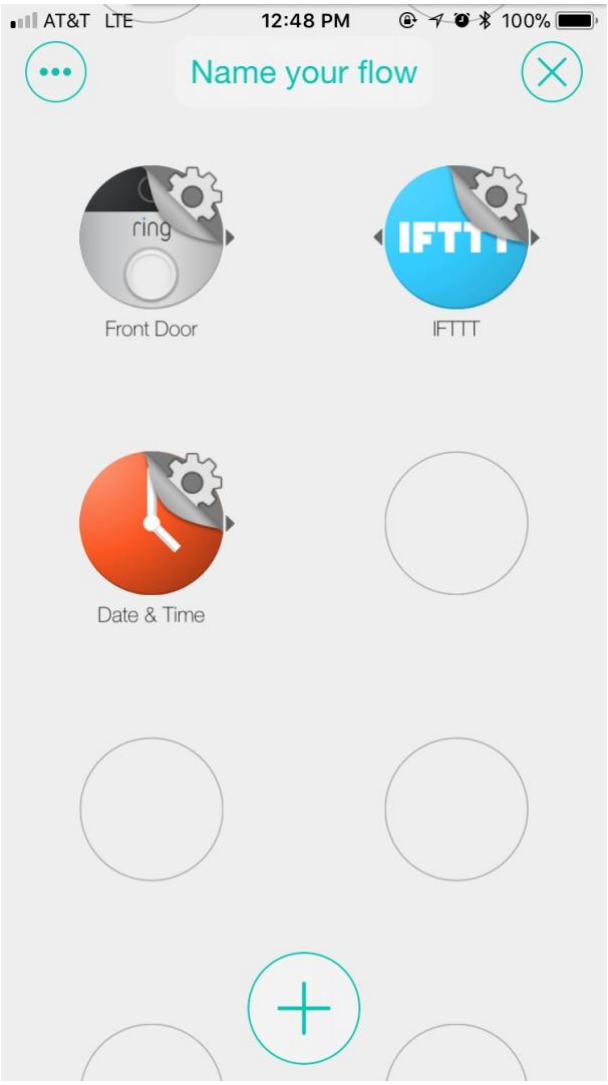

Fig. 3: Stringify setup before setting the connections. [19]

The only thing left to do now is to associate the things and for that, the user just needs to click on which thing and decide what trigger it should perform. In this case we will set the front door (where the motion sensor is) to trigger if it senses movement, the date/time to trigger at night time and the IFTTT thing to run an applet. Finally, the user must draw a line between things and done! The automation is made and complete. (Fig. 4).

Fig. 4: The final setup. [19]

As seen, IFTTT offers good solution when creating automation tasks but it is dependent of outside services to accomplish that.

2.3 Amazon Echo

Echo is a smart speaker created by the US company Amazon. It recognizes voices and can perform several tasks such as tell the weather, read books, set alarms, among others. It can be used as a controller for a home automation system such as Insteon [5].

2.3.1 Protocol and Modulation

Amazon presents the user with a wide range of connectivity possibilities. To connect devices locally it uses the ZigBee, a wireless technology that uses the IEEE's 802.15.4 personal area network (WPAN) to communicate with others ZigBee devices. Those devices cannot be more than 15-20 meters apart from each other. The WPAN present in ZigBee operates in three frequencies, 2.4 GHz, 900 MHz and 868 MHz and, just like Insteon, ZigBee allows mesh networking between Local Area Network (LAN), Virtual Local Area Network (VLAN) or Wireless Local Area Network (WLAN) [6].

Other option provided by Amazon is connecting a device control cloud to Alexa using the Smart Home Skill API. Smart Home Skill API allows the user to give voice commands to Alexa that describes devices present in the cloud and properties or events that they support. Alexa uses that information to trigger a search for the specific device. Furthermore, Alexa also supports the capability of providing updated information to the user about a specific device when it receives a command with that objective [7] (Fig. 5).

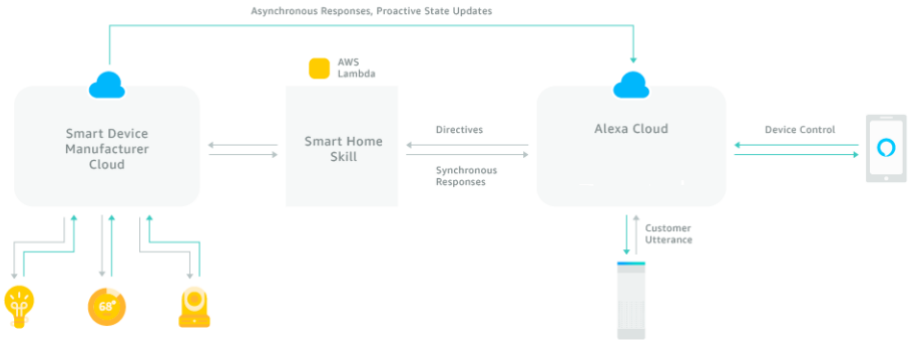

Fig. 5: Smart Home Skill API

2.3.2 Control

Amazon Echo, when connected with other services, can practically do anything the user wants. If the user wants to turn on a light in a room, he only needs to connect to a service that allows that kind of action and then create a voice command to execute it. Although this seems

good when we talk about smart homes, Echo lacks some automation mostly because the user must execute a voice command to start a service. Echo Auto is a new product by Amazon, based on the Echo, that can execute an automated action using GPS signal [8].

Echo Auto is a portable device that the user installs in his car. When connected with a service that lets the user control the lights of his home, like Philips Hue, it is possible to turn some light on based on the GPS signal of Echo Auto. Echo costs between 50 dollars and 120 dollars, depending on the model, and Echo auto around 50 dollars.

Echo allows the user to create their own skills. Skill is an action that the user commands Alexa to execute using his voice. There are several ways a user can create a skill. The most known is the one already described, the Smart Home Skill API since it is really simple to use, making it user-friendly as many common users lack deep knowledge about this subject. Because it creates the interaction and the commanding words itself, the Smart Home Skill API only creates skills that are defined to connect with some known smart device, light a lamp, a thermostat, etc [20]. If the user wants something different and have more control of a skill, a custom skill can be created that allows, for example, to search for information in a web service and connect to one to order something specific such as buying tickets to Dubai from Emirates [21]. Although the Alexa Skill Set allows the user to create custom skills, it always depends on voice command in order to start it.

2.4 Conclusion

Most home automation systems available nowadays in the market can’t do proper automation tasks and don’t offer good systems that allow the user to create their own tasks from scratch. Most of them achieve an automated action by combining their systems such as Amazon echo auto and Philips Hue, IFTTT and Philips Hue, etc. Insteon, among the systems analysed, is the one that provides better automation actions but has the problem of having to create a new logic every time a new component is added to a division of the property.

The solution proposed, and that it will be analysed next, is generic and that's how it differentiates itself from the systems mentioned above.

3 DomoBus

DomoBus is a home automation system created by Professor Renato Jorge Caleira Nunes [10]. It was developed as an academic project with the goal of being a simple but extremely powerful and expandable system. The way this can be achieved is with its distributed architecture involving modules, each one with the power to control several types of devices. With this approach, DomoBus is actually a very good solution when it comes to making an extremely reliable automated home.

DomoBus consists essentially of Console Modules and Supervision Modules that communicate with each other over a network. A console module is a simple processor that can be connected to power lights, movement sensors, thermostats, among others, and communicates directly to only one supervision module. A supervision module controls and, as the name suggests, supervisions the system. It receives rules, defined by the user, and uses them to process requests received by the console modules and send commands as response. The proposed solution will be done on the supervision module.

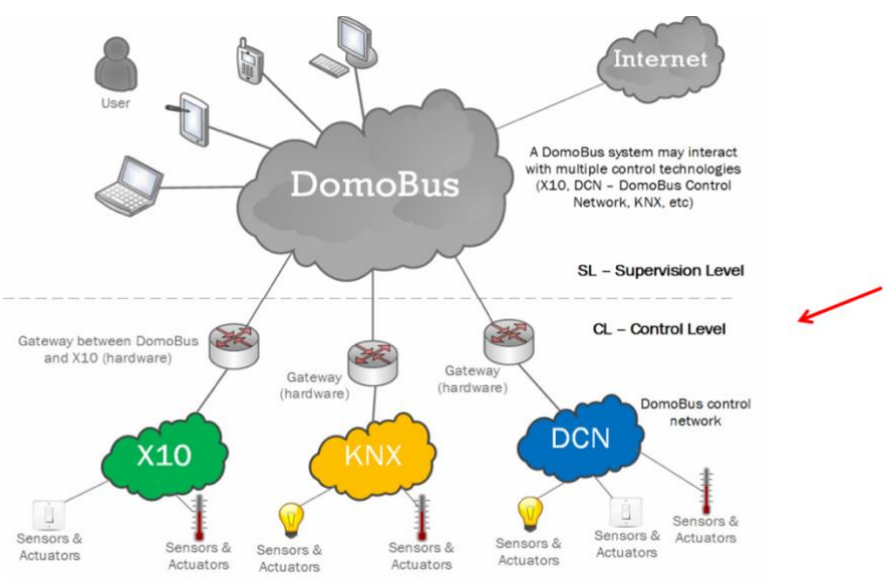

Fig. 6: DomoBus Control Level [13]

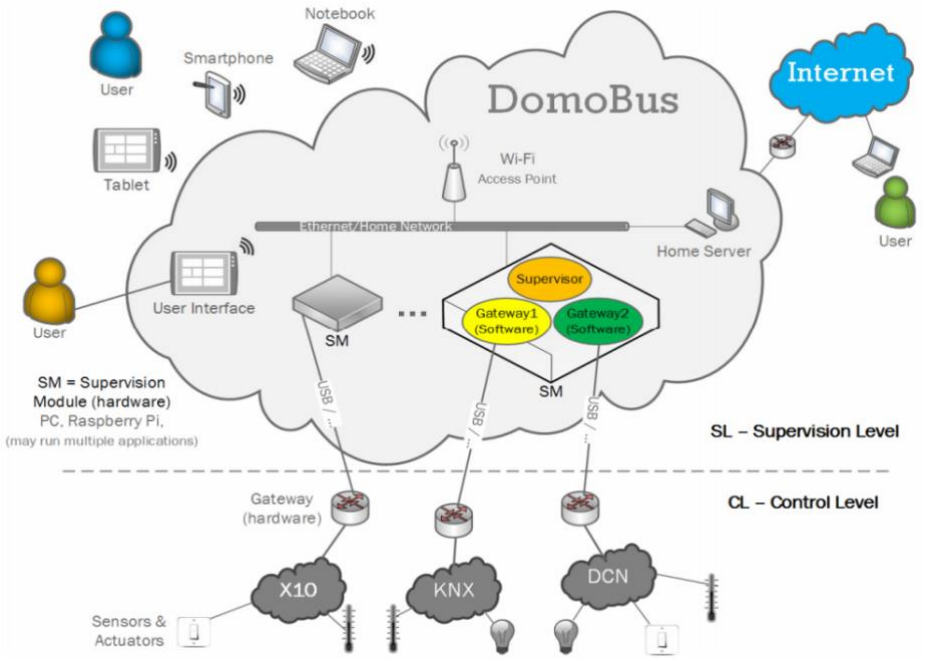

Fig. 7: DomoBus Supervision Level [13]

There are 3 type of messages that write (SET), read (GET) or inform of a new value (NOTIFY) of a device that is included in the DomoBus Protocol. Every DomoBus system is specified by an XML-based document divided in two parts [11]. One that allows the specification of devices and their properties and other that specifies the house and all their components. XML allows the user to define generic applications that read it and fit the system. The hierarchy accepted by the XML is House > Floor > Division > Device. Every device is a generic entity that is characterized by properties depending on the device. For example, if we have a light, we can have two properties, one that turns the light ON/OFF and other property that controls the brightness of the light. Each property has a value that can be one of three types:

- **ScalarValueType** - It's an integer that can be represented by 8 bits (from 0 to 255) or 16 bits (from 0 to 65535).
- **EnumValueType** - It's a pair of [name, value] and the value is represented by an 8-bit quantity.
- **ArrayValueType** - It's an array of bytes.

Below, in Fig. 8, we can see an example of how these properties are defined.

```
<ScalarValueTypeList>
    <ScalarValueType ID="1" Name="Percentagem (0-100)"
                                        NumBits="8" MinValue="0" MaxValue="100" Step="1">
        <ValueConversion Type="NONE" Ref="" />
    </ScalarValueType>
    <ScalarValueType ID="2" Name="Potência"
                                        NumBits="16" MinValue="0" MaxValue="800" Step="10">
        <ValueConversion Type="FORMULA" Ref="2" />
    </ScalarValueType>
</ScalarValueTypeList>

<EnumValueTypeList>
    <EnumValueType ID="1" Name="On-Off">
        <Enumerated Name="Off" Value="0" />
        <Enumerated Name="On" Value="1" />
    </EnumValueType>
    <EnumValueType ID="2" Name="Comando Ar Condicionado">
        <Enumerated Name="Desligado" Value="0" />
        <Enumerated Name="Aquecer" Value="1" />
        <Enumerated Name="Arrefecer" Value="2" />
    </EnumValueType>
</EnumValueTypeList>

<ArrayValueTypeList>
    <ArrayValueType ID="1" Name="Nome Empresa" MaxLen="10">
        <ValueConversion Type="NONE" Ref="" />
    </ArrayValueType>
    <ArrayValueType ID="2" Name="Float IEEE" MaxLen="8">
        <ValueConversion Type="OBJ" Ref="1" />
    </ArrayValueType>
</ArrayValueTypeList>
```

Fig. 8: Example of different properties values

A device also has a name and an access mode that can be “RW” - Read and Write, “RO” - Read Only and “WO” - Write Only.

In a low-level, a device is identified by a number with 32 bits and the properties by a property descriptor with 8 bits which is divided in 3 sections: bits from 0 to 4 represent the property ID, bit 5 represents an invalid flag and bit 6 and 7 represent the property type. The property type can be: 8_BIT (b7=0, b6=0, value=0), 16_BIT (b7=0, b6=1, value=64), 32_BIT (b7=1, b6=0, value=128) and D_ARRAY (b7=1, b6=1, value=256). The D_ARRAY is a DomoBus Array that has an additional byte that indicates the size useful data inside the array.

However, in a high level they are defined with XML. Below it is possible to verify an example of devices definitions using the already defined properties in Fig. 9.

```
<DeviceTypeList>
    <DeviceType ID="1" Name="Lâmpada-Regulada" RefDeviceClass="1" Description="-">
        <PropertyTypeList>
            <PropertyType ID="1" Name="On-Off" AccessMode="RW"
                                                    ValueType="ENUM" RefValueType="1" />
            <PropertyType ID="2" Name="Intensidade" AccessMode="RW"
                                                    ValueType="SCALAR" RefValueType="1" />
        </PropertyTypeList>
    </DeviceType>

    <DeviceType ID="2" Name="Sensor-Temperatura" RefDeviceClass="2" Description="-">
        <PropertyTypeList>
            <PropertyType ID="3" Name="Temperatura" AccessMode="RO"
                                                    ValueType="SCALAR" RefValueType="1" />
        </PropertyTypeList>
    </DeviceType>
</DeviceTypeList>
```

Fig. 9: Example of devices definition

4 Proposed Solution

DomoBus Automation Block (DAB) is an automation block, already documented by Professor Renato Jorge Caleira Nunes [9], that can be programmed to perform almost any task that a user desire. The block receives a series of arguments that are processed and executes a result. The arguments are defined as inputs and the result as outputs, allowing the definition of complex rules of automation. In combinatorial logic, the outputs depend on the actual values of the inputs, but in the sequential logic, besides depending on the actual values, they can also depend on past values. This is achieved because the automation block contains a state machine internally that can have one or more states and because of that it can depend on the values of past states. Blocks also contain one or more timers that can be started, cancelled and checked to see if expired.

Finally, a block has one or more states. A state is composed by one or more transitions that are divided into two components: a condition and an action. An expression is a group of conditions. One or more actions are triggered when a condition is met. When this happens, the transition is executed. The actions executed by the transition can be of two types: local or normal. A local action is responsible for changing the current state and manage timers by starting or cancelling them. A normal action only deals with the output changes. It is responsible for changing the values of the outputs.

A block and all its components are defined in text file following some rules that will be explained next.

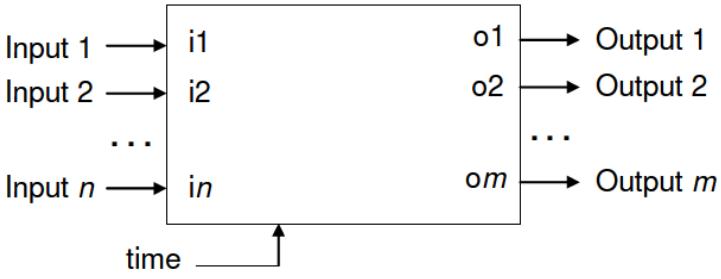

Fig. 10: Representation of a DomoBus Automation Block [9]

4.1 Input aggregation

An automation block can be generically defined since an input can allocate several properties of one type by using combinatory logic with AND or OR operator. For example, one light is connected to two movement sensors, one in every corner of the room. Those two sensors will be connected to only one input using the OR operator if the user decides that it does not matter which of them was activated once they all perform the same action, which is turning on or off the light. Without this feature, those two sensors would need to be defined in separated blocks. This means that the user would need to use a *n* number of blocks for *n* sensors in the room and each block would have the exact same logic as the others. If the user decides that the light must turn on only, and only if all sensors are activated, the input is aggregated with the AND operator because every condition must be true at the same time.

4.2 Default inputs

In order to keep the blocks in the most flexible and generic possible way, they allow inputs to be left open and without any connection. This is useful because the user can use a block that performs the desired action even if one of the properties that the input receives is missing. The value of the default input should be 1 or 0 equivalent to true or false respectively.

4.3 Input type

Like mentioned above, each input of a block is connected to a property type. Those types are represented by the values 0, 64, 128 and a 192 (DomoBus Array). Some properties can work in a mode called “action”. When working in this mode, a property, when activated, triggers an action and then returns to its initial state. An example is a pushbutton switch that, when clicked, returns that is active, 1, and then returns that is inactive again, 0. When a property is signalized as an action, then it should add 1 to its type so it becomes 1, 65 or 129. A DomoBus array can’t work in action mode.

4.4 Input condition

An input can receive any value if it agrees with its type, but the internal logic of a block only operates with Boolean values, true or false. Consequently, an association is added to every input with the purpose of converting them to a Boolean value. The operators available to define those associations are EQ-equal, NE-not equal, LT-less than, LE-less or equal, GT-greater than, GE-greater or equal, DO-true for any no value and INV-true if the value is invalid.

As an example, let's assume a light that only turns on when it is dark. In order to accomplished this, a light sensor is required and a value for dark needs to be defined. By defining that value to 50, for example, it is decided that if the sensor value is less or equal that 50, then it returns true which means an action is triggered and, in this case, a light turns on.

4.5 Input definition

After explaining the main features of the inputs, now will be demonstrated an example of how to define them. Every input definition starts with the letter 'i' and is followed by a number, up to 255. This number works as an ID that identifies every input. After that comes the type identifier: 0, 64, 128 or 192 or if the property works in action mode, as previously mentioned, 1, 65 or 129. Next comes the condition used to create the logic with the property from the input. This condition includes a value and an operator. The last two fields are the aggregation operator (AND or OR) and the default value of the input. Example:

i0 0 GT 50 OR 0 - The following example indicates an input that is identified by the number 0 and has type 0 (8 bits) which means it is an enum or a scalar. The properties will be tested to see if they are greater than 50 and return the respective Boolean value. If there are more than one property of the same type, those properties will be aggregated using the OR operator that will return the value of that input (true/false). So, if two properties are connected to it and one returns true and the other returns false, following the combinatory logic, it returns true. If no property is defined, the default value will be 0, which means false.

4.6 Output Definition

An output is very simple to define. It starts with a letter ‘o’ and receives an ID, just like the input, and a property type that must also be 0, 64, 128 or 192. Example:

o0 0 – The following example indicates an output that is identified by the number 0 and has type 0 (8 bits) which means it is an enum or a scalar.

4.7 Timer Definition

A timer is defined with the letter ‘t’, an ID and an integer value that represents, in seconds, the time that the timer waits. Example:

t0 10 – The following example indicates a timer that is identified by the number 0 and the time value of 10 seconds.

4.8 State Components Definition

As mentioned earlier, a state consists in one or more transitions that are divided in conditions and actions. The definition of the components is done hierarchically and will be explained next.

4.8.1 Condition Definition

A condition is defined with the letter ‘f’, an ID, an operator and one or two operators depending on the operator. The operator’s EQ and NOT only accept one operand. The condition represents a logical expression and uses the Polish prefix annotation. In this annotation the operator precedes the operands. The operands of a condition can be any of the properties of the input as well as other previous conditions.

As mentioned previously, a group of conditions represents an expression. An expression is defined with a letter ‘e’, an ID, and the conditions. Example:

e0 f1: OR *x y* f2: AND f1 *w* f3: NOT f2 – The following example indicates an expression that is identified by the number zero and has 3 conditions: f1, f2 and f3. The first condition is identified by the number one and represents the Boolean value of the property *x* OR the property *y*. The second condition is identified by the number two and it is an example using a previous condition as an operand. Represents the Boolean value of the first condition AND the property *w*. Finally, the last condition, represented by the number 3 uses only one operand because uses the NOT operator. It represents the negations of the Boolean value of the second condition.

4.8.2 Action Definition

An action is not defined by any letter or ID and, as mentioned previously, it can be local or normal. An action represents the definition of local and normal actions, each one ending with a single ‘.’. A change of state and starting or cancelling a timer are local actions. The change of state is represented by the letter ‘s’ and the ID of the state that will be changed. Starting a timer is represented by the letter ‘t’ and the ID of the timer. Cancelling a timer is represented by the letter ‘c’ and the ID of the timer. Changing output values is a normal action and its represented by a letter ‘o’, the ID of the output, the equal symbol ‘=’ and the value of the output that will be changed to 0 or 1. Examples:

s3 c2 . o1=0 . – The following example represents two local actions and one normal action. The first local action indicates that it will change to the state with the ID three and the second indicates that the timer with the ID two will be cancelled. The normal action indicates that the output with ID one will turn false.

s3 c2 . . – This example has the same local actions that the previous example but has no normal action.

4.8.3 State Definition

A state is defined by the letter ‘s’, an ID and all the components that will define it in a hierarchically form. Before the definition of the states, a line is reserved to list the IDs of every state. The line starts with capital letter ‘S’ following the two dots symbol ‘:’ and the IDs of the states. Example:

S: 1 2
s1
e0 f1: OR *x y* f2: AND f1 *w*
s3 c2 . o1=0 .
s2
e0 f3: NOT f2
. o1=1 .

- The following example represents two states. The first line declares the states that will be defined next. The next lines are just the combination of the definitions previously explained with the exception of the line that indicates the start of each state. In this example we define two states with the IDs 1 and 2.

4.9 Instantiation Process Definition

The instantiation process is what generates the final file that has the connections between the block and devices. A text file is generated with several commands when those connections are made. The file is called a template and starts by identifying the block by using the letter ‘u’ followed by its ID. Next the inputs of the block are associated with the chosen device properties. Each line defines one input and starts with a letter ‘i’ followed by its ID. Next, each property linked to the input is defined in three numbers, the first two separated with commas. The first number is the ID that identifies the device which the property belongs to. The second number identifies the property descriptor that was explained in section 3. The last number identifies the property configuration. This property is different between properties types. For

8_BIT, 16_BIT and 32_BIT it represents if the property acts in normal (0) or action mode (1). In case of a DOMOBUS_ARRAY property, the last field represents the size of useful information present in the array. The output definition follows the same analogy that the inputs. The file must always end with the letter ‘e’. Example:

u5
i0 10,0 1 15,1 0
o2 12,0 0
e

- In this example we have a template for a DomoBus Automation Block with ID 5 that has one input with two properties associated and one output with one property connected. The first property linked to the input is from the device identified by the ID 10, has a property descriptor of zero, which means that has an 8_BIT property type, and has 1 as property configuration which means the property works in action mode. The other input and the output follow the same analogy. Finally, like mentioned before, the file ends with the letter e.

4.10 Configuration

Each block will also have several configurations that are set by the user that creates it but that can be changed by the user that uses it. For example, a block that is used to turn on lights based on movement may have a default timer to turn the lights if the sensors do not sense any movement for more than one minute, being the user free to change it for a desired value. These changes can be set in one input that receives a file with all the configurations. With this, every block should have at least one input that belongs to its configuration file.

When a block is instantiated, the inputs and outputs are final and cannot be changed. On the other hand, the configuration file can be freely edited by the user every time it is necessary.

4.11 Solution

All the points previously explained are powerful and allow the user to have a very good control of the system. The downside is that the way that all the system is defined right now makes it impossible to be used by a common user, even being very difficult to use by users comfortable with programming language and with the system itself.

The solution consists in an interface that would allow users to create and manage different types of blocks in their homes in a very accessible way. A simple interface with simple steps that encapsulates all the complexity of creating and managing a DAB to allow any user to enjoy the system.

The reasons behind choosing an interface web are due to its practical usage, not being exclusively connected to one operating system, allowing several users at the same time and updating in an easy and fast way. This solution will be divided in 3 parts: a server, a database and an interface. Defining and creating DAB is not easy and because of that, the interface should guide the user through the full process, avoiding errors and performing validations.

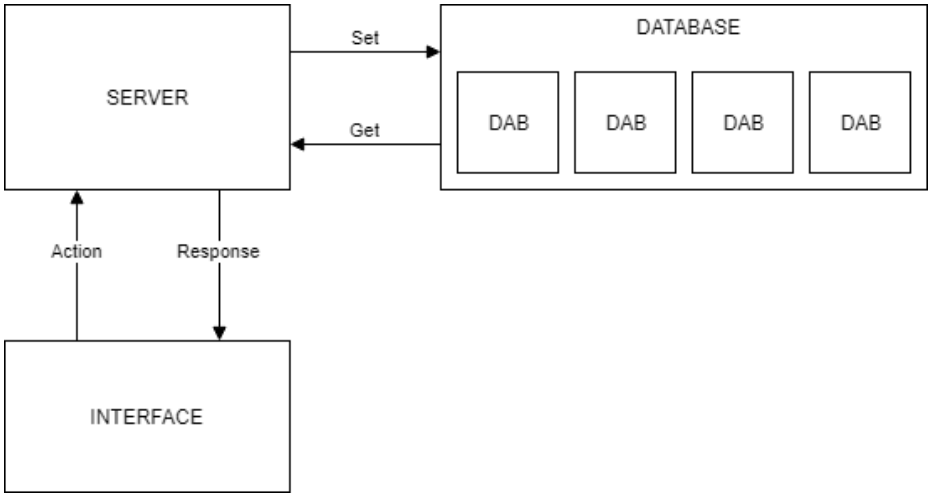

Fig. 11: Simple architecture for proposed solution

The technologies used were Ruby on Rails for the server, Vue.Js for the interface and PostgreSQL for the database.

Ruby on Rails is a framework designed to build web applications using Ruby, an object-oriented programming language. It's an MVC which translates to Model, Views and

Controllers. The model is where the data is stored and can be retrieved. In this project we have several Models such as *DomoBusAutomationBlock*, *Input*, *Output*, etc. The view is where is displayed the information regarding a Model. It is the interface that the user sees in a website. The controller is where the actions that are seen or received from the views are manipulated before calling the model to save it, update it or any other action. In this project the server side will work only as an API and will only be used models and controllers since the views will be made using a user interface builder named Vue.Js.

Vue.Js, unlike React that is a library, is a JavaScript framework. Each view is divided into three tags: <*template*>, <*script*> and <*style*> so, unlike standard html, in Vue styles and JavaScript functions are not centralized but rather divided in each file (Fig. 12). The <*template*> tag is where all the html code is written. The <*script*> tag is where the functions that will be triggered and the variables that will be used in the html are defined. The <*style*> tag is where the CSS is declared.

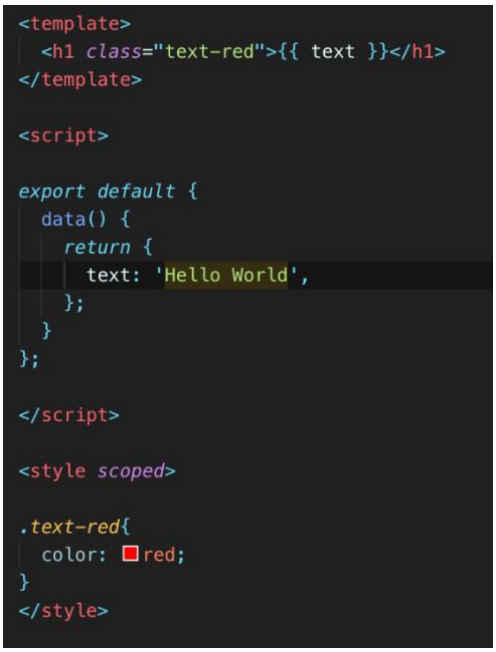

Fig. 12: Simple Vue.Js component.

5 Implementation

In this chapter it will be described all the parts of the application that was implemented to allow users to create DABs in a very simple way, and to use them to connect to different devices present in their homes.

5.1 Architecture

The Domobus system is composed by two components. One web application made with Vue.Js and an API made with Ruby on Rails. The application sends requests to the API that, after certain validations that will be explained later, returns a response to the web.

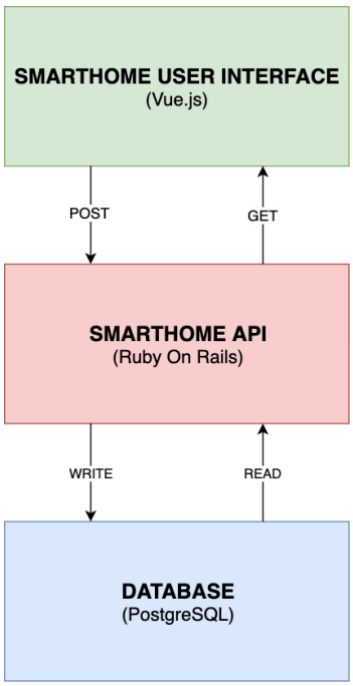

Fig. 13: System Architecture.

5.2 Users

The system has three types of users that we will call User A, User B and Admin. User A is the “normal user” that doesn’t have knowledge of how the system works. It is then very important that the user interface and experience are as good as possible so the user can add a house to the system and use DABs present in the server to create their automated actions without much difficulty.

User B is the type of user that has knowledge of how the system and the DABs work and how to program and create them. User B can also add houses but besides using DABs in the server, the User B can create and add DABs to the server himself.

The Admin is the superuser of the system and is the user that decides if a user is the type A or B. The Admin can give and revoke permissions for any user as well revoke any DAB that, in his opinion, doesn't seem correct or useful.

5.3 Authentication

The system is used by only two types of users right now and each of them must create an account in order to be able to access and use the application. The user can create an account in the “Sign In” page by inserting his name, email and a password. When the user clicks “Sign In”, the user data is sent to the API where is processed. There the user is created only, and only if his email is unique. If it is not, then the user won’t be able to create an account and that information will be presented to him in the interface in a form of a toast error. A toast message works like an alert box but has some styles associated and can be timed to disappear after a few seconds. If the user successfully creates an account, the user information is stored in the database, with the digested password and the login process will begin right after, in order to sign up the user into the application. To manage the login session, Json Web Token is used (JWT).

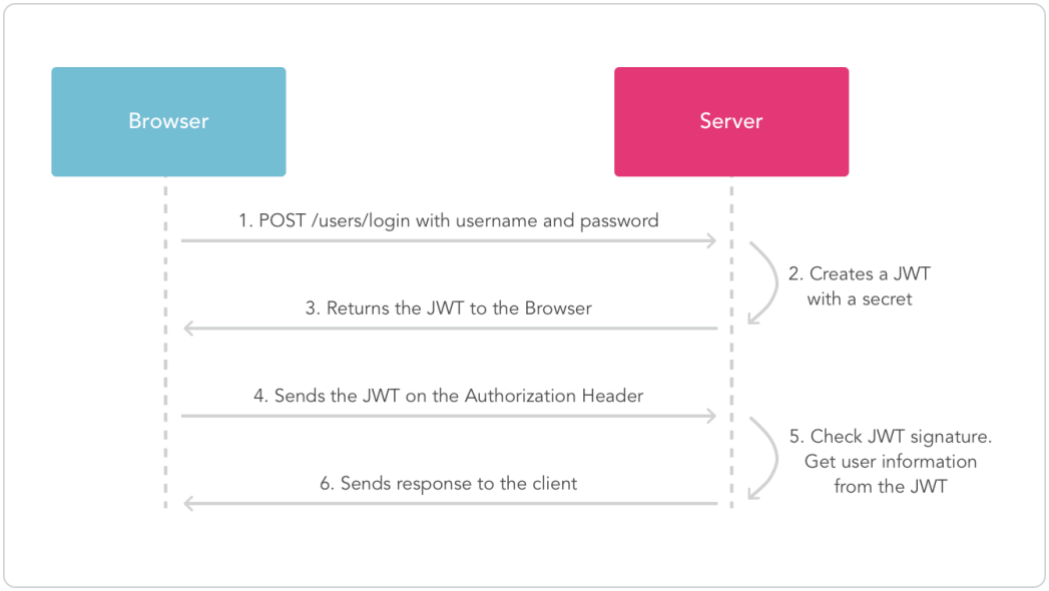

Fig. 14: JWT diagram. [14]

When logging in, a unique token is generated using a secret and assigned to the user with a 24-hour validity. The validity of the token can be changed for the desired value. Then, that token is sent to the browser where it is stored and added to the authorization header every time a request is sent to the API. Before every action, the API calls a *authorize_request* method that verifies the validity of the token and retrieves the user's information from it. If the user's information is correct and correspondents to the current user, then the request goes through. If not, the API returns a 401 error (unauthorised) to the browser and the user is logged out. This method protects the system against any attacks with the objective of forging the token.

```ruby
def authorize_request
  header = request.headers['Authorization']
  header = header.split('"').last if header
  begin
    @decoded = JsonWebToken.decode(header)
    @current_user = User.find(@decoded[:user_id])
  rescue ActiveRecord::RecordNotFound => e
    render json: { errors: e.message }, status: :unauthorized
  rescue JWT::DecodeError => e
    render json: { errors: e.message }, status: :unauthorized
  end
end
```

Fig. 15: authorize_request method

5.4 XML File

A user can add is house using an XML file that must contain all the information required by the API to perform the action. If the information provided in the XML is not valid, a toast error is presented to the user in the interface. XML stands for E**X**tensible **M**arkup **L**anguage and it uses tags to define objects. Those objects can have child objects and so on [14]. Below it is presented an example of an XML file.

```xml
<House ID="1" Name="Jose's Home" Address="R. Latino Coelho 24, 1050-132 Lisboa" Phone="219321243">
  <FloorList>
    <Floor ID="1" Name="Ground floor" HeightOrder="0" />
  </FloorList>
  <DivisionList>
    <Division ID="1" Name="Kitchen" RefFloor="1" />
    <Division ID="2" Name="Living room" RefFloor="1" />
  </DivisionList>
</House>
```

Fig. 16: XML example file

5.5 Adding a house

An authenticated user is presented with a navbar that is always present at the top of the page. User A only has present one tab in that navbar: Home, that is the principal page and where the user is present with a list of their registered houses and from where the connections between existent DABs and their houses are made. User B, with permission to create DABs, besides the Home tab, also has a tab Automation Blocks designed for the purpose. Both users also have, in the right corner of the navbar a, Log Out tab where the user can finish their session. Logging out, the user token will be cleared making him lose access to the system and being consequently redirected to the homepage.

In the Home page (Fig. 17) is present a list of the current houses registered in the system and at the bottom there is a button and a file submit area that allows the user to add a new house. If a user desires to add a house, a file must be chosen from the submit file area. As mentioned before, the file must have the XML format meaning the user isn't allowed to submit a different type of file because a validation is made directly on the file input. This prevents the user to mistakenly select another type of file.

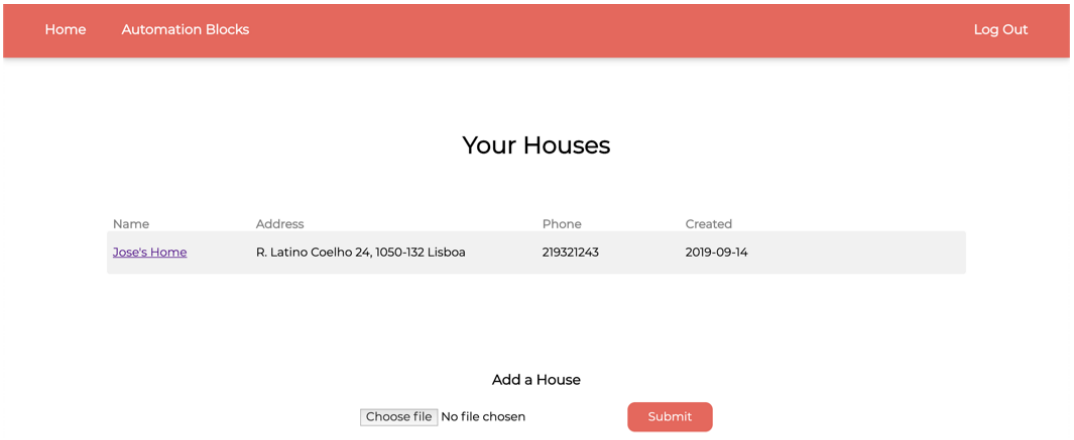

Fig. 17: Home page of the system.

After a file is chosen, the user must click in the "Submit" button. The click will trigger a method that will send to the API a json object with a *formData* that contains the file. In the API side, is used *rexml*, a toolkit for *Ruby on Rails* to parse XML and the file is processed [16]. The

way this is done is by using a top-down approach. The file is iterated and when an XML object is found, its type and tags are detected and the object, that is saved in the database accordingly to what was found before, proceeds to check its children. In the Fig. 18 is possible to see an example of the iteration of an XML file of the Fig.16. *Xmldoc* is the variable that contains the root variable of the document itself. Notice that when iterating over the Floor or Division elements it is not used the root variable but only the variable that contains the child of the House tag, in this case the variable 'e'. This is because Floor and Division are both children of House.

Because all the database action are SQL statements, they are performed inside a transaction that only succeeds if all of them succeed as well. If one fails an exception is raised, the transaction is cancelled, and a rollback is performed to reset the database to the state it was before the execution. If the transaction is successful, the API returns to the browser the updated list of houses that belongs to the user so that the list on the Home page is up to date.

```
xmldoc.elements.each("DomoBusSystem/House") do |e|
  h = House.create(name: e.attributes["Name"], address: e.attributes["Address"], phone: e.attributes["Phone"], user_id: @current_user.id)
  h.save

  e.elements.each("FloorList/Floor") do |s|
    f = Floor.create(name: s.attributes["Name"], height_order: s.attributes["HeightOrder"], house_id: h.id)
    f.save
    floors.push([s.attributes["ID"], f.id])
  end

  e.elements.each("DivisionList/Division") do |s|
    di = Division.create(name: s.attributes["Name"], floor_id: floors.select{|n| n[0]==s.attributes["RefFloor"]}[0][1], house_id: h.id)
    di.save
    divisions.push([s.attributes["ID"], di.id])
  end
end
```

Fig. 18: Small example of the processing of a house XML.

5.6 Creating a Domobus Automation Block

Like mentioned before, users of type B can create DAB's. This can be done by accessing Automation Blocks through the top bar and then clicking on the button "create a block". The Domobus creation workflow has 5 steps, DomoBus Automation Block, Inputs, Outputs, Timers and States, that will be explained below. It will be used an example of an automated light using a movement sensor. In the Fig. 19 is possible to see the state machine diagram of the example.

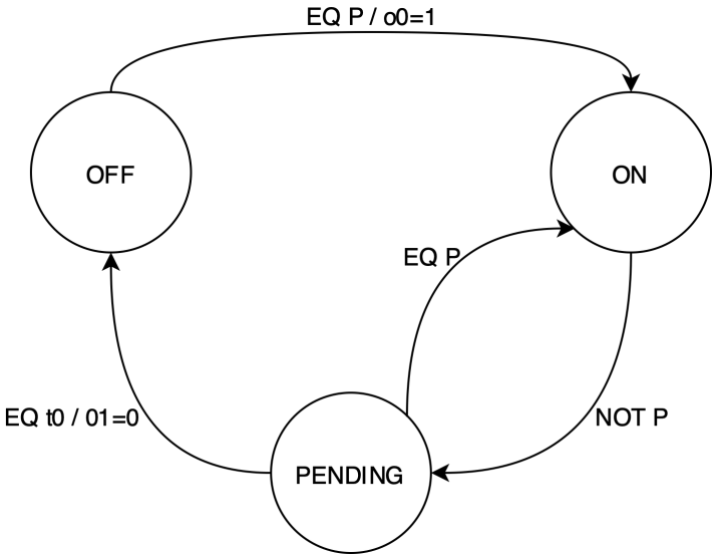

Fig. 19: State machine diagram.

The DomoBus Automation Block step is where the user chooses a name and a small description to the block. It must be self-explanatory and straight forward so that the users that decide to use the DAB don't have any doubt of what it can and should do.

In the Inputs step the user adds all the inputs that the DAB accepts. The inputs will be connected with the properties of a device (sensors for example) to trigger an action. For each input, the user chooses a name, a property type, a mode, the operator, the value, the condition and a default value in case the block is used without this input.

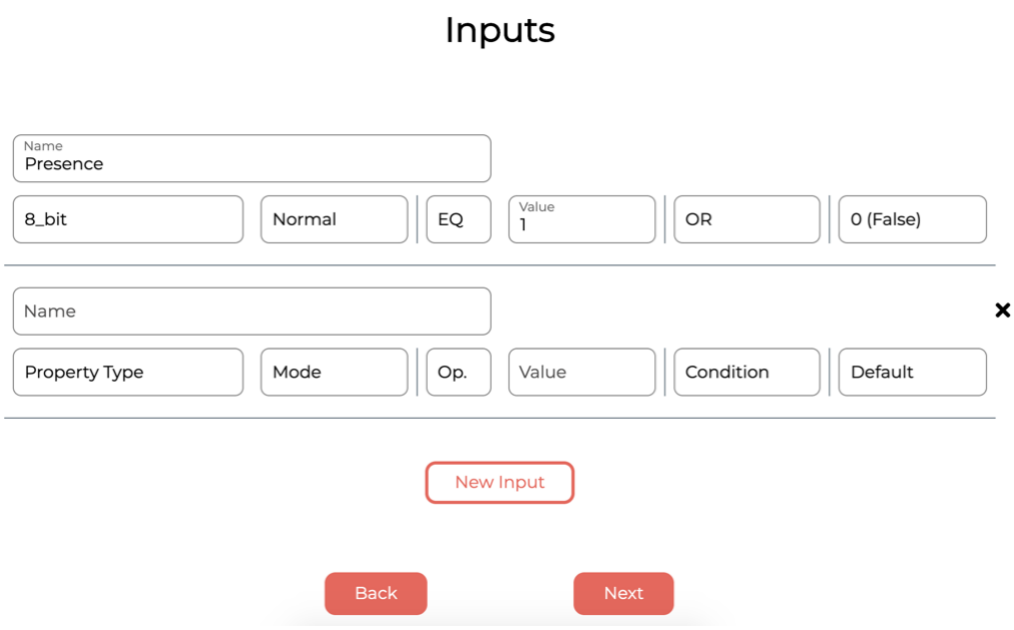

Fig. 20: Second step of the create algorithm - Inputs

In Fig. 20 is possible to see an input with the name Presence and a property type of 8_bit that works in the normal mode, which value must equal (EQ) 1 in order to be activated. In case the user has more inputs of this type, they will be aggregated using the OR condition and the default value in case the block is used without this input is 0. It is possible to add more inputs by clicking on the "New Input" button and delete them by clicking on the 'X' present at the top right of each input. This shows that system is flexible and scalable. The shown design was also chosen to allow the adding of new objects to be user friendly.

In the next step, the outputs are defined by choosing a name and a property type to each one. The outputs will connect to a property of a device such as a lamp, an AC, etc. The actions triggered in the inputs may or may not affect the outputs. Just like the inputs, it is possible to add multiple outputs using the button "New Output".

Setting the timers is the next step. This is the easiest step because the user only needs to add a timer and define its time in seconds. The user can add as many timers as he desires.

The final step is the hardest one and is where the user must use his programming skills and his knowledge of the system. In this step is where the user creates the states, defines the transition(s) required to change from a state to another and the action(s) triggered when changing state. A state must have a name and one or more transitions that are divided in

conditions and actions. The conditions together make an expression. When a condition is met, the respective actions are triggered. A state represents all the ways an output can be represented. In the case of a light, it can have the state “On”, “Off” and a “Waiting” state if the block allows a movement sensor as input. In Fig. 21 it is possible to see the first state of a very simple automated light block.

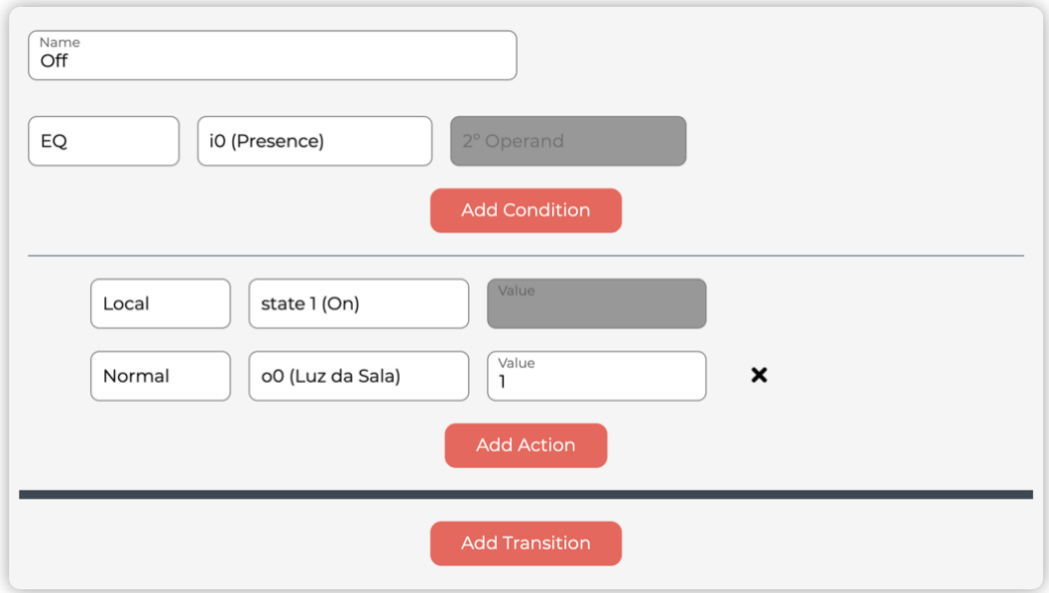

Fig. 21: First State - “Off”

In the first state, there is one transition with one condition and two actions. The name of the state is “Off”, and it’s the first step. When the sensor, that is represented as i0 (abbreviation for input 0), is equal, which means that it's true, the two actions are triggered. The first action is local and changes from the state 0 “Off” to state 1 “On” and the second action, which is normal, sets the output to one which means the light will be turned on and stay that way. In Fig 22 it is possible to see the second state, “On.”

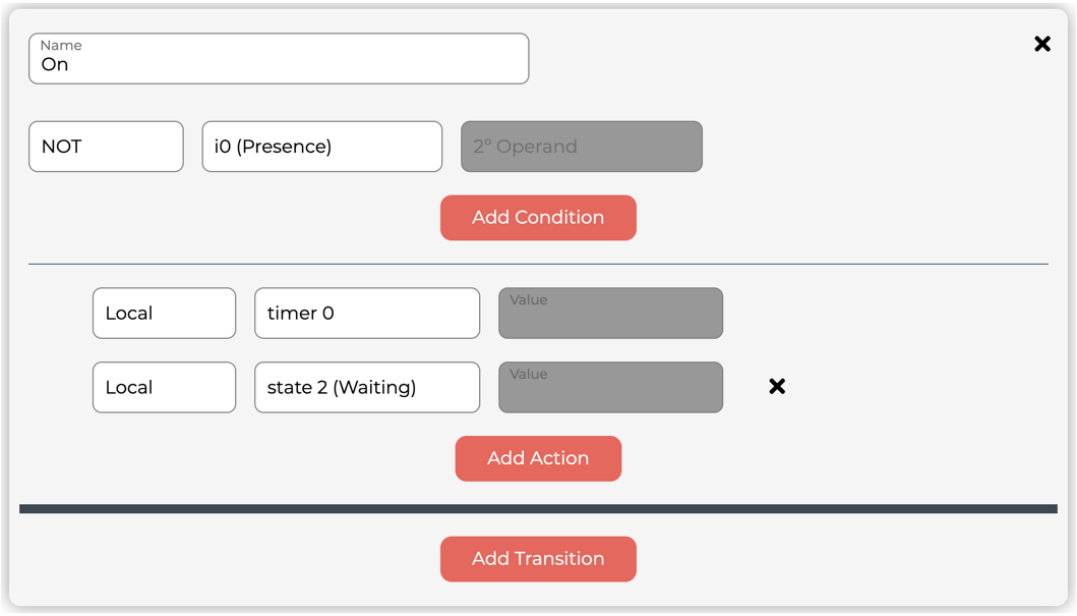

Fig. 22: Second State - “On”

In the second state, the light is on, which is what gives the name to the state. In this state we have again one transition with one condition and two actions. When the input 0, the sensor, does not detect any movement the timer will start and the state will change from “On” to “Waiting”, the third state.

Fig. 23: Third state "Waiting"

In the “Waiting” state the light is still on, but the timer is running because the sensor didn’t detect any movement. At this point there can be two outcomes and that is the reason why this state has 2 transitions. In the first transition, if the timer equals, which means it is true and it has expired, then it will be cancelled, the state will be changed from “Waiting” to “Off” and the light will be turned off by changing the output value to 0. In the second transition, however, if the sensor detects any movement while the timer is still running, the timer will be cancelled, and the state will change again from “Waiting” to “On”.

5.7 Write the block to a file

After the user finishes defining all states, a request is sent to the API when a trigger is activated by clicking in the finish button. The request contains a JSON object with all the data created during the DAB creation workflow and is organized hierarchically. In Fig. 24 is possible to see an example of a state object organized.

Fig. 24: State object organized hierarchically.

When the JSON object arrives at the API side, it is parsed, and all the valuable information is used to create all the components that together creates a DAB. That information is saved in a database in form of an Input, Output, Timer, Transition, Condition, Action and State (Fig. 25). Then, the DAB is written to a file following the rules defined by Professor Renato Nunes in the DAB documentation and that were explained in section 4.

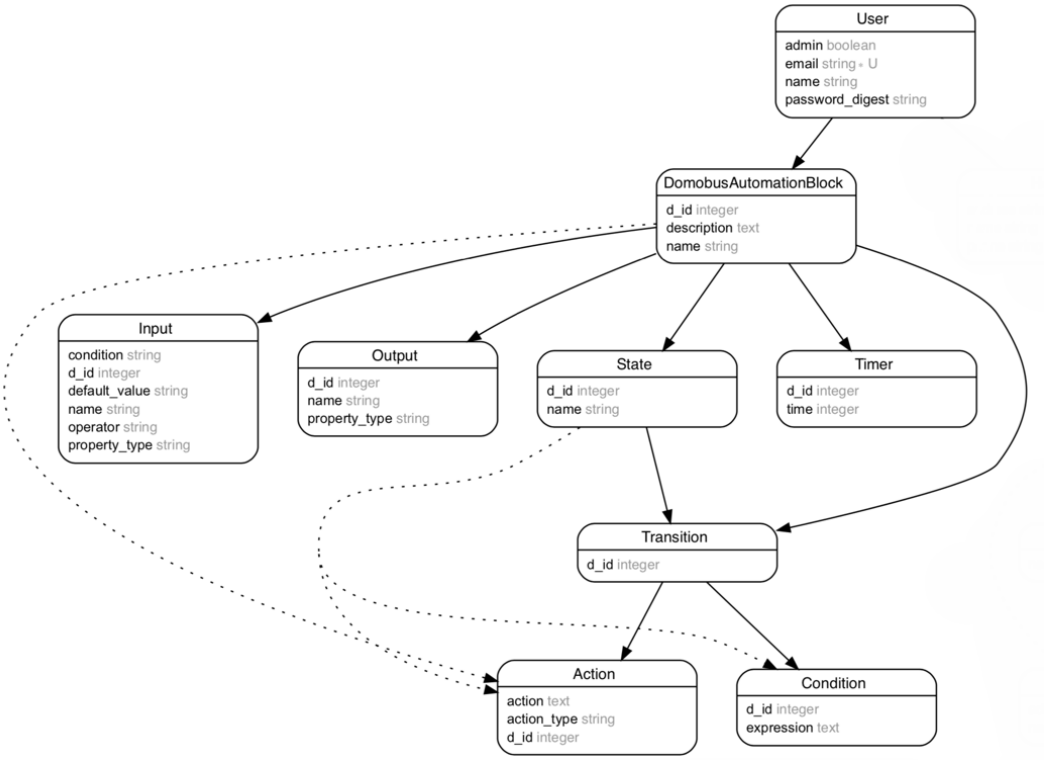

Fig. 25: DomoBus Automation Block Entity-Relationship Diagram.

5.8 Editing a Domobus Automation Block

The ability of editing a DAB is actually one of the most important features of the system, for several reasons. It allows the possibility of correcting any mistakes that its creator committed without having to create a new DAB from scratch. It also allows the possibility of extending it and improve it at any given time so the DAB can stay updated with the requests desired by the users.

When designing the process of editing a DAB there were some things that were important to reflect. The usability of the method was important so that a user could easily edit the block without too many constraints. It was important that the information of the block was shown so that the user could see its current state before making the desired changes. For all this reasons, and because the create method is very clear and practical, it was decided to make the user capable of editing a DAB by reusing the steps of creation, being this the best option because,

not only it has a good usability, but also because it avoids the necessity of learning new workflows for very similar methods.

To edit a DAB, it was then added a pencil icon to every block on the list. This icon is a common symbol to edition of a certain object.

Fig. 26: Pencil symbol to edit a block and the button to create a block.

There were some things that needed to be changed on the create algorithm so that the integration of the edit functionality could be done successfully. Because the create and edit respective buttons will send the user to the same view, it was necessary to give the system some kind of information if the user wanted to create or edit a block. When the user clicks in the pencil, the ID of the block he wants to edit is saved in a local storage, as well in the vuex [23]. The local storage is a web storage present in the browser that gives websites the possibility to save information without an expiration date using JavaScript. The vuex is a library of the Vue.Js framework and works as a centralized storage where it is possible to store data as state between different views.

As seen in the Fig. 12, the data object inside the <*script*> tag is where the state variables are defined but they are only available to the component itself, so there's when the vuex comes in. Vuex allows the user to declare some kind of state variables in a store that are available to all components. The way it works is pretty straight forward: a component X declares a state variable A that later the user decides that will be needed in other component Y, so the state variable is removed from the component X and is declared in a store file. After declared, the variable needs a mutation, an action and a getter. A user calls an action, with or without

parameters, that then calls a mutation that executes the code and saves, or not, the information passed as argument in a certain variable. The getter is used to retrieve the information from the store.

Although this may seem confusing, with an example it all comes clear. It will be used the example when editing a DAB. When clicking in the edit button, immediately before, the action "EDIT_DAB_SAVE" is called with DAB ID as argument because this is the information that will be saved. Inside the store, the action called triggers a mutation "editDABSave". Inside this mutation the state variable "editDAB" is set to true and the ID of the DAB, that was passed as parameter, is saved in the local storage. The process of saving information to the store is now complete. Now onto the part where information is retrieved from the store. After clicking in the edit button, the new view starts but before it renders it is possible to call the *created ()* method. In this method is where the code, that needs to run before the view is rendered, must execute. API calls, for example, are made in this method. In order to fill the view with the information of the DAB an API call is required. Before the call is made the ID of the block is fetched from where it was saved earlier: the store. Thanks to the getter defined in the store file, the call "this.$store.state.editDAB" is needed to have access to the "editDAB" and that's it! This is how data is sent, saved and retrieved from the store.

Now that the DAB ID is known, the API call is made to get the its information. If the ID, for some reason is not recognized and the server does not find any block with that ID, an error message is provided to the user. When the result arrives, in JSON format, the message is processed to fill the view and display the current status of the DAB. When the rendering process is finished the user can start modifying the block by deleting, changing or adding information to it.

In Rails, when a model object is updated, in this case the DAB, the server knows what fields are different and automatically updates those fields when the update method is called on them. But the problem this update presents is that, besides updating the information of the block, it can also update the relationship models that are connected to it (Input, Output, State, etc.). In order for the API to know what models where modified in front-end, some changes needed to be made on the *create/edit* view. To have a better understanding of what the major changes were made, an example will be explained regarding the Input model. As seen in Fig. 21, the information required to create an input is a name, a property type, a mode, the operator,

the value, the condition and a default value. In background what the input object saves after processing is what is described on Fig 27.

Fig. 27: Input object processed in the front-end.

The first problem with this solution is that there is no information that can make the API identify a specific input. This approach faces no challenges when we are talking about the create method because an object model has an identity, not when it is instantiated, but only after its creation. When a block is created from scratch, all the objects models are being created so there is no need of identity information. But when it is updated this change because the server needs to know what inputs were modified. To solve this problem, it was added an ID attribute to each input. Besides identifying the input, this attribute also fixes the problem of distinguishing the new inputs from the existing ones. On the server side, when iterating over the inputs array, a verification is made with the ID of each object. If the ID attribute is present and is not empty, then the server knows that it's an input that already existed, so it checks for updates and execute them. If not, it creates a new input with the corresponding information.

There was one last challenge regarding the update: how does the server know when an input is removed? To solve this problem, it was created a *syncData* method on the server side that receives two arguments: the name of the model and the array corresponding to it that was sent in the update request from the front-end. In this example, the model is "Input" and the array is inputs. The method iterates over the array and checks if every ID in the database is in

the array. If one of them is not in the array of inputs, then the servers knows that it was deleted on the front-end and deletes the record from the database. The objects without ID are ignored.

Although it was used the Input as example, the algorithm is valid to every model.

Some problems were faced during the implementation, most of them because the lack of organization of the edit algorithm. The "create" algorithm was already made so it was hard to find a way of implementing everything together without breaking everything.

5.9 Connecting house's devices to a Domobus Automation Block

One of the last steps of this workflow is allowing the user to use a DAB made by another user that is present in the database to connect to a device in house. It will be explained how the connections are made, the steps taken to prevent the user to mistakenly link properties and how the file with the connections is created and generated.

Designing the functionality of this feature was one of the most challenging steps. Linking objects in a user interface is not easy to execute and hardly usable. In a first approach, an algorithm of drag and drop was made. This method consisted in dividing the view between a device and a block. The block and device would have boxes that would represent the input and output for the block and the properties for the device. The user would drag the properties from the device to the respective box on the DAB. Later on, this solution was cut off due to complications related to scalability and flexibility of the method. Another reason was the usability of the system. Since the system allows the block to be connected with properties from different devices from different divisions, there is a huge amount of information needed to be provided to the user when constructing the connection. It was almost impossible to have a clean and usable algorithm of drag and drop without giving up on some part of the crucial information. The solution chosen, and present in the system, is based on an html *<select>* element. This solution faces all the previous problems in a very clean way and will be explained below.

The information needed to be shown to the user related to the device is the name, its division, its properties and the type of the properties. To connect the block, the user needs access to a list of public DABs and, when one is selected, the information about its inputs, outputs and properties. As mentioned earlier, there is a lot of important information that needs

to be presented to the user in the most usable way. Next it will be explained how that was done and the connections made.

The houses added by a user are present in the house list on the Home page (Fig. 17). When the user clicks on any of them, a view with the information of the selected house appears (Fig.28). This is the first step of the workflow that is being explained and first piece of information needed to present to the user. It is possible to see that every division is shown on the respective floor. The user can enter a division by clicking in it.

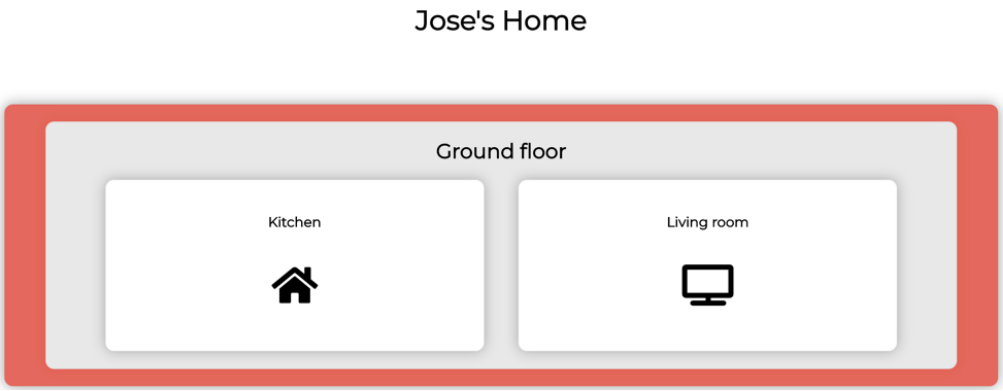

Fig. 28: View of a house.

Like mentioned previously, when making the connection between a DAB and a house, is important that the system has the flexibility that allows the user to switch between divisions without losing the properties already assigned to the block. In order to do this, inside the division's view, it was needed a way of changing between divisions without going back to the house view. This was accomplished by joining all the divisions, and respective functionalities, in one large object and set up an arrow in the top corner of the properties container to move between them. The division view (Fig. 29) is divided in a container on the left side that encapsulates the divisions with their devices and respective properties, and another container on the right side that contains the information about the DAB.

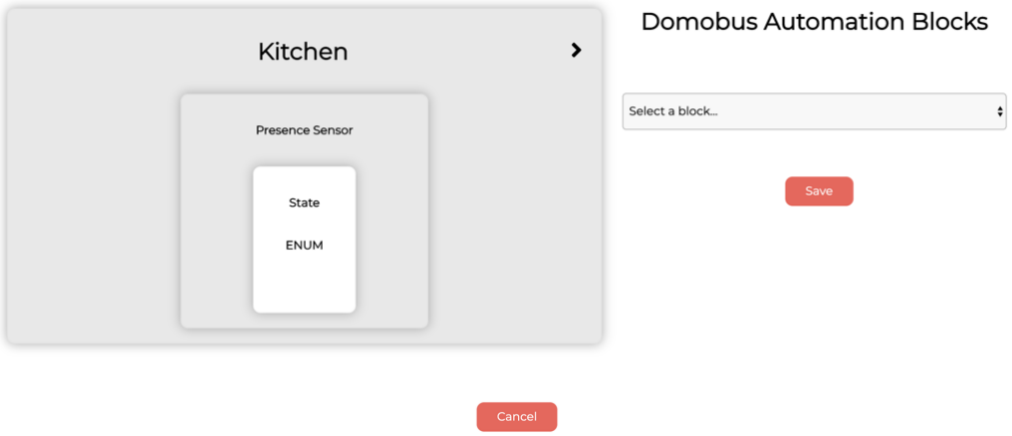

Fig. 29: View of a division.

From the <*select*> element present below the DAB title, in the left container, it is possible to choose between all the blocks that exist in the database and that were made by other users. When choosing a block, a container between the <*select*> element and the save button appears. From this container is where the user can actually make the connection between the properties that are inside each device present in a division and the inputs and outputs of a block. The user is presented with two different tabs, one aimed for inputs and the other for the outputs. When selecting one of the tabs, the inputs/outputs are listed in the form of a block and each one with a respective <*select*> element. Inside that element is listed every property that exists in every device present in every division. The element starts by listing only the properties presents in the selected division and adds the rest of the properties when the users changes between divisions.

As previously mentioned, an input can only be linked to properties with the same type and to prevent users from mistakenly choose a wrong one, the input only lists the properties that are allowed to connect to them which means they have the same type. To be more user-friendly, and to give the user some feedback about which property in the <*select*> list corresponds to the property in the division's devices, when passing the mouse over the property, the container of the correspondent property on the left side turns green. This allows the user to have a better perception of which device the selected property belongs to.

Other functionality that the algorithm must support is to make an input accept several properties from the same or different devices. This is a key feature of the DomoBus system and is what makes it so powerful and flexible. This was accomplished by making the <*select*> element accept several clicks and properties. The properties selected are then pushed to an array

that is displayed bellow the name of the input. If a user desires to disconnect a device's property from the input, all he needs to do is click on the arrow that appears on the top-right corner of the properties that are displayed in the array. In the Fig 30 is possible to see all the functionalities, mention above, at work.

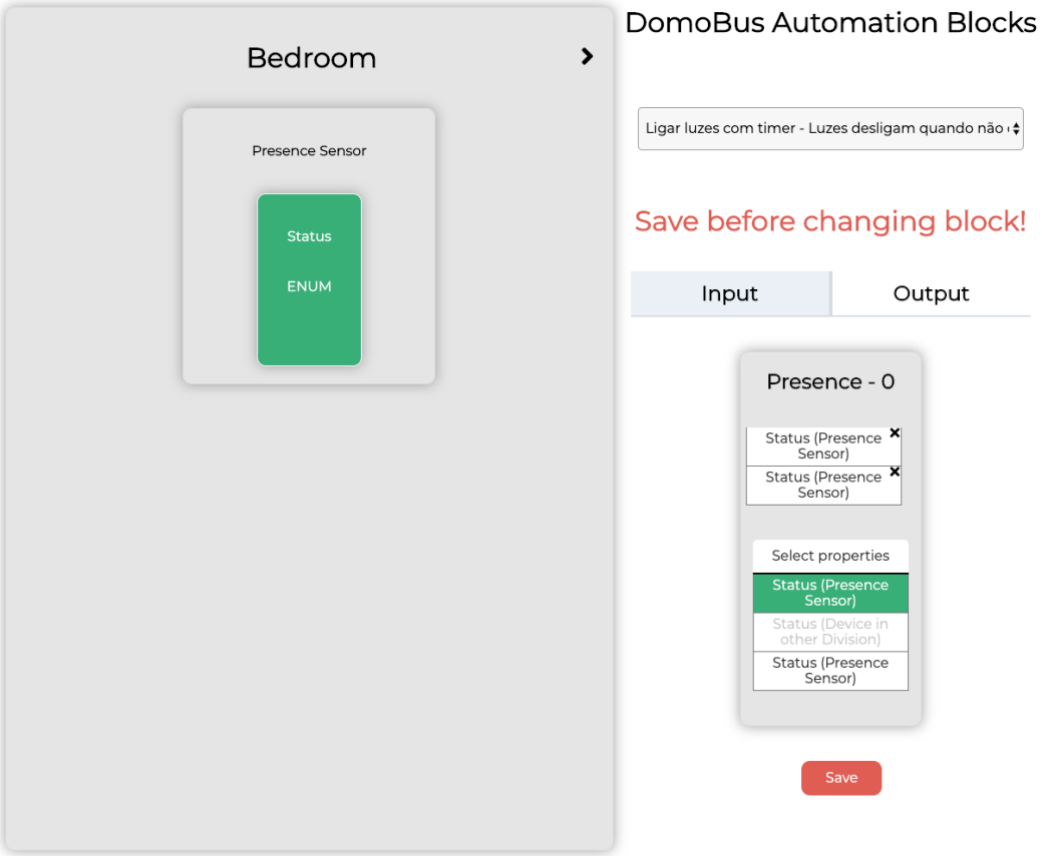

Fig. 30: Input Presence with two properties connected.

6 Evaluation

In order to test the interface and its usability, two different tests were conducted. In both tests there were steps to follow and a scale where the user should identify the difficulty when executing the step. The scale was from 1, meaning the task is not usable at all, to 5, the task has a very good usability. It was also registered the number of wrong clicks the user took before making a correct one. It will be detailed one example for each type of user.

In the test designed for the user A, there were thirteen questions. From all the thirteen, one question received a 3 grade, two received a 4 grade and eight received a 5 grade. The lower grade was awarded to the task of adding a house with the justification that he did not had previous knowledge of XML and had a hard time understanding the concept of it. The user made one wrong click in the tasks 'Verify the Inputs and Outputs of the DAB', 'Connect a Device to an Input' and 'Connect a Device to an Output'. It was asked the user to make a final comment about the system. The user gave a grade of 8. The user liked the design and general functionality of the system but also commented that the interface must have more labels that explain what each component does.

In the test designed for the user B, that has permissions to make blocks, there were seventeen questions. From all the seventeen, one received a 2 grade, four received a 3 grade, five received a 4 grade and seven a 5 grade. The lower grade was awarded to the step of adding an input to the DAB, the second step of creating a block. The reasons behind it was due to two factors: the first contact with the creation workflow and the poor label for properties type. The user felt an initial confusion when started the process of creating a block that was later dissipated when he got more used to the workflow. The other cause was the fact that the properties types are defined in 8_BIT, 16_BIT, 32_BIT and DOMOBUS_ARRAY which the user thought that, even for the users that have some knowledge of the state, are not user friendly at all. For the final comment the user gave a grade of 8. He said that the system is well design and very usable in general and that it facilitated a lot the work for DAB creators. He also said that some actions could have labels with small text explaining their purpose.

7 Conclusion

Home automation systems are growing in number and features. They must allow the user to make anything they want in a fast and flexible way. In most cases when a user wants to manage a device in his house, he just needs to follow simple steps depending on the system. But what if he wants to more than that? What if he wants to control what he can do? What if he has more than one device and he wants all of them to perform the same task? Does he need to create logic for each one independently although they will all perform equals tasks?

The Smart Homes web application allows the user to have control of their system by using DAB. The system encapsulates most of the complexity behind the creation and management of the blocks allowing almost every user to be able to use it in a simple way. The flexibility that the DAB presents offers the possibility of creating logic that can be reused for several devices as long they have all the same property type.

The results with experimental users were positive and show that the system drastically eases the understanding and use of the DAB. The common consensus is that system has a good design and functionality in general. It also allows the common user to add and manage blocks to his home in an easy way although some more labels and explanations could be added for better understanding.

Even though the application allows the user to perform the core functions of the system, there is still some further work that can be done in the future in order to optimize it, such as; allowing the user to manage his houses by manipulating them directly in the application, or even deleting them, preventing the user from manipulating the XML file and having to update it every time he wants a new division or device; allowing the user to check which block he is using and where he is using it; creating even better ways to manipulate the complexity of the DABs by, for example, designing a more user-friendly way of displaying the different properties type to the user.

DomoBus gives power to the user to create their own automated tasks using DABs and they can do that using the Smart Home application.

Technology is becoming an integral part of our life and the next step is letting it enter our homes. The future holds many secrets, but one thing is for sure, Smart Homes is the smart way to live.

8 References

[1] Lumpkins, W.: Home Automation: Insteon (X10 Meets Powerline). IEEE Consumer Electronics Magazine, 140-144 (October 2015).

[2] What is Insteon?, http://www.rfwireless-world.com/Terminology/Insteon-basics.html, last accessed 2018/12/8.

[3] Carbonette, B.: Connecting Arduino to IFTTT. In: Hackster.io (April 2017).

[4] Insteon page, https://www.insteon.com/, last accessed 2019/1/3.

[5] Amazon Alexa page, https://developer.amazon.com/alexa/connected-devices, last accessed 2019/1/4.

[6] ZigBee explanation, https://internetofthingsagenda.techtarget.com/definition/ZigBee, last accessed 2018/12/27.

[7] Smart Home Skill API page, https://developer.amazon.com/docs/smarthome/understand-the-smart-home-skill-api.html, last accessed 2019/1/5.

[8] Amazon Echo auto page, https://www.amazon.com/Introducing-Echo-Auto-first-your/dp/B0753K4CWG, last accessed 2019/1/6.

[9] Nunes, R.: DomoBus Automation Blocks, Programming the DomoBus System, Technical Report, Instituto Superior Técnico, University of Lisbon, 2018/09/16, last accessed 2019/10/18.

[10] Nunes, R.: The DomoBus System Specification Language, Programming the DomoBus System, Technical Report, Instituto Superior Técnico, University of Lisbon, 2018/09/16, last accessed 2019/10/18.

[11] Nunes, R.: Especificação XML de um Sistema DomoBus, Technical Report, Instituto Superior Técnico, University of Lisbon, last accessed 2019/10/18.

[12] Welcome Home, https://ifttt.com/applets/kCYK8bu3-welcome-home?term=welcome%20home, last accessed 2019/1/7.

[13] Nunes, R.: AI_03_Domotics_DomoBus, Instituto Superior Técnico, University of Lisbon, last accessed 2019/1/5.

[14] JWT Diagram, https://cdn.auth0.com/content/jwt/jwt-diagram.png, last accessed 2019/1/5.

[15] XML.: https://fileinfo.com/extension/xml, last accessed 2019/1/5.

[16] Evans, J.: REXML: library for Ruby On Rails, https://github.com/ruby/rexml, last accessed 2019/1/5.
[17] IFTTT.: API, https://platform.ifttt.com/docs/api_reference, last accessed 2019/29/9.
[18] SmartLabs Technology.: INSTEON_Developers_Guide, http://cache.insteon.com/pdf/INSTEON_Developers_Guide_20070816a.pdf (August 2007), last accessed 2019/29/9.
[19] Stringify is on IFTTT, https://ifttt.com/discover/stringify-is-on-ifttt, last accessed 2019/10/5.
[20] Build Skills with the Alexa Skills Kit, https://developer.amazon.com/en-US/alexa/alexa-skills-kit/resources/training-resources/alexa-skill-python-tutorial, last accessed 2019/10/6.
[21] Understand Custom Skills, https://developer.amazon.com/docs/ask-overviews/build-skills-with-the-alexa-skills-kit.html, last accessed 2019/10/6.
[22] Czrabode.: Creating Complex Automation Using IFTTT and Stringify, https://medium.com/@czrabode/creating-complex-automation-using-ifttt-and-stringify-72ec01df60e0 (November 2017), last accessed 2019/10/7.
[23] What is Vuex?, https://vuex.vuejs.org, last accessed 2019/10/8.