



TÉCNICO
LISBOA



Home Smart Home

Ambient Intelligence

Instituto Superior Técnico

Master's degree in Computer Science and Engineering

Group 04

Isabel Soares (89466) & Rodrigo Sousa (89535)

May 2021

Abstract

With the recent developments of technology, more and more frameworks and systems are produced with the objective of automating households and their devices. Although most of them focus on the remote and commodity approach rather than supporting a house schema, the heterogeneity of devices and scalability of the system.

Home Smart Home expects to be a home automation and control system that considers all the aspects requested in this kind of system, namely the commodity, the scalability and even the automation process. Regarding this last point, our system not only will support the creation of user preferences, shortcuts that will allow the user to change various devices at once but will also allow their scheduling and will automatically try to identify these routines.

We successfully developed this system from the perspectives that we had initially set out do, an intuitive system that allows for home control and automation using a real database and real dynamic information. The system is implemented in a way that is fully scalable and that it would be able to support further improvements and developments that were not developed due to time constraints.

Keywords

- Human-centered computing ~ Human computer interaction (HCI) ~ Interactive systems and tools ~ User interface management systems
- Computer systems organization ~ Architectures ~ Distributed architectures ~ Client-server architectures
- Computer systems organization ~ Real-time systems ~ Real-time system architecture

Table of Contents

1. Introduction	1
2. Related Work.....	2
2.1. General Structure	2
2.2. Database & Authentication	2
2.3. Backend	3
2.4. Frontend.....	3
3. Description of the Solution	4
3.1. The Authentication.....	4
3.2. The Database.....	5
3.3. The Backend	6
3.4. The Frontend	7
3.4.1. The Utilities: Endpoint API.....	7
3.4.2. The Utilities: Store	7
3.4.3. The Classes	8
3.4.4. The Components	8
4. Details of the Solution.....	9
4.1. Login	9
4.2. Framework of the Application	9
4.3. Division Selection	9
4.4. Devices	10
4.4.1. Device properties and its values	11
4.4.2. Graphic Visualization.....	11
4.5. Scheduled Events	12
4.6. Favorites.....	13
4.7. Settings.....	14
5. Evaluation.....	15
6. Conclusion	16
7. References.....	17

1. Introduction

Nowadays the process of automating simple and daily tasks takes a big focus on the development of new technologies. The big focus is due to the want and need for the removal of painfully and repetitive tasks from the everyday person, basically what we assume that a machine would be able to do as well as us every day we would like not to do it. Some quick examples of this kinds of tasks include closing automatic blinds at a certain hour, suspend or place devices on standby mode after a certain hour, prepare a coffee at predetermined time, among others.

In addition to the examples given before, where the focus is on scheduling this type of events and delegating them to a specific system, there is also a wish for commodity, the user might want to change the state of certain devices without having to go to them, remotely controlling them. Such cases include turning on or off all the lights without having to go to each one individually, changing the television channel when we do not know of the remote, opening an automatic blind from bed, among others.

As an additional aspect that most users aspire these systems to have is the capability of recognizing patterns of users' routine and themselves scheduling these types of tasks. That means that more than allowing for the control of the environment they want this system to have characteristics that resemble pro-activeness and adaptivity. Just like we mentioned before, if we open the blinds every day at the same time, we would like this task to be automated and scheduled, but additionally we would want the system to automatically recognize this as a repeating task and schedule it.

Even though new technologies and frameworks, for instance Alexa, seem to be developed every day, they all lack in a simple aspect, compatibility between different devices, which can be a problem.

With the intention of solving that problem, we developed *Home Smart Home*, a framework system that allows the user for the creation of a model that describes the user's houses, floors and devices, as well as the devices states, their history and allowing for the scheduling of automated tasks on these devices. As an additional feature this system also recognizes simple daily patterns on the user's routine and develops schedules autonomously.

2. Related Work

During the development of the project a vast of technologies and frameworks were used in order to either facilitate the process of development or due to their already verified results and efficiency.

2.1. General Structure

The most crucial part of this project is supported by an already developed framework called *DomoBus* [1][2]. The framework defined in *DomoBus* already consider the necessities and crucial points that a system to manage a Smart House should have, for example the heterogeneity between devices and their properties, as well as considering the instinctively hierarchical nature in a system like this.

DomoBus, as it can be seen in Figure 1, specifies a system focused on a house in a hierarchical way, although the model is defined in an XML like syntax, the schema can be transformed and adapted maintaining their properties.

DomoBus, besides defining houses, floors, divisions, and devices in this hierarchy, also defines Device Types which give a schema of properties to a Device which can in turn be populated with values (denominated in our system by Property Values).

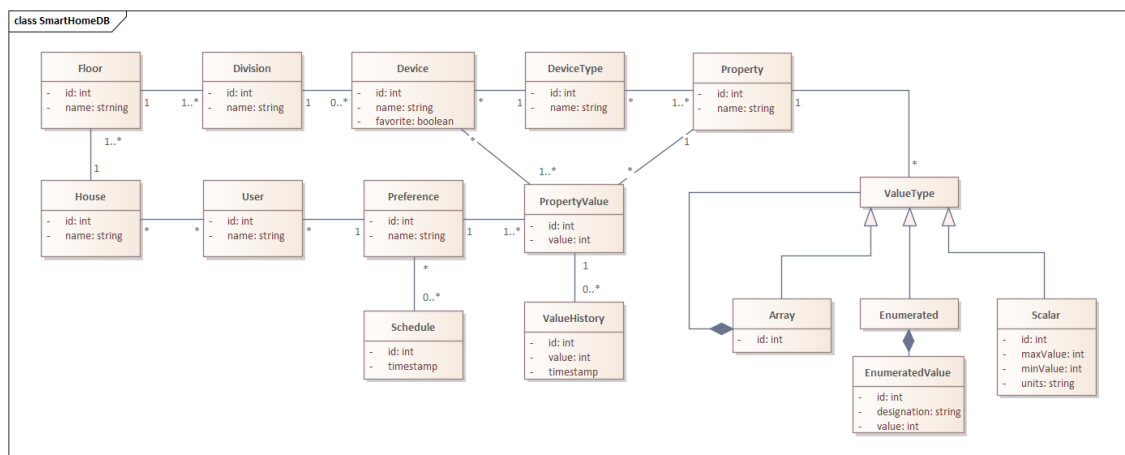


Figure 1 - Database Model.

On top of what is defined by *DomoBus* some changes were implemented. Classes such as *Preference* and *Schedule* were created to allow for the definition of user's routines which can then be triggered either manually or automatically at a specified time.

2.2. Database & Authentication

The project developed relies on *Firebase* [3] for its Database and Authentication service. *Firebase* is a Google's platform that englobes and allows for much more than we used it for, it allows for machine learning algorithms capable of interpreting text, photos, faces, etc. It also allows for services of chatbot messaging among other services. In summary, it englobes services that facilitate the creation and development of mobile and web applications.

Our **Database** relies on *Firebase's* Cloud Firestore, which is NoSQL a cloud database which is flexible, scalable and focused on the development of mobile and web applications. Being a NoSQL database means that it is not relational, instead of having every object represented through a row on a specified table, it has documents that are inserted in collections. The main

advantage of this kind database is that information does not need to be spread out across several tables and connections identified through foreign keys, in fact every piece of data can be nested inside other pieces of data, either directly or through references, this allowed for an easier update of database information in real time since when something changed the rest could be updated through its references.

The **Authentication** system provided by Firebase allows for much more than simply creating and logging users, which was what we used for the development of the project. It allows for various sign in methods (email, phone, Google, Facebook, Twitter, GitHub, etc.), specification of authorization rules, email verification, phone verification, among others.

2.3. Backend

The backend was implemented entirely on Typescript with the assistance of Node Package Manager (npm) [4] which facilitates the use, installment, and update of several packages on a single project as well as possible automation of development modes and tests.

Most packages used in the backend are used to support the development of the application, meaning that are not fundamental, for example *Nodemon* [5] (which allows for the automatic restart of the application when changes in development are detected), but there are some packages that are worth mentioning, for example: *Express* [6] which allows for the creation of an API; and Firebase which allows the connection to Firebase authentication and database.

2.4. Frontend

The frontend was also built using Node Package Manager (npm) but this time integrated with *React* [7], *Electron* [8] and *Redux* [9], as well as many other packages which simplify or aid the development of the frontend but that are not worth mentioning due to their small importance.

Electron allows for the almost effortless creation of desktop applications; in fact, it allows for the creation of multi-platform native applications.

React allows for straightforward approach when creating and designing the applications. Its syntax follows closely the syntax of HTML files but merged with the capabilities of Javascript, it is called *JSX* [10] (Javascript XML). In React, everything is encapsulated in components which hold their own state and even logic functions.

Redux comes as an addition to React, meaning that we used mainly React-Redux, which allows for definition of a global store that stores global logic and state across all the application, instead of component based. This was particularly useful to save the current user as a state that could be accessed globally throughout the application.

Most of the components used in the development of the application were either simple HTML components or components from *Material-UI* [11]. Material-UI has already a full set of components that can be graphically customized and for which logic can be defined. We decided to use these pre-made components since it would allow us to focus on the logic behind and on the application and less on making sure that the components were properly built.

The frontend also made use of packages which either improved the development process or allowed us to access information. We also used Nodemon, on the frontend, for the same reason as mentioned before. We used firebase to authenticate the users and Axios [12] to connect to our backend. Finally, we used a small package called Recharts [13] which permits the creation of charts with React components, which was used when displaying a graphical visualization of the change in value of device properties.

3. Description of the Solution

Our proposed solution can be divided in four main modules: The Authentication, the Database, the Backend and the Frontend.

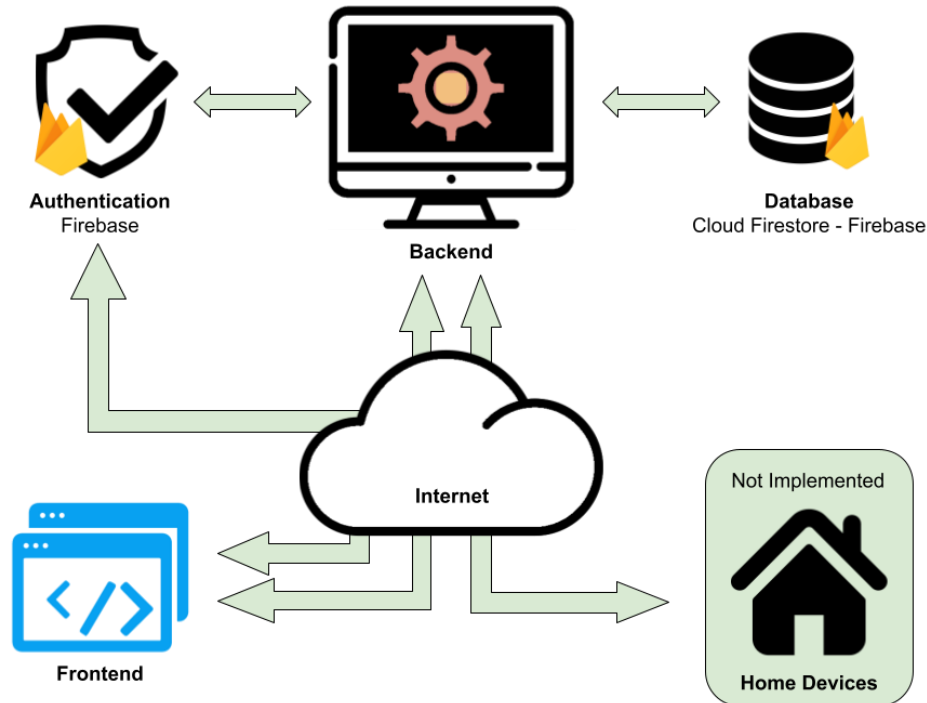


Figure 2 - Solution Architecture (Global View).

3.1. The Authentication

The Authentication module is straightforward. As mentioned before, we make use of the Firebase Authentication system where we log the users and their respective password, maintaining this hashed in order to maintain the security of the system. The users and the amount of new registered users can be checked through Firebase's interface as well as blocked and removed. Each user has an automatic ID associated to it.

<input type="text" value="Pesquise por endereço de e-mail, número de telefone ou UID do usuário"/> Adicionar usuário				
Identificador	Provedores	Data de criação	Último login	UID do usuário ↑
rod_sousa@hotmail.com	✉	17 de mai. d...	17 de mai. d...	8cYdLEXVCrTyHHeDE40KXstiCoA2
isabel.srsoares@gmail.com	✉	17 de mai. d...	18 de mai. d...	NSe0rULEMUWpS3d1PEpqFc2DT...
manuel@hotmail.com	✉	17 de mai. d...	18 de mai. d...	TOY2oGBkZITXliFZehd0rqQHIN62

Linhas por página: 50 1 – 3 of 3

Figure 3 - Authentication Firebase.

3.2. The Database

Since Cloud Firestore is a NoSQL database we translated the proposed database model in Figure 1 into a non-relational database. As it can be seen in Figure 4, we defined, in total, eleven collections each with its own set of attributes:

- **Users:** a name, an array of references to its houses and an array of references to its preferences.
- **Houses:** a name, an array of references to the users it pertains to and an array of references to its floors.
- **Floors:** a name, a reference to the house it belongs and an array of references to its divisions.
- **Divisions:** a name, a reference to the floor it belongs and an array of references to its devices.
- **Devices:** a name, a favorite boolean, a reference to the division it belongs to, a reference to its device type, an array of references to its current property values and an array of references to the values history associated to this device.
- **Device Type:** a name, an array of references to properties.
- **Property:** a name, a boolean to represent if the value is writable and a type, depending on if the type is:
 - **Scalar:** a number representing the minimum value, a number representing the maximum value, a number representing the step, the units as a string.
 - **Enumerator:** an array of possible values as strings, the index being the key for the value, the string the value itself.
- **Property Values:** a value, a reference to the device it pertains to and a reference to the property it pertains to.
- **Value History:** a reference to the property value and the timestamp in which it was set.
- **Preference:** a name, a boolean representing if its pendent (if it was not yet accepted by the user), a timestamp representing till when the scheduling is disabled, a reference to the user it pertains to, an array of references to the property values which are to be set and an array of references to the schedules for this preference.
- **Schedules:** the timestamp in which it was set.

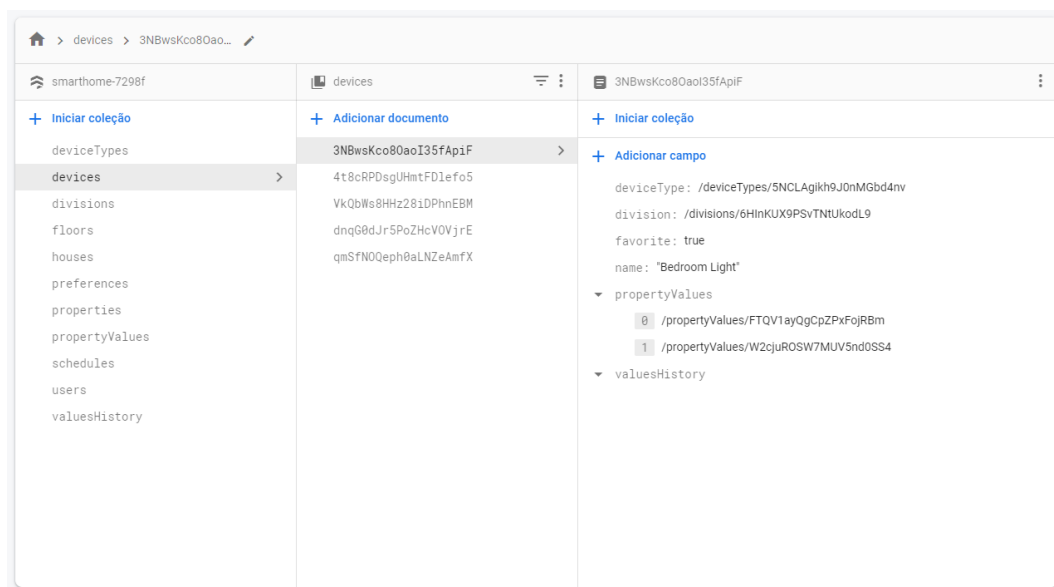


Figure 4 - Database Firebase Interface (device collection).

3.3. The Backend

The Backend deploys an API with twenty-nine endpoints defined in *server.ts* that can all be accessed remotely. Each one of these endpoints is crucial to our system and they make use of the 2 types of requests: GET requests and POST requests. Some examples of endpoints created by our backend are: GET for documents for each one of collections in our database, the POST of new houses, floors, divisions, devices, user and accept changes of property values.

The Backend communicates with the Firebase Authentication and Cloud Firestore through the firebase npm package. For simplicity we defined a *firebase.ts* file which serves as interface between our backend and firebase implementing:

- *get_reference*: which gets a reference to a document through the collection and document name.
- *get_all_references*: which gets references for every document in a specified collection.
- *get_document*: which get the firebase document given its name and the collection name.
- *change_document*: given a collection name, document name, and an Javascript object with the attributes and values to be updated in the document and returns a reference to the document.
- *create_document*: creates a new document in the given collection and the data to be allocated to this document, the id is automatically generated, and returns a reference to the new document.
- *create_document_with_id*: creates a new document in the given collection, the data to be allocated to this document and the id to be associated and returns a reference to the new document.
- *get_user_from_email*: from an email return a User record stored in the authentication system.

For each one of the collections that the database contains, the database has implemented a class each one having a load function which accepts the data from a firebase document retrieved with one of the functions defined above. By doing this, we translate the firebase information into backend objects, which not only is convenient but also allows for the introduction of logic into these objects. In turn these objects are sent through the API by converting them into JSON and placed in the body of the response.

Backend also has an *utils.ts* file which implements more complex logic than the one described in *server.ts*. In this new file, there are functions for the creation of users, houses, floors, divisions, devices, addition of users into already existing houses, changing of property values, getting all favorite devices of a user and changing schedules and preferences. These functions are implemented here, due to the higher complexity and because they involve other updates in the references of another documents.

There is one more file in backend *intelligence.ts* which is responsible for the automatic detection of routines in users' schedules. The main concept behind this is that every time a user changes a property value this change is introduced in intelligence (its value and property), and occasionally, in a time interval specified by us, this module runs through the changes and checks if this change is considered routinely. The process of identification of routines is quite simple:

- we run through the history of the values limiting how far back we go into this history (predefined seven days)

- For each value history in this property, we convert them to binary, being 1 if it changes to the value stored in change and 0 otherwise.
- We then group them by day and create samples through the day.
- We then do the average; the moment the average surpasses a threshold we have identified a routine to be scheduled.

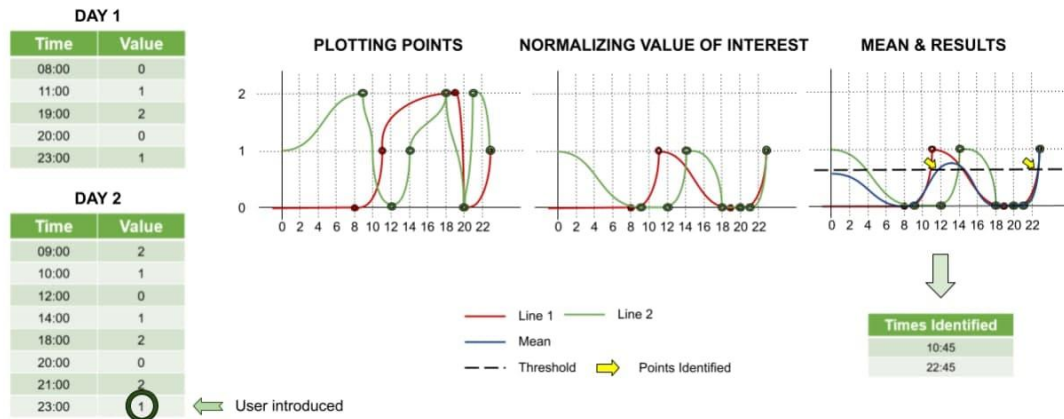


Figure 5 – Exemplification of the process (using only 2 days).

3.4. The Frontend

The Frontend can be divided into three different directories responsible for totally different aspects of the application: The utilities, which can be further divided into API Endpoint and Store, the Components and the Classes.

3.4.1. The Utilities: Endpoint API

The endpoint through which the frontend communicates with the backend is defined in `./utils/endpointAPI.ts`. This file implements and instantiates a class named Router which serves as interface between the frontend and both the backend and the firebase authentication system directly.

We communicate straight with the firebase authentication system, due to security reasons. In order to not pass the password in unprotected requests to our backend, we make use of already implemented firebase functions which deal with the login process correctly and safely.

When the router communicates with our API (backend), it is also responsible for converting the information it receives and sends appropriately. Meaning that it builds the request bodies and converts the response information into the appropriate frontend classes.

3.4.2. The Utilities: Store

As mentioned before, React deals with state in a component wise approach, but the current user must be stored globally, so we make use of React-Redux which allows the creation of a global store which gives a state to the overall application.

- The file `./utils/store/type.ts` defines what the state holds and the available actions through constants;
- The file `./utils/store/actions.ts` defines the functions that the store supports;
- The file `./utils/store/reducer.ts` creates a reducer from the possible actions;

- The file `./utils/store/store.ts` create the store itself;
- Lastly and more important, the file `./utils/store/storeEndpoint.ts` which creates an interface for the store that can be used in an easier way in each React component.

3.4.3. The Classes

For simplicity, we also implemented classes in frontend. When the router receives an object (in JSON format) from the backend this is translated into a frontend object through load functions. Essentially, we have the same classes implemented in the backend and frontend the difference being in the logic. Frontend classes besides having a load function also have defined load functions for its attributes since some only have initially references to other objects not yet gathered from the database. In Figure 6, we display the functions that the class `User` has implemented in it.



```
import Router from '@utils/endpointAPI';

export class User {
  // ATTRIBUTES

  constructor(id: string, name: string, paths_preferences: string[], paths_houses: string[]) {...}

  // ===== LOAD OBJECTS =====
  async load_preferences() {...}
  async load_houses() {...}
  async load_floors() {...}
  async load_divisions() {...}
  async load_devices() {...}
  async load_favorites() {...}

  // ===== SELECTION OBJECTS =====
  select_house(id: string) {...}
  select_floor(id: string) {...}
  select_division(id: string) {...}

  // ===== CREATE OBJECTS =====
  async add_new_house(house_data : {'name': string}) {...}
  async add_new_floor(floor_data : {'name': string}) {...}
  async add_new_division(division_data : {'name': string}) {...}
  async add_new_device(device_data : {'name': string, 'favorite': boolean, 'deviceType_id': string, 'propertyValues': {'property_id': string, 'value': number }[]}) {...}
}

export function load_user(id: string, data : {name: string, paths_preferences: string[], paths_houses: string[]}) : User {...}
```

Figure 6 - Classes on Frontend.

3.4.4. The Components

The components are defined in the directory `./components/`. The components created can be divided according to their use in the following way:

- The *LoginUser* component used as an initial menu for the user to do login or sign in.
- We developed a component called *MainPage* which is responsible for the overall framework of our application, where the top bar and sidebar are defined, and space is allocated for panels which change depending on the menu selected by the user.
- Several *Panels* that are placed inside the space allocated by the *MainPage* component dependent on the current menu. The possibilities are: *InfoPanel*, *DevicesPanel*, *SchedulesPanel*, *FavoritesPanel* and *SettingsPanel*.
- The *SelectionPanel* which is used only as an auxiliary component for convenience and less verbose code throughout the project.
- Several *Dialogs* that are overlayed over the current application state using modals. The possibilities are: *AddComponentDialog*, *AddDeviceDialog*, *DeviceDialog*, *PreferenceDialog* and *AddPreferenceDialog*.

4. Details of the Solution

4.1. Login

When the user first opens the *Home Smart Home* application it is greeted by the following screen in Figure 7. In this screen, the user can either insert its email and password and **login**, in the case, that is already registered in our system or insert its name, email and password to be *signed in* on our system. Either one will result on the user being authenticated through the system.

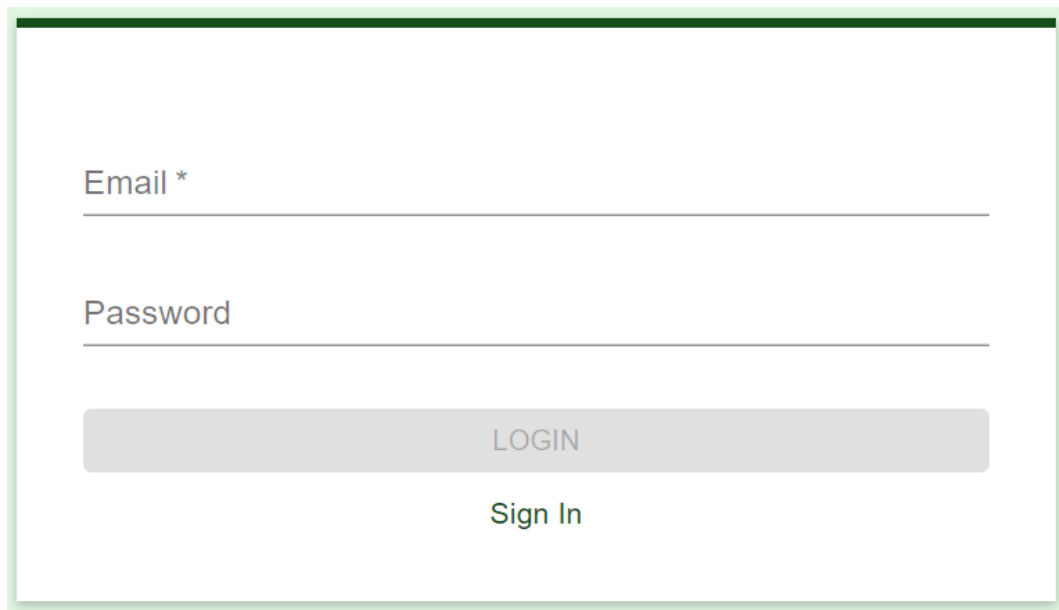
The image shows a user login interface. It features two input fields: the first is labeled 'Email *' and the second is labeled 'Password'. Below these fields is a wide, light gray button with the text 'LOGIN' in capital letters. Underneath the button, the text 'Sign In' is displayed in a smaller, green font. The entire form is enclosed in a thin green border.

Figure 7 - User Login.

4.2. Framework of the Application

Independently of the menu selected by the user the application will always display: The **Top Bar** in which the user can open or close the **Side Bar** (a drawer bar) and where the current menu name is displayed. The **Side Bar** either opened or closed allows for the selection of the current menu.

4.3. Division Selection

When the user is properly authenticated, it will be redirected to the main page with the menu of **Division Selection** being shown by predefinition.

The Division Selection menu is composed of several elements in which the logic for house selection, floor selection and division selection is accessible, respectively. Since the selection follows a hierarchy, it means that for example the box for the floor selection will only be displayed in the case that the house is already selected and the same logic is implemented between floor and division.

The element can be selected by the user simply by clicking on it. For each one of the 'boxes' a new component can also be added by clicking the corresponding 'plus' button. When this button is clicked, a dialog opens in which the user can introduce the name of the house, floor and

division that is trying to create and a button to confirm this new component that when pressed saves this new element both locally and on the database.

In Figure 8, we demonstrate how the application would appear when the user has already selected a house, in this case 'Together House', a floor, in this case 'First Floor', and a division, in this case 'Living Room'. At any point, the user can freely change its previous selections simply by clicking other elements.

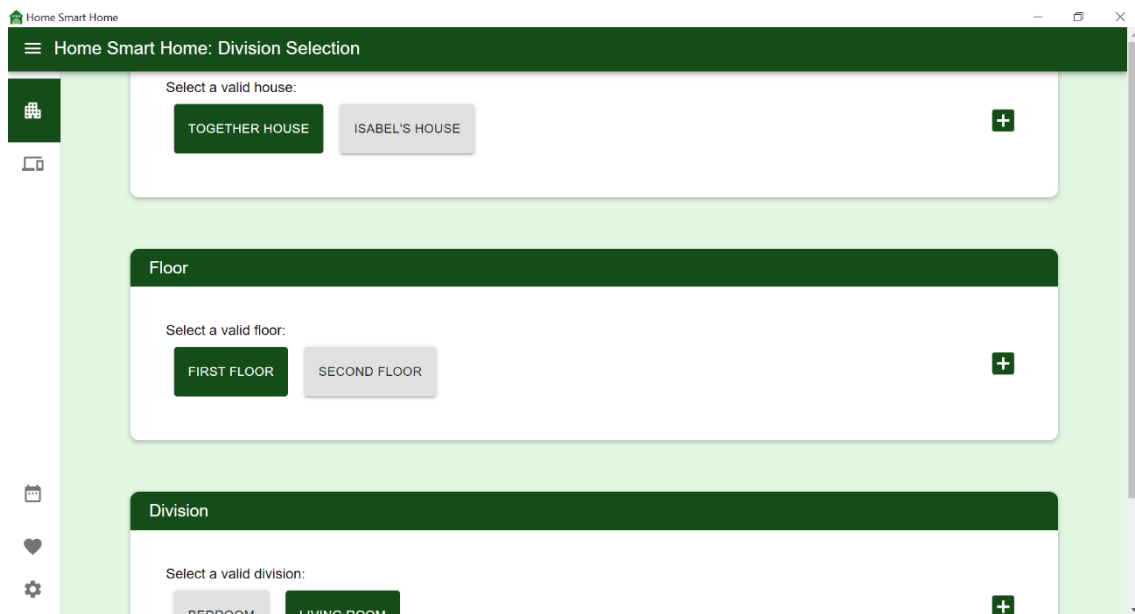


Figure 8 - Division Selection.

4.4. Devices

If the user selects the menu **Devices** than a similar box to the previously displayed is shown. In this box, the various devices in the selected division are displayed. Each device is displayed through a button like element in which is visible the name of the device, the type of the device and a heart icon that displays if the device is favorited or not, through a filled icon or a not filled icon, respectively.

Just like in the components mentioned before, the user can add a new device to the current division devices, although the interface is more complex in this case. When the user clicks on the add button located on the right side of the box a dialog open asking the user to input the name of the device if it is a favorited device and the type of the device. Depending on the device type selected, the user will need to input the initial property values to be set on the device, being the exception the properties which are not writable in which the program will internally generate a random value to populate that property.

Intuitively, if the user clicks on a device to get more information about the device, a dialog will open. In this dialog once again, the name of the device and the device type is displayed, as well as the icon representing if the device is favorited or not. The only difference being that this icon is an active button, that when clicked changes the favorited property accordingly, if its already favorited becomes not favorited and not favorited if it was favorited.

Besides displaying these basic attributes of a device, it is also displayed two ‘accordions’ (elements in a box which can be closed or opened), which we implemented in a way that at any point only one of these accordions will be opened, in order to save space and not clutter the user with non-pertinent information.

4.4.1. Device properties and its values

The first accordion element displays **the device properties and its values**, one by one in a column. As displayed in Figure 9, each property has displayed on the left side its name and the type as well as pertinent information, if it is of type Enumerator or Scalar, and if its Scalar properties such as the minimum value, maximum value, step value and units. On the right side has a text field which can be changed (in the case of being an enumerator it provides a selection menu). If the value to which is changed is invalid, then the line becomes red signaling that the value introduced is incorrect. If the property being displayed is not writable, meaning that the value it displays cannot be changed (likely a sensor), the text field is deactivated, and the user is unable to change its value. Finally, if the user has introduced any valid changes to the properties of the device the save changes button will be activated, once pressed it will save the current changes of properties to database and locally.

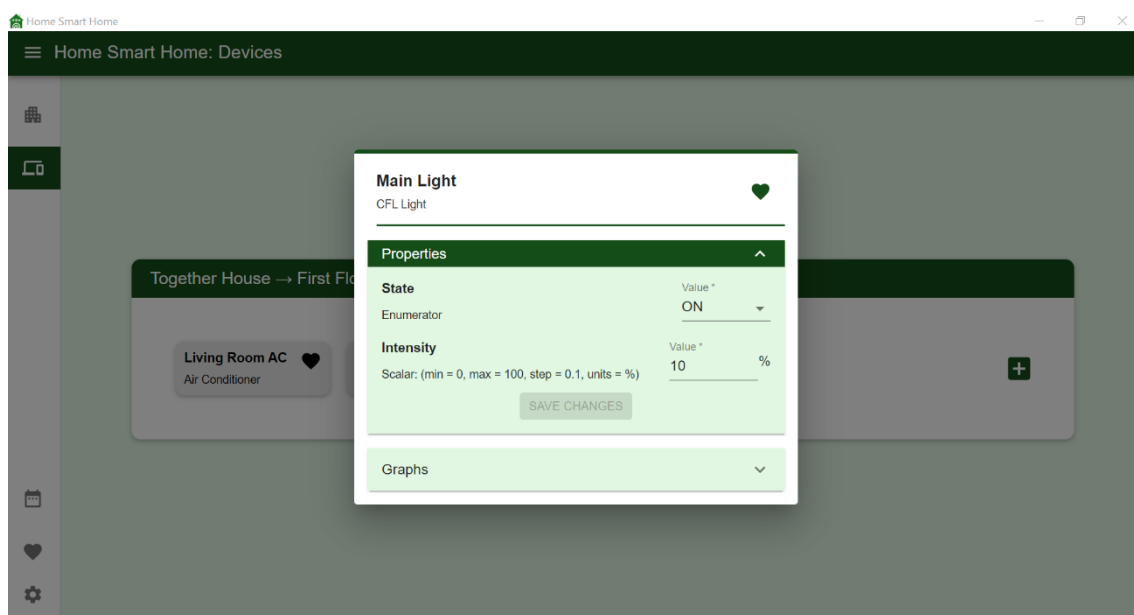


Figure 9 - Properties of a device (in this case CFL Lights).

4.4.2. Graphic Visualization

The second accordion, as shown in Figure 10, displays a **graphic visualization** of the changes made to this device in the last week for a selected property, where each line represents the weekday, the x axis the hours of the day (in this case grouped in units of two) and the y axis the value of the property at any given time. By displaying the changes graphically, the user has a better interface to gage how he interacts with the device even possibly identifying a routine in its own behavior. It also serves as an aid to the user when the backend identifies some routine in its pattern of behavior, the user can then verify visually if that routine really exists.

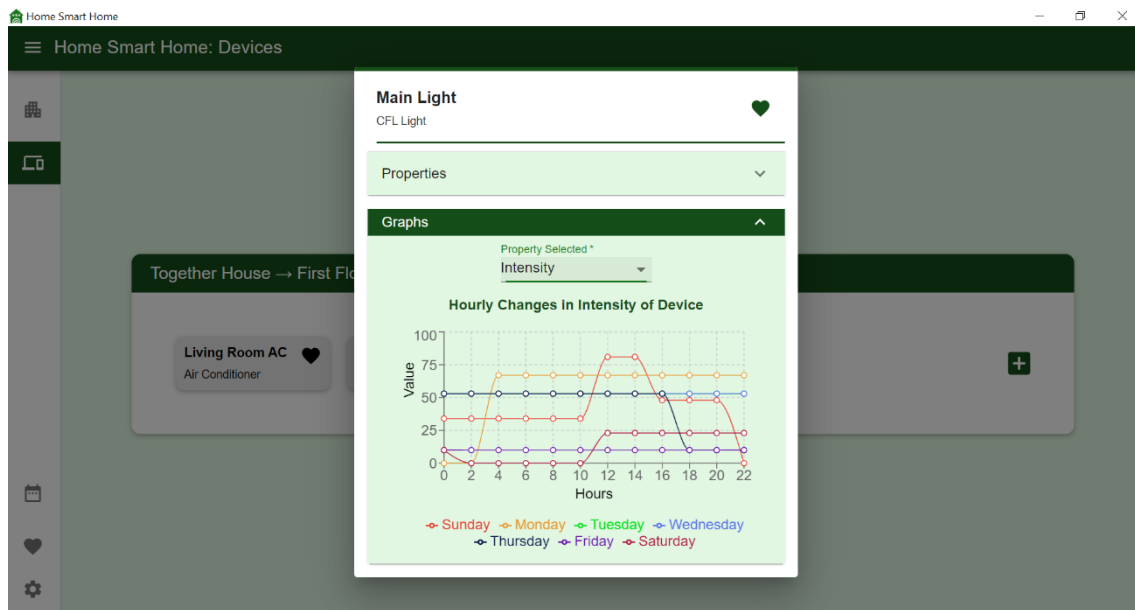


Figure 10 - Graphs that describe the hourly changes in this device.

4.5. Scheduled Events

If the menu selected is **Scheduled Events**, then two components are displayed both having preferences / schedules represented through the button like elements just like before. The difference being that the preferences in the first (top) component are yet to be reviewed by the user and were the routines automatically identified by the system by analyzing the changes that the user made, while in the second component the preferences were either already accepted by the user or directly created by the user.

When the user clicks on a preference a dialog opens, exemplified in Figure 11. In this dialog, the user can change the name of the preference, the properties and values that the preference include and change the scheduled times for it. Both the properties and scheduled time can be added or removed freely.

The difference between the two components on this menu is that:

- if the preference has yet to be accepted, at the end there are two possibilities either of accepting or rejecting the schedule formulated by the system;
- if the preference was already accepted than one button will update straight away the properties to the values predefined (by doing this, the user not only can schedule events but can directly affect multiple devices and their properties all at once) and another button which deactivates the scheduling for twenty-four hours.

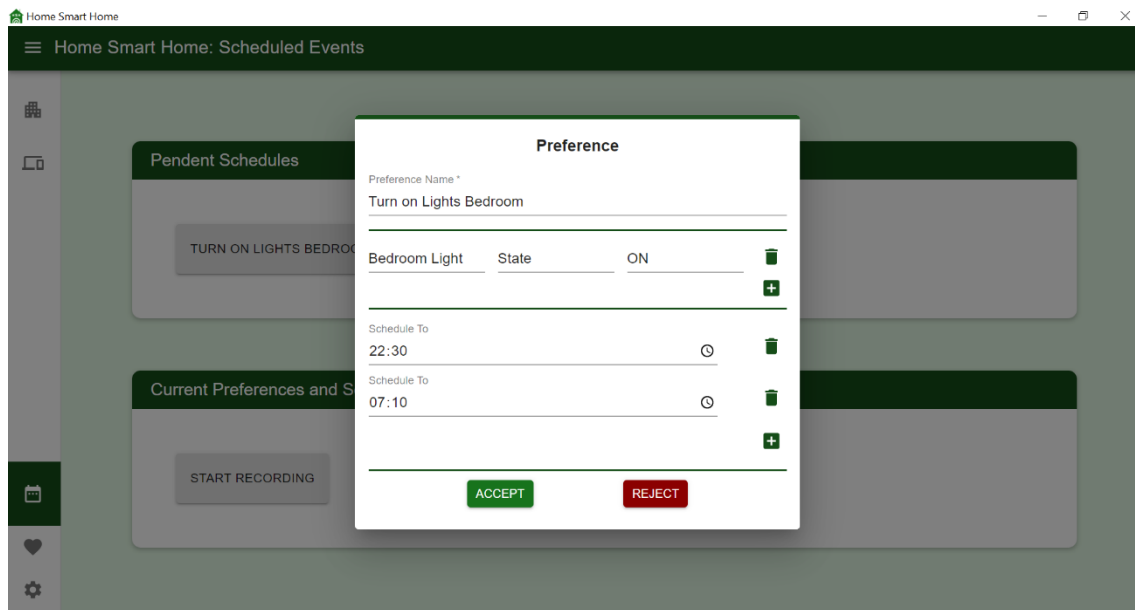


Figure 11 – Preference with pendent schedules.

4.6. Favorites

When the user selects the menu **Favorites** a component is displayed similarly to the one displayed in the menu **Devices**, in fact, in terms of logic is the same, as it can be seen in Figure 12. The user can click on the device to access its properties and the graphic visualization of changes, the only differences are that the devices displayed in this component are all his favorite devices (not only the ones in the selected division) and that there is not a button to add devices on this component, since it would not make sense to add a device from this point. For the same reason, we considered better not having a button to add more favorites in this panel.

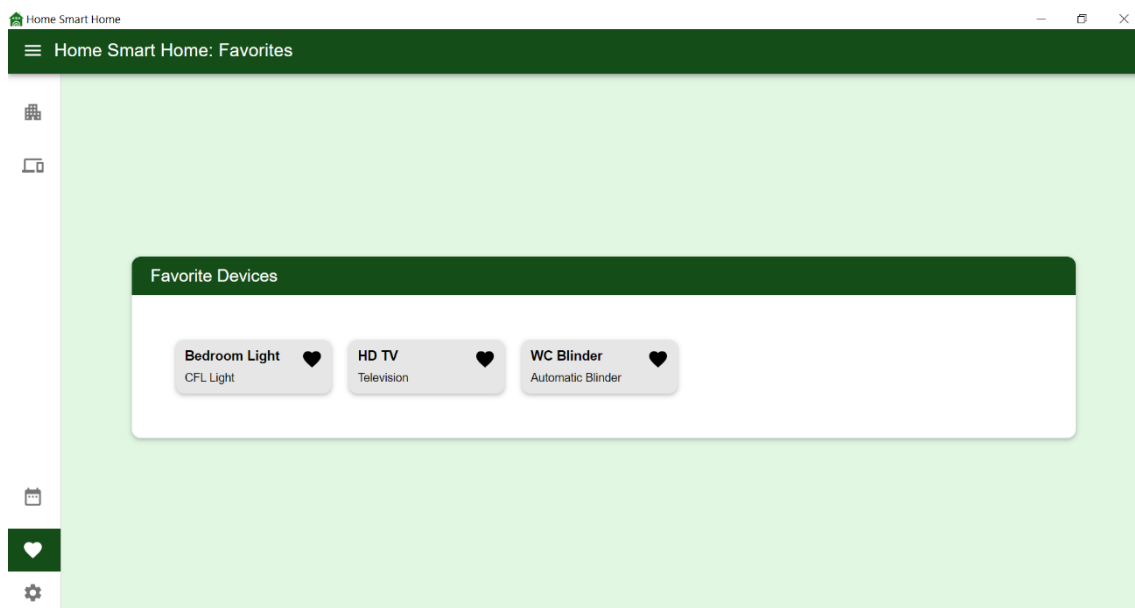


Figure 12 - Favorites Devices Panel.

4.7. Settings

As it can be seen in Figure 13, when the user selects the menu **Settings**, a component is displayed with information of user namely user's photo, name and email. In this panel, user can also logout from the application.

Then, the user has an option to add another user to one of his houses. In the case that the user wants to add another user to one of his houses, the user must select the house to share with this new user and input the new user's email, by doing this we allow two or more different users to share a common household.

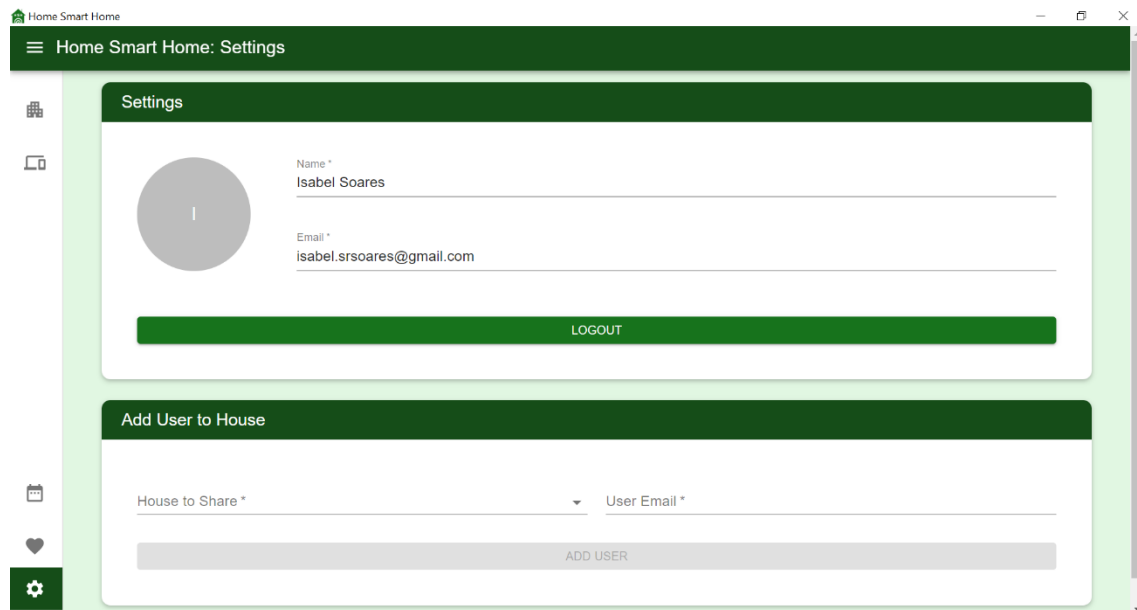


Figure 13 - Settings Panel.

5. Evaluation

During the development of the system several strategies were used in order to evaluate and test out the performance and correctness of the implementation.

More specifically in the backend, one of the most important tools used was *Postman* [14], an application that allows for the test of API's. *Postman* allowed us to quickly create and send requests directly to our local backend. We did this, since testing it with our own frontend application could become cumbersome when trying to identify where was the fault of the system.

Another tool that was used during the development of the backend were the interfaces provided by *Firebase* to evaluate the state of the database as well as change directly some values in it.

The methodology for evaluating the backend usually followed the steps:

- Implement the endpoint and the logic on the backend without committing results to the database and instead simply logging them.
- Test the logic using requests sent through *Postman*.
- Change the backend logic to commit the results to the database.
- Test once again using requests sent through *Postman* but now verifying if the changes were correctly inserted through the *Firebase* interface.

With the purpose of evaluating the frontend, we would typically use the Developer Tools, like the one provided by Google Chrome. With this tool, we could see the HTML that was created using the React framework, alter the CSS properties of classes, as well as check the console for any error or information that we might want to be outputted to it. Only after this last step, the system was tested completely. We would then use our own application to cause changes either on the database, frontend or both and use the interfaces provided by *Firebase* to verify if the changes were completed successfully.

At the end, the performance of the system overall was tested. This was done through the interface provided by *Firebase*, which displays some relevant statistics. We did a good amount of stress testing to the backend by quickly firing requests to it and it behaved accordingly, even reaching a couple thousand reads to the database. We quickly verified that the bottleneck of the overall system was in the communication between frontend and backend mostly due to both being executed locally on our machines and not in dedicated servers.

6. Conclusion

With our project we feel like we accomplished everything we had set out to do. We developed an intuitive and easy to use interface to manage devices part of an automated home system. With our system, the user can add multiple devices, change their properties, schedule events to trigger changes in devices and even share houses with other users that can be co-managed.

Our system even has the addition of being completely and easily scalable in terms of device types and new properties, and having an intelligent and automated module in it, which allows for the detection of routines in users interactions.

Although, as mentioned above, we were successful in achieving what we had set out to do there a lot of other features and changes that given the time we would like to do.

First and foremost a **code refactoring**, like with any other system that is developed, it comes a time when a refactoring could be beneficial, we feel like, with time this would be one of our priorities.

Implement an Authorization service in our backend. Now we simply have an authentication service implemented through Firebase, with time we would like to use Firebase Authorization Service as well as define rules in our backend to deal with this authorization schema.

Create an API dedicated with interacting with devices, the current API is dedicated to frontend applications, communicating and displaying results, although the same API is able of supporting changes in properties, we would like to implement a full-fledged API with the intent of communicating with the devices.

Implement a proof-of-concept low-level device which would communicate with the previously mentioned API, transmit its property values and accept changes from the server.

Develop a mobile application which would improve the commodity of using our system: the current frontend desktop application could be quickly interchanged by a mobile android application, Electron supports this change, but the application was not conceptualized with this in mind so certain things would not adjust well on a mobile screen. We would like to create another frontend application this time intended for mobile devices.

Offer the **possibility to users of exporting and loading from and to the database** a house schema: although the database supports loading and exporting of data, this is a feature that is only given to paid subscriptions of *Firebase*, although this feature could be fully implemented from our side, this would take a good amount of time and effort.

7. References

- [1] Nunes, R., *The DomoBus System Specification Language*, Technical Report, Instituto Superior Técnico, University of Lisbon, V2.0c 29/02/2016.
- [2] Nunes, R., *DomoBus - A New Approach to Home Automation*, 8CILEE - 8th International Congress on Electrical Engineering, Portugal, July 2003, pp.2.19-2.24.
- [3] Google Developers (2021), *Firebase*, <https://firebase.google.com/docs> (accessed 18 May 2021)
- [4] NPM Inc. (2021), *Node.js*, <https://nodejs.org/en/docs/> (accessed 18 May 2021)
- [5] Sharp R. (2021), *nodemon*, <https://www.npmjs.com/package/nodemon> (accessed 18 May 2021)
- [6] StrongLoop, IBM and other expressjs.com contributors (2017), *Express*, <https://expressjs.com/en/5x/api.html> (accessed 18 May 2021)
- [7] Facebook Inc. (2021), *React*, <https://reactjs.org/docs/getting-started.html> (accessed 18 May 2021)
- [8] Open JS Foundation (2021), *ELECTRON*, <https://www.electronjs.org/docs> (accessed 18 May 2021)
- [9] Abramov D. and the Redux documentation authors (2021), *React Redux*, <https://redux.js.org/introduction/getting-started> (accessed 18 May 2021)
- [10] Facebook Inc. (2021), *React Introducing JSX*, <https://reactjs.org/docs/introducing-jsx.html> (accessed 18 May 2021)
- [11] Material-UI organization (2021), *Material-UI*, <https://material-ui.com/getting-started/installation/> (accessed 18 May 2021)
- [12] Zabriskie M., Uraltsev N., Morehouse E. (2021), *axios*, <https://axios-http.com/docs/intro> (accessed 18 May 2021)
- [13] Recharts Group (2021), *< Recharts />*, <https://recharts.org/en-US/api> (accessed 19 May 2021)
- [14] Postman Inc. (2021), *POSTMAN Learning Center*, <https://learning.postman.com/docs/publishing-your-api/documenting-your-api/> (accessed 18 May 2021)
- [15] Amorim F. (2020), *Setting Electron + React with Typescript*, <https://dev.to/franamorim/tutorial-reminder-widget-with-electron-react-1hj9> (accessed 19 May 2021)