**Wrocław University of Science and Technology**
**Faculty of Information and Communication Technology**

Field of study: **INF-PPT**
Speciality: **-**

BACHELOR THESIS

# Implementation of the Smart Mirror desktop application

Krzysztof Małczak

Supervisor
**Marcin Zawada, BEng, PhD**

Keywords: Smart Mirror, Electron, React.js

Wrocław 2022

# Contents

# Introduction

Rapid technology development influences all areas of human life, however in the last couple of years one of the most affected regions was definitely a home environment. Every year new smart home solutions become more accessible, as advanced smart home devices are introduced with enhanced functionalities and features. Newly created devices become much more compatible with each other by increasingly using common protocols or communication mechanisms, since their integrity is one of the most important issues in this topic, as the more devices can be linked together, the better smart home infrastructure can be built.

Integration of some of these smart upgrades may not only make life more comfortable and enjoyable by providing devices such as smart light bulbs, automated vacuums, or by supplying convenient access to popular streaming services like Spotify, Netflix or YouTube, but also tackle more serious tasks e.g. devices become responsible for home security systems or have a significant impact on increasing the work efficiency of users. That is due to the fact that using smart home appliances is not limited to delivering weather forecast or news, they can also be used as powerful tools for time management improvement by providing mechanisms for events and reminders scheduling. In other words these systems and devices are not just for entertainment purposes anymore, but they also fulfill some substantial needs.

Smart Mirror system was designed and implemented in order to contribute to the expansion of smart home devices family, by providing a software that can be easily incorporated on the Raspberry Pi platform, which in conjunction with couple of additional components becomes powerful intelligent mirror, providing such features as:

- Multi-user access
- Various widgets usage and modification
- Web access

As mentioned in the first paragraph, devices of this kind tend to be part of more oversized products or ecosystems, meaning their integrity with other devices is crucial to make them reliable. Therefore, the Smart Mirror comes as software integrated with Google services, as many other already existing devices consume the same resources (Google APIs), which allows keeping all of them in sync, e.g. adding a new calendar event via voice command given to Google Home smart speaker will be reflected in the user interface of the Smart Mirror calendar widget, as they use shared data storage for this purpose.

The scope of this thesis is the implementation of the central pillar of the Smart Mirror system, namely an Electron-based desktop application, and its integration with the remaining system components.

The thesis was divided into six chapters. The first one briefly outlines the thesis background and defines the scope and contents of the thesis. The second focuses on defining the exact purpose of the Smart Mirror system, including its functional requirements; it also provides a short analysis of solutions satisfying similar functionalities. The third part is devoted to the system's design, including its general architecture and diagrams essential for understanding how the system operates. Chapter four contains implementation details of the essential parts of the system, including Google and face recognition module integration, and an explanation of workarounds for problems encountered during the implementation phase. As mentioned in this introduction, to use the Smart Mirror system as software for an intelligent mirror, it must be built first, meaning a strictly defined target environment containing particular components should be assembled. A detailed description of its composition and how to install the software is presented in chapter six. At the end of the thesis, there is a summary of the work. Also, potential directions of the project's development are identified.

# Purpose and analysis

This chapter outlines what the purpose of the system is, what problem it solves, and what functionalities are provided to do so. It is also compared to both: open-source and commercial solutions from the same technology branch.

## 2.1 Purpose of the system

A smart home refers to a convenient home setup where appliances and devices can be controlled remotely and connected to provide many life-simplifying functionalities, e.g. allow users to control home security, temperature, lighting, and home theater remotely. They can be set up through a direct hardwired connection, or like in the case of the Smart Mirror, which uses an internet connection for integration purposes, they can use a wireless means of communication. Devices from this family are also usually linked with mobile applications that can affect their behavior or content. It is also somewhat related to the Smart Mirror system, as any mobile application capable of altering data hosted by Google API that the Smart Mirror software consumes affects the content displayed in this system (e.g. Gmail mobile application[1]).

The general purpose of smart home devices usually comes down to both: life standards and work efficiency improvement. This also applies to the Smart Mirror system, as it includes many simple and yet essential tools of everyday use, such as:

- e-mail box
- digital calendar
- multifunctional clock
- weather forecast
- news
- music player
- web browser

inside a single application. Which improves a standard of life by providing easy and convenient access to entertainment but can also enhances the quality of work or time management by supplying time scheduling functionalities.

The problem that the Smart Mirror project is trying to solve is the possession of multiple devices that fulfill single responsibilities (e.g. home weather station, digital clock, Bluetooth speaker, etc.), by enclosing all of them in the form of user-friendly widgets inside of a piece of furniture, as the target device of the system makes up the inner part of an actual mirror (see chapter 5 for more target environment composition details).

[1] Mobile application allowing to interact with email inbox provided by Google

## 2.2 System requirements

The analysis phase is inevitably present in every software development process, no matter what exact SDLC[2] has been chosen. Its most critical part is the analysis of requirements, as it allows to define and fulfill user needs.

In this section, two types of Smart Mirror system requirements will be presented - functional requirements, summarizing what operations should the user be able to perform, and non-functional requirements, summing up everything else about the system but the functionalities.

### 2.2.1 Functional requirements

Functional requirements were divided into two subsections, namely general and widget-specific ones. They describe what the user can do when interacting with the system as a whole and with a particular widget, respectively.

1. General
    - register via Google account
    - login using face recognition
    - login using Google credentials
    - select screen orientation
    - modify the content of the set of visible widgets
    - modify the arrangement of widgets (layout)
    - browse web
    - log out, shut down and restart the system
    - interact with particular widgets
2. Widget specific
    - Calendar widget:
        - view events from Google calendar
        - select the exact day of displayed events
    - Mailbox widget:
        - view a list of recent unread messages from Gmail service
        - view message content and sender
        - mark message as ’read’
    - Digital clock widget:
        - view current hour
        - view current date including day of the week
        - view time zone
        - switch between clock modes (default, stopwatch)
        - use stopwatch functionality (start, stop, restart, lap)
    - Radio player widget:
        - choose music genre
        - switch between radio stations
        - view name and logo of current station
        - pause and play the music at any time
        - change the volume level

[2] Software Development Life Cycle - refers to a methodology defining exact stages of software development

  - mute the widget
- Weather forecast widget:
  - view today's weather (weather icon, short description, current temperature, temperatures range, wind speed, humidity)
  - view weather forecast for the next 4 days (weather icon, short description, temperatures range)
- News feed widget:
  - read news headers
  - open web page containing a full article
- Logged in user indicator widget:
  - view avatar, username and Gmail address of currently logged in user

### 2.2.2 Non-functional requirements

This subsection contains a list of non-functional requirements, excluding the choice of programming language, frameworks and libraries - these important aspects are discussed thoroughly in the implementation chapter (see 4.1).

- *Extensibility* - it is very easy to extend the set of system's functionalities, by implementing new widget components;
- *Flexibility* - system's architecture and implementation approach allows dealing with future changes in requirements;
- *Integrability* - system supports different communication protocols which can be used for additional modules integration;
- *Response time* - user interface of the system quickly responds to user's actions;
- *Adaptability* - user interface of the system adapts to the screen ratio and size (is responsive), as these aspects are completely dependant on users choice;
- *Fault tolerance* - system can continue work even if external resources cannot be provided (e.g. widget displayed on the screen, that consumes unavailable API does not break the application);
- *Reusability* - important parts of the system are completely reusable (e.g. face recognition module, UI components);
- *Dependency on other parties* - in order to keep system in sync with other devices it is based on Google APIs, meaning it depends on Google services and other external APIs;
- *Usability* - system is easy to use for people of all ages and education levels, by providing user-friendly interface;

## 2.3 Systems and devices satisfying similar functionalities

As mentioned in the introduction, smart home solutions and intelligent home appliances are becoming more and more common on the market of electronic devices. This growth affects the general popularization of electronic devices and, most importantly, the increasing competition, as the more devices are built, the more they satisfy similar functionalities. This issue also applies to the case of the Smart Mirror system, as even though it is not an entirely innovative idea, it still brought some functionalities that do not exist in similar solutions.

The main inspiration for the system described in this thesis was *Magic Mirror* project [9]. Which is by far the most popular solution of this type. Similarly to the Smart Mirror, it allows to transform display of any kind into an intelligent mirror, by connecting it to a Raspberry Pi computer equipped with *Magic Mirror* software. It is an open-source project with an advanced modular architecture, allowing any contributors to extend the system's functionalities by implementing modules. This fact distinguishes it from the Smart Mirror, as the latter only allows the addition of new features by altering existing source code. These plugins can be installed independently by users, however, the default version of the system already contains a set of pre-installed modules, which provide very similar functionalities to the ones included in the Smart Mirror, e.g. clock, calendar, weather forecast, a news feed, and by authenticating in external services such as Google or Spotify users also gain access to mailbox and music player.

*Magic Mirror* is accessible as software that has to be directly installed on the Raspberry Pi OS. It is another thing that differs it from the Smart Mirror, which does not require the user to take any additional steps in order to include the system on their machine, besides placing an SD card in the Raspberry Pi slot (see 5.2 for more details). Also, the Smart Mirror system delivers a higher standard of UX[3], as it allows users to arrange widgets in any way by providing a modifiable screen layout (see 4.7.2), which is not available in the *Magic Mirror*. Log in with face recognition is one of the default functionalities of the Smart Mirror, while the competitor requires an installation of an additional module to provide this feature. The last functionality that the Smart Mirror surpasses this other solution is direct access to the Chromium browser, which allows users to browse the web freely and visit their favorite web services such as Netflix, YouTube, or Instagram (see 4.7.1 for additional details).

Another piece of technology that satisfies functionalities similar to intelligent mirrors is *Google Nest Hub* speaker. Although the producer refers to it as a speaker, it actually is a touchable tablet mounted on top of the base that contains the speaker. It comes as a pre-built device, so the user does not have to assemble it all by himself in contrary to both Smart and Magic Mirror solutions (see 5.1). However, similarly to appliances mentioned in the previous paragraphs it also allows to listen to music via Spotify, view news, check the weather, and watch Netflix.

The main improvement introduced by the *Google Nest Hub* is voice and gesture control, meaning it is possible to play or pause a song or video, snooze an alarm, and stop a timer without touching the screen. Quick gestures feature uses motion sensing to detect hand moves, meaning it does not need a camera. Also, its main advantage is complete Google support, as it is a Google product. It can be easily integrated with any smart home device provided by this vendor, such as smart speakers, intelligent cameras, and doorbells. However, its layout is predefined and cannot be changed by the user as in the Smart Mirror.

[3]User Experience - how user experiences the product

# System design

In order to ensure a smooth development phase, the design of the software had to be defined first. This part briefly describes the architecture of the system as a whole and then focuses on its main pillar's flow and operation presented in the form of diagrams. At the end of the chapter, database content has been described in depth.

## 3.1 System architecture

Four main components can be distinguished in the architecture of the Smart Mirror system:

- Desktop application
- In memory database
- Set of external APIs
- C++ face recognition module

All of the elements mentioned above, excluding external APIs, operate on the same machine - user's machine (see chapter 5). Figure 3.1 illustrates this architecture.

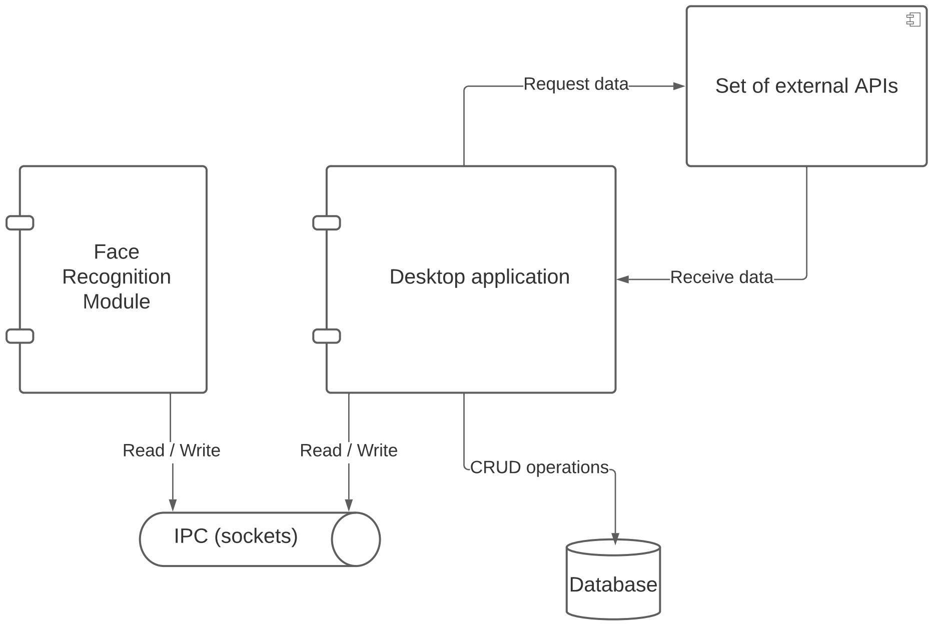


Figure 3.1: Smart Mirror system architecture

This thesis focuses on the desktop application, which is responsible for interacting with users by displaying interface and reacting to their actions accordingly. It also provides the logic required to process, consume and dispatch data to all of the remaining components. It uses both IPC[1] and HTTPS[2] protocols to communicate with face recognition module and external APIs respectively.

The face recognition module is completely independent of the desktop application, however, it is consumed by it via socket communication. It contains all of the logic related to the face recognition feature of the Smart Mirror system, including registering users' face metadata, saving it, and using it later on in the default login approach proposed by the system (see 4.5.1).

External APIs refer to any resources located outside the user's machine that are consumed or altered not only by the widgets but also by the application in general. They include both: high responsibility APIs like Google authorization, and the ones only accountable of providing data e.g. weather forecast API.

## 3.2 Desktop application diagrams

In order to provide a better understanding of how the system operates, how actions taken by the user affect the state, and in what scenarios desktop application communicates or consumes external resources, several diagrams will be presented. Starting with the general state flow of the application followed by in depth analysis of each of the identified states.

### 3.2.1 General state

Application is in large part built using a framework, which requires to write state-based logic (see 4.1.2), therefore presenting it at a high level of abstraction by using a state diagram is very helpful at a later implementation stage. State diagram of the application can be seen in figure 3.2.

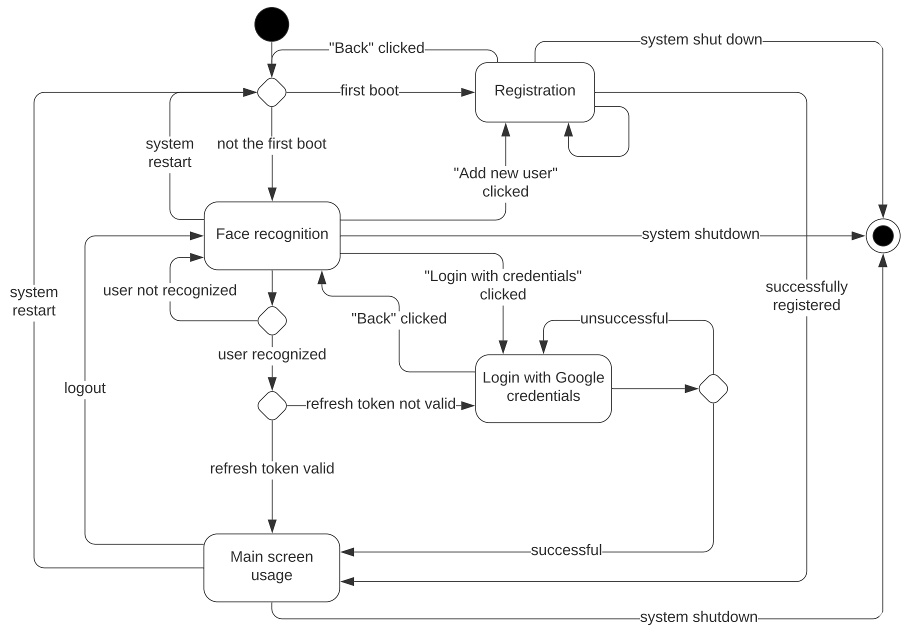


Figure 3.2: Desktop application state diagram

[1]Interprocess Communication - ways of message passing between different processes running on the same system [32]

[2]Hypertext Transfer Protocol Secure - internet communication protocol that protects the integrity and confidentiality of data between the user's computer and the resource [20]

State of the application changes when specific actions are taken by the user (e.g. logout action) or when certain conditions are met, which usually is an outcome of events happening while the application is in a particular state. In order to show what these events are, each of the states was presented by its own diagram in the appropriate of the following sections.

### 3.2.2 Registration

This is the state in which the user registration process takes place, and was presented using a combination of state and sequence diagrams. It consists of six stages, from which 'Face registration' and 'User authorization sequence' are the most important, as they require interaction with external resources. The first one exchanges information with the face recognition module and the second one with Google authorization API (the exact sequence of authorization can be seen in diagram 3.4 in the next section).

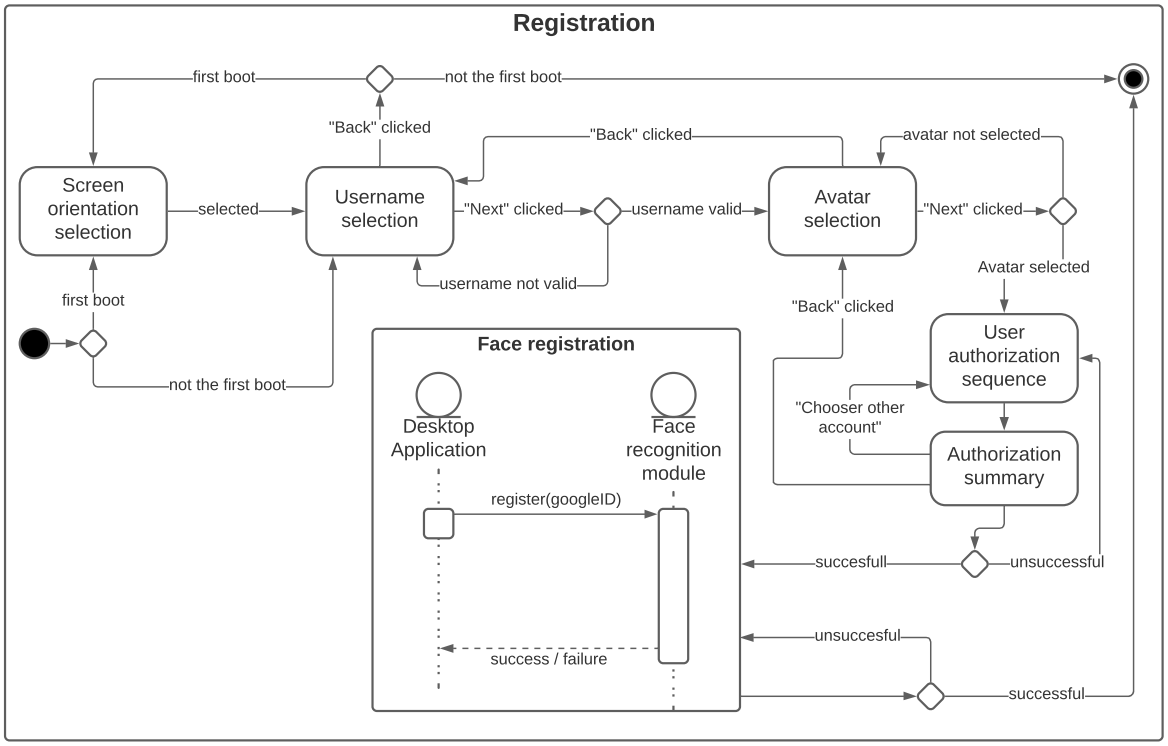


Figure 3.3: Registration flow diagram

### 3.2.3 User authorization sequence

In this state, the application goes through a series of events, one of which requires user interaction, namely authenticating with Google credentials. A Positive result of this sequence allows the Smart Mirror system to become synchronized with Google services, as after obtaining valid access and refresh tokens, private data can be fetched into Google-dependent widgets. It was presented with the use of sequence diagram in figure 3.4.

Whenever the application reaches the 'Login with Google credentials' state (see 3.2) it also uses the same sequence to authenticate the user, with the difference that it does not start automatically, it rather has to be invoked by the user interaction, as it was shown in the general state diagram 3.2.1.

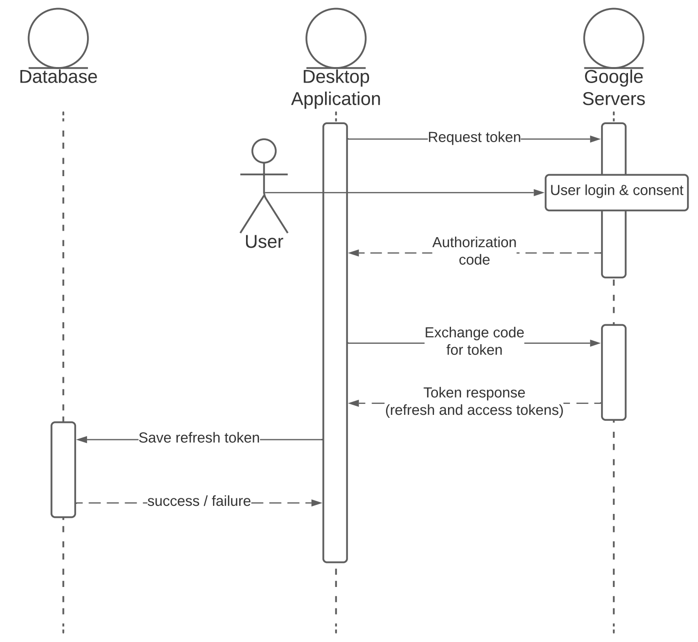


Figure 3.4: User authorization sequence diagram

### 3.2.4 Face recognition

As long as the user stays unrecognized and does not take any other actions (see 3.2) the application will remain in this state trying to recognize the face of the user standing in front of the mirror. However, the desktop application is not responsible for recognition itself, it informs the external face recognition module that the recognition procedure should be conducted and awaits its response (see diagram 3.5 below).

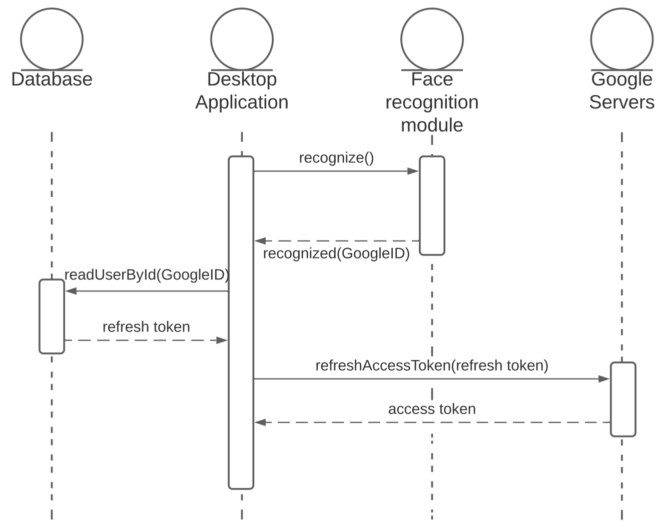


Figure 3.5: Face recognition sequence diagram

### 3.2.5 Main Screen usage

In this case, the application again can be treated as a state machine and can be in one of three states, presented in diagram 3.6. Depending on the current state the application interacts with different elements of the system's architecture. 'Widgets usage' state, which is the default one, requires the application to make requests to the APIs that currently displayed widgets are dependent on. In the 'Web browsing' state HTTPS protocol is used to view pages selected by the user. While 'Layout edition' mode needs access to the database, in case the user decides to save the newly created layout.

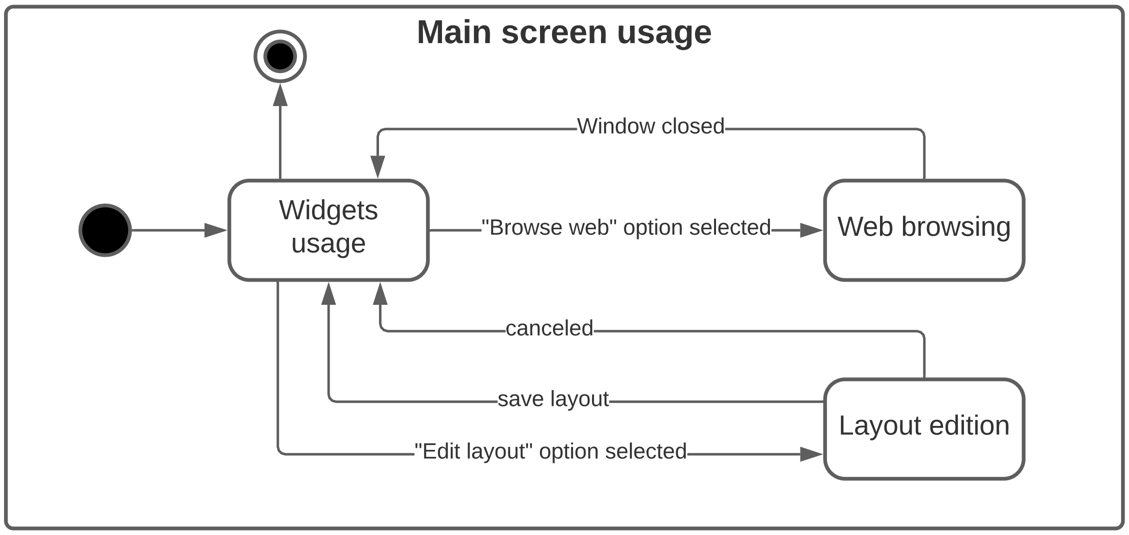


Figure 3.6: State diagram of the main application screen

## 3.3 Database design

During system requirements and behavior analysis, a data persistence mechanism appeared to be needed. However, the amount of data that has to be stored locally turned out to be relatively small in the current application state. It is because the desktop application depends on many external data sources, from which data is dynamically fetched and displayed inside of the widgets (see table 4.2 for more details on widgets). Also, it does not have to hold any data related to the face recognition, as all of that is handled by the other system component - the C++ module.

The application needs to hold information about each registered user and the global settings of the system. Users are not related to each other in any way and the global settings apply to all of them with no exceptions. Therefore non-relational database solution was chosen for the Smart Mirror system (see 4.1.3), as there were no relevant relations that could be distinguished.

The database consists of two separate collections:

- `users` - holds a single document for each user
- `global settings` - holds only one document with application's settings

The exact schemes of documents belonging to these collections can be found in tables 3.1 and 3.2 below.

| Data field name | Type | Description |
|---|---|---|
| `userName` | string | Username provided during registration (must be unique and at least 5 characters long, but no longer than 37) |
| `avatar` | string (URL) | Avatar chosen during registration (these are URLs to images generated by Dicebear Avatars API) |
| `layout` | array[obj] | Describes what is the set of visible widgets and how are they distributed on the screen (see 4.7.2 for exact layout array structure reference) |
| | `googleData` | |
| `tokens` | | |
| `refreshToken` | string | Allows to request new valid access token (see 4.4.2 for information on what is the purpose of the access token). It makes it possible to consume Google APIs when signing in to the desktop application via face recognition (see 4.5.1) |
| `scope` | string | Scope of resources to which user granted access (see 4.4 for more details) |
| `tokenType` | string | Type of the refresh token (used in Google APIs requests) |
| `userData` | | |
| `id` | string | Google ID uniquely identifying user of the system. Face recognition module uses it for binding face metadata to a particular user (see 3.5). Desktop application uses it to validate user authenticated via Google (see 4.5.2 for more details) |
| `verifiedEmail` | boolean | Informs whether the account that the user authenticated with has verified email |

Table 3.1: Contents of each document from `users` collection

| Data field name | Type | Description |
|---|---|---|
| `orientation` | string | Indicates what is the orientation of the mirror (possible values: `horizontal` / `vertical`). Has to be set during initial boot of the system (see 3.3). Can be changed anytime |

Table 3.2: Contents of the only document from `global settings` collection

# System implementation

This chapter describes the development phase of the Smart Mirror desktop application and its integration with all of the other elements that the system contains. It starts with a brief description of tools used for the implementation, then goes through the exact architecture of the application along with the rationale of taken approaches, most essential components implementation, and workarounds, which were a necessity resulting from the limitations of the technologies used.

This part contains many references to the source code of the application. Each path with the following format:

`/directory1/directory2/fileName.extension`

is relative to `smart-mirror` folder containing project source code (see appendix A).

## 4.1 Technologies

Description of used technologies and rationale of choices.

### 4.1.1 Electron

Electron is an open-source framework for building desktop applications using web technologies, i.e. HTML for the content and structure, CSS for the styling, and JavaScript language for functionality. It is not however limited to the technologies mentioned, in fact, it can utilize many modern JavaScript frameworks and libraries like Angular, Vue, or like in the case of the Smart Mirror system - React.js (see 4.1.2). This platform allows developers to maintain one JavaScript codebase and create cross-platform applications that may work on Windows, macOS, and Linux with minimal or no platform-specific code [6]. Originally released in July 2013 by GitHub engineer Cheng Zhao as a part of the effort to produce a new code editor, Atom [27, p. 2]. Initially, the project was known as the Atom Shell but was rebranded simply as Electron in April 2015 [31] and had its first official release in May 2016 [29]. Since December 2019 the project is hosted by the OpenJS Foundation [30] which also manages projects like jQuery, Node.js, or webpack [8].

The objective that Electron framework achieves by allowing the usage of web technologies is not only developer experience improvement but also wide portability and exposition of additional capabilities to the developers in the field of desktop development e.g. capability of implementing complex UIs with great UX. It does so by combining two existing technologies: Chromium and Node.js. Chromium is an open-source version of Google's Chrome web browser [27, p. 2]. It uses V8 as the JavaScript engine and Blink[1] as the layout engine [26, Chapter 5]. Chromium's core purpose is to render user interface, and it accomplishes this through retrieving and rendering HTML, loading and parsing CSS, and executing JavaScript as well [27, p. 2]. Access to operating system primitives, like the file system and shell, as well as a vast ecosystem of libraries is provided through the Node.js JavaScript runtime which is meant for running JavaScript code outside of the browser and therefore includes APIs for traditional OS features, and its corresponding package repository called NPM (Node Package Manager). On top of this Electron adds its own APIs for native platform functions like menus, dialogs, tray icons, etc.

The flexibility of the platform and the need of creating complex, fully custom UI with a high standard of UX, determined that Electron.js was used for the Smart Mirror desktop application implementation.

[1] Open-source layout engine for interpreting and laying out HTML [11]

### 4.1.2 React.js

React is an open-source JavaScript library for building composable user interfaces [17]. It encourages the creation of reusable UI components which present data that changes over time [28]. Originally a prototype of React was created by Jordan Walke, a software engineer at Facebook, and had its first public release in May 2013 [18]. As the most popular front-end technology it is used by large, established companies like Facebook, Netflix, Uber, Instagram [24].

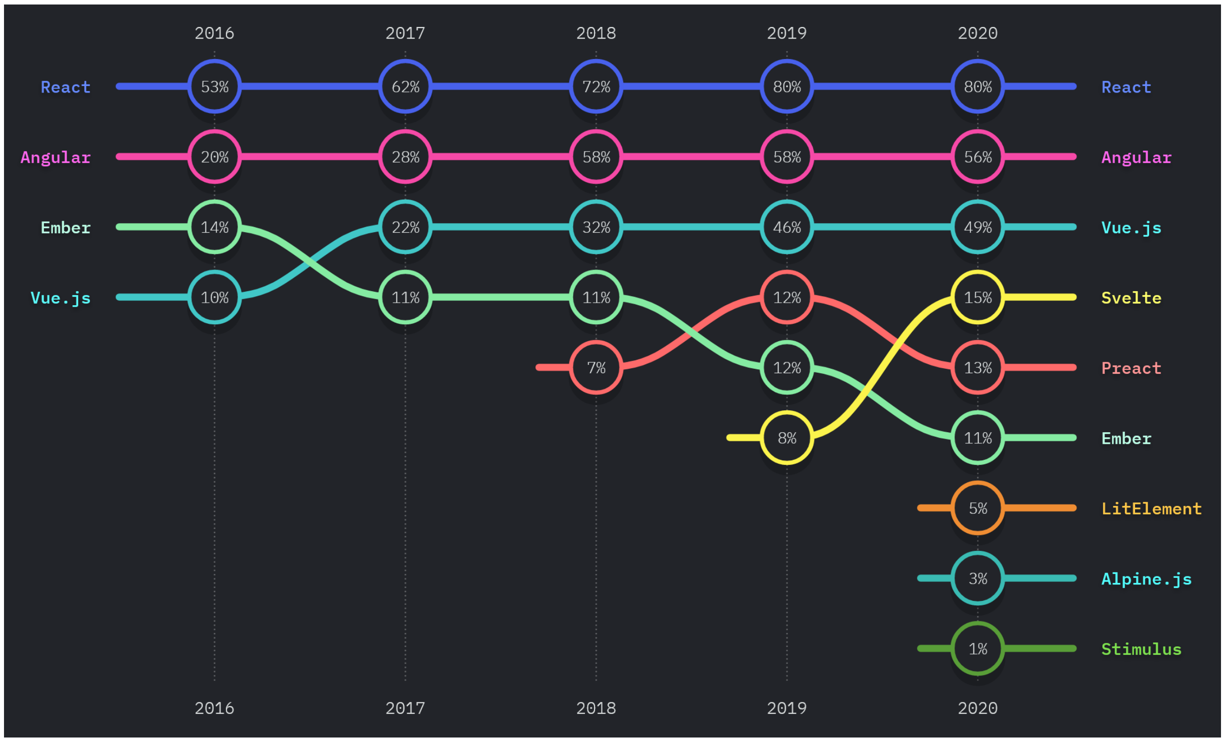
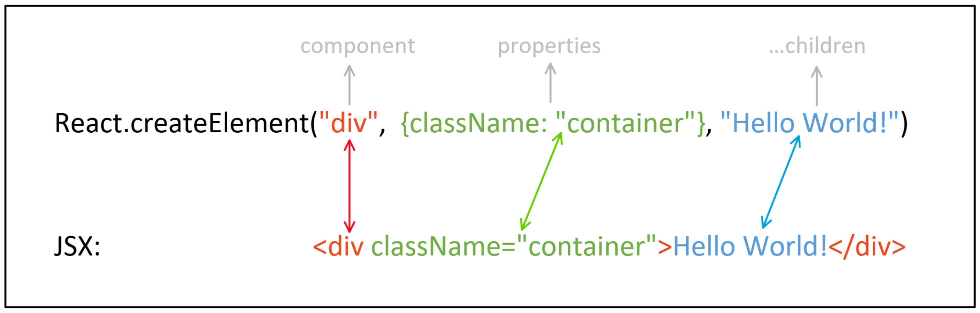


Figure 4.1: Front-end technologies usage ranking, source: [7]

Traditionally, web application UIs are built using templates or HTML directives [28] and go by the 'Separation of Concerns' approach. This means that all of the code related to a certain view is separated into concerns: content (HTML), styling (CSS), and functionality (JavaScript). React breaks this pattern, it unifies the markup with its corresponding view logic, by using a real, full-featured programming language (JavaScript) to render views [28]. Content of the view is then defined using `React.createElement` calls [19], but React also comes with an optional syntax extension called JSX (JavaScript XML) so that HTML-like code can be used to specify the content of the view instead of raw JavaScript (see figure 4.2) [28].

React also introduces superior to traditional JavaScript applications approach of updating the DOM. Instead of looking for data that changed and imperatively making changes to the DOM, it generates a lightweight representation of the component (the view). From that representation, a string of markup is produced and injected into the document [28]. Every time data changes, a new representation of the component is generated and then compared to its previous version. This comparison allows React to find the minimal set of changes that should be applied to the DOM, introducing a way to update it as efficiently as possible [28]. Combining this fact with previously mentioned markup and logic unification, many available libraries (e.g. MaterialUI[2]), and dedicated browser developer tools, React becomes the top choice, and that is why it was chosen for the Smart Mirror UI implementation.

[2]Library providing a set of prebuilt and highly customizable react components from which complex UIs can be easily built.



```
React.createElement("div", {className: "container"}, "Hello World!")

JSX:                <div className="container">Hello World!</div>
```

Figure 4.2: `React.createElement` call vs. JSX syntax

### 4.1.3 NeDB

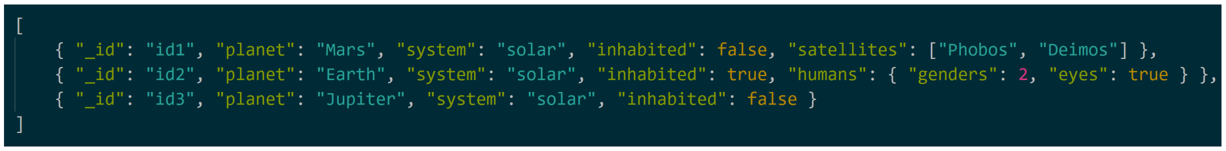
NeDB is a lightweight document-based DBMS[3] [4] written completely in JavaScript, with no binary dependencies [14]. It supports Node.js, Electron, web browser environments and uses an embedded database system approach - the database system is on the application when it is being run because it is imported to the application as a library [4]. Therefore it is installable as an NPM package.

Originally developed as an open-source project by Louis Chatriot since May 2013. In 2017 project was finished and since then was no longer maintained by the initial creator, however, several independent teams have forked the original NeDB repository to maintain and further develop features of the library[4] [4].

NeDB

Figure 4.3: Logo of the original NeDB package source:[13]

NeDB is not a relational database but rather a document store solution [4], meaning it does not consist of tables sharing certain relationships, but collections containing possibly nested documents, represented in JSON format (as shown in figure 4.4), similarly to MongoDB[5].



```
[
    { "_id": "id1", "planet": "Mars", "system": "solar", "inhabited": false, "satellites": ["Phobos", "Deimos"] },
    { "_id": "id2", "planet": "Earth", "system": "solar", "inhabited": true, "humans": { "genders": 2, "eyes": true } },
    { "_id": "id3", "planet": "Jupiter", "system": "solar", "inhabited": false }
]
```

Figure 4.4: Sample contents of the NeDB collection

NeDB stores all collections in memory in order to perform fast find and query operations, which causes it to only perform well on smaller datasets. Its' JavaScript API is a subset of MongoDB API. Therefore the syntax for the query commands is the same as MongoDB, which allows a smooth migration once the amount of data that needs to be stored exceeds the capabilities of NeDB [4]. This fact along with the lightness of this solution were crucial reasons to choose it, as in the current state Smart Mirror does not hold a large amount of data (see 3.3), it might in the future though (see 6.3).

[3]Database Management System
[4]The exact fork of the NeDB library that was used in Smart Mirror can be found here: [12]
[5]Non-relational, document-oriented database management system written in C++ [10]

## 4.2 Desktop application multi-process model

As mentioned in 3.1, one of the main parts of the Smart Mirror system is an Electron-based desktop application. Because Electron inherits its multi-process architecture from Chromium (see 4.1.1), this part of the Smart Mirror system is architecturally very similar to a modern web browser [16].

### 4.2.1 The multi-process model justification

In the earlier days, browsers usually operated using a single process for all of their functionalities. Although this pattern meant less overhead for each of the tabs the user had open, it also meant that one website crashing, hanging, or running into a fatal error would bring down the entire browser. To solve this problem, Chromium isolates tabs (windows) from each other and from the browser itself, by rendering each tab as a separate process. This approach limits the harmful or malicious influence a code on a web page can have on an application as a whole. There is also a single browser process that controls these processes, as well as the application lifecycle [16]. The diagram below visualizes the multi-process model of a web browser described above (see figure 4.5).

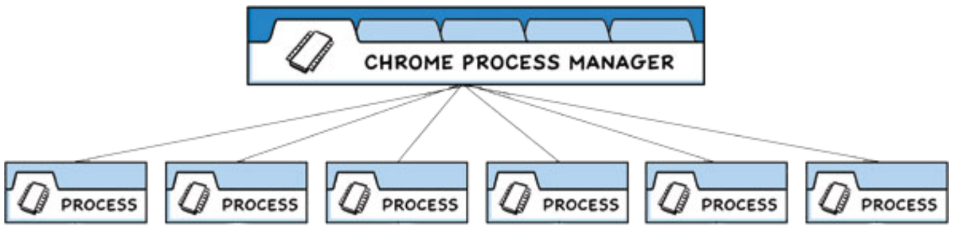


Figure 4.5: Chrome multi-process model source:[2]

The desktop application of the Smart Mirror system is structured in the same manner. It runs across multiple processes, namely the main and renderer processes [27, p. 6]. Its exact model can be seen in figure 4.6 below.

### 4.2.2 The main process

The application consists of a single main process (see figure 4.6), which `main.ts` script acts as the application's entry point (see `/src/main/main.ts`). It runs in a Node.js environment, meaning it can `require` node modules (modules from NPM repository, especially `electron`[6]) and use all of the built-in Node.js APIs (e.g. `path`[7]) [16]. One of the primary responsibilities of the main process is creating and managing application windows. It does so by using `BrowserWindow` [16], which is a sub-module of the `electron` package. Each instance of the `BrowserWindow` class creates an application window as a separate renderer process [16] (see `/src/main/main.ts`, lines 36, 66 and 162 for `BrowserWindow` usage reference). Because the main process creates and controls these instances, it has access to all of the renderer processes, which are isolated and can control themselves only [1]. When a `BrowserWindow` instance is destroyed, its corresponding renderer process gets terminated as well [16].

### 4.2.3 The renderer processes

While the main process has access to Node.js and Electron.js APIs, it cannot access DOM APIs [5] (see figure 4.6), those are only available to renderer processes, which simply are Chromium browser instances, responsible for - as the name implies - displaying UI. Currently, the application allows to concurrently

[6]Electron applications are essentially just Node.js applications with the electron package installed as a dependency [25]
[7]Node.js built in module providing useful functions to interact with file paths

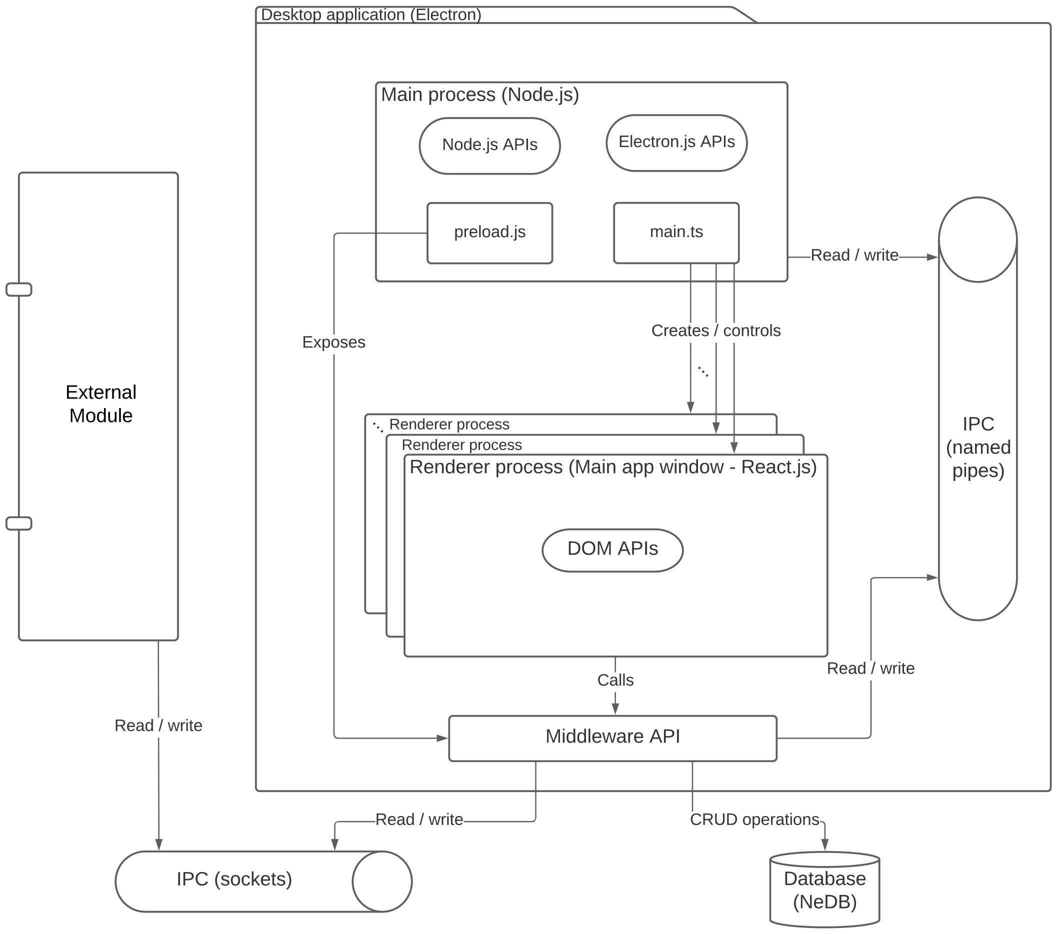


Figure 4.6: Smart Mirror multi-process architecture

run maximum of two renderer processes, one of which is always the main application window[8], and the second one is Chromium instance with certain URL loaded - either authorization URL for signing in with Google (see 4.4.1) or any URL chosen by the user while browsing the web (see 4.7.1).

Code ran in a renderer process should behave according to the web standards, therefore it had to be written with the same tools and paradigms that are used on the web [16]. This means, that in order to use React.js (see 4.1.2) in the UI implementation of the main application window, a bundler toolchain had to be used - in this case it was webpack[9] from OpenJS Foundation. The reason for this is the fact that any renderer process by default cannot consume modules (this is a security concern, see 4.2.4). It means that actions that require the presence of a specific module, like querying a database or communicating with the main or external (e.g. face recognition module) processes through IPC are impossible from the renderer process level. Even though this default setting could be disabled for the main application window, and at the same time it would not expose the application to any security dangers (main application window in contrary to other renderer processes does not use the HTTPS protocol to load its content - it loads fully trusted local HTML file using file protocol), it is still not encouraged nor considered a best practice. Importing Node.js or Electron.js modules directly in the main application window would be an architectural vulnerability, as it would decrease the number of potential features that could be implemented in the future (see 6.3), which is against the 'Open-closed' principle. For example,

[8]The most important renderer process, which uses React.js to display all of the custom UI (see figure 4.6)
[9]Static module bundler for modern JavaScript applications [23]

executing code from remote server or embedding websites inside widgets would require refactoring almost the whole main application window in order to keep it secure.

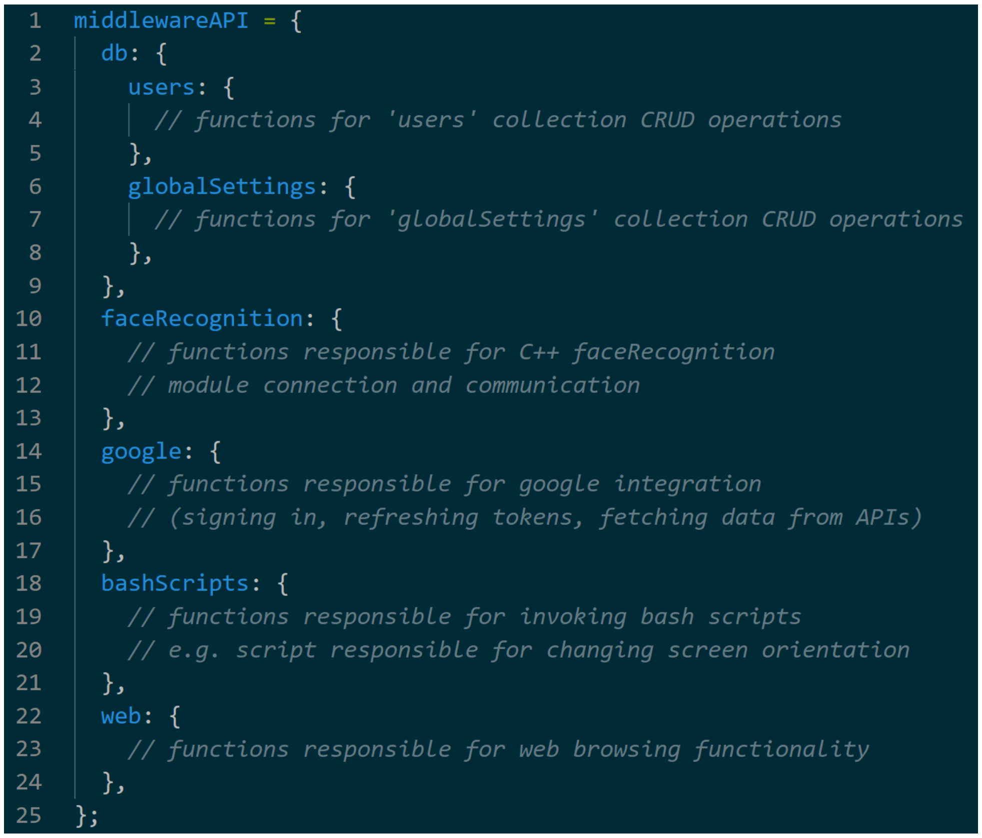
Taking this into consideration, the direct access to modules was not granted to any renderer process in the application, however main application window capabilities are enhanced by consuming Middleware API (see 4.2.5).

### 4.2.4 Renderer process isolation

A security issue exists whenever the renderer process executes code received from a remote source locally (e.g. from a remote server) [21]. Which is a case in the Smart Mirror desktop application, as it allows users to browse websites (see 4.7.1). If an attacker somehow managed to change received content (either by attacking the source directly or by sitting between the Electron application and the destination, from which content was requested), they would be able to execute native code on the user's machine [21], by accessing Node.js modules (e.g. they could delete files using `fs` module). That is why `nodeIntegration` is disabled for all renderer processes, meaning they do not have a direct access to `require` or any other Node.js API. Thanks to this potential attacker is not able to harm user's machine.

### 4.2.5 Middleware API

Essentially Middleware API is just a JavaScript object composed of functions that contain most of the application's logic (see `/src/middleware/index.js` for full source code reference). These functions



```
middlewareAPI = {
  db: {
    users: {
      // functions for 'users' collection CRUD operations
    },
    globalSettings: {
      // functions for 'globalSettings' collection CRUD operations
    },
  },
  faceRecognition: {
    // functions responsible for C++ faceRecognition
    // module connection and communication
  },
  google: {
    // functions responsible for google integration
    // (signing in, refreshing tokens, fetching data from APIs)
  },
  bashScripts: {
    // functions responsible for invoking bash scripts
    // e.g. script responsible for changing screen orientation
  },
  web: {
    // functions responsible for web browsing functionality
  },
};
```

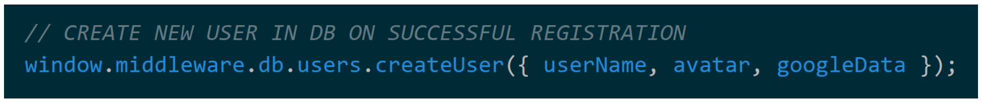
Figure 4.7: Middleware API object structure. Each of its sub-objects has certain responsibility (so-called responsibility slices)

can be called from the main application window (see figure 4.6), but at the same time are granted more privileges by having access to Node.js APIs [16] and the ability to `require` modules, which makes them capable of e.g. fetching data from the database, communicating with the external face recognition module or using IPC to inform the main process that new renderer process should be created (see figure 4.7 for structure reference).

As shown in figure 4.6 Middleware API is exposed to the main application window through `preload.js` script (see `/src/main/preload.js`), which bounds it to a global `window` object of this renderer process, so that all of the Middleware API functions can be referenced as follows:

```
window.middleware.responsibility_slice.specific_function(args)
```

Thanks to the Middleware API approach higher security standard is guaranteed no matter what features will be implemented, as none of the renderer processes directly `require` modules (see 4.2.4). Also, the code complexity of the main application window renderer process is much lower - to perform extensive operations React-based code simply calls Middleware API functions from appropriate responsibility slices.



```
// CREATE NEW USER IN DB ON SUCCESSFUL REGISTRATION
window.middleware.db.users.createUser({ userName, avatar, googleData });
```

Figure 4.8: Middleware API example call
(see `/src/renderer/components/register/registerSuccessScreen/RegisterSuccessfulScreen.jsx`)

## 4.3 Multi-step registration component

Before gaining access to the full potential of the Smart Mirror application, every user is obligated to register. Registration flow consists of a maximum of five user engaging steps (see diagram 3.3), including mandatory integration with Google account (see 4.4). This obligatory nature comes from the fact that some of the application widgets are based on Google services (see 4.7.3). From the user perspective, steps of the registration flow are as follows:

1. Choosing a username
2. Choosing an avatar
3. Google authentication
4. Face scanning

When starting the system for the very first time the user is automatically redirected to the registration flow (see diagram 3.2) enriched by one additional step, in which screen orientation can be chosen (see figure 4.9). During application booting, the top scope `<App />` component (see `/src/renderer/App.jsx`) of the main application window accesses `globalSettings` database collection (see table 3.2) to indicate whether this is the first boot. If `globalSettings` collection is empty, then screen orientation was never set, meaning no user has registered successfully in this system instance. That is how the system determines what set of register steps should be presented to the user. For each step of the registration flow, there is a corresponding React component implemented (see `/src/renderer/components/register`), which displays predefined UI elements and manages its own state. These components are managed and supervised by stateful variables of one parent component common for all of them - `<Register />`.

In order to successfully register, all steps of the flow must be performed. The first step requires the user to choose a unique username, which accessibility is validated with database contents via Middleware API (see 4.2.5), before moving on to the next step. In the second step, the user chooses his avatar, which in conjunction with the username will serve as an indicator of who is currently logged in (see table 4.2). Afterward, the user authorization sequence begins (see 4.4.1), where the user authenticates using Google account credentials. If authorization is successful, the user will be presented with a summary (see

'Sync with Google' in figure 4.9). In the last step `window.middleware.faceRecognition(googleID)` is called, to send `REGISTER googleID` command to the face recognition module (see table 4.1). The external module then executes the registration procedure simultaneously informing the application about progress, which is reflected in the UI.

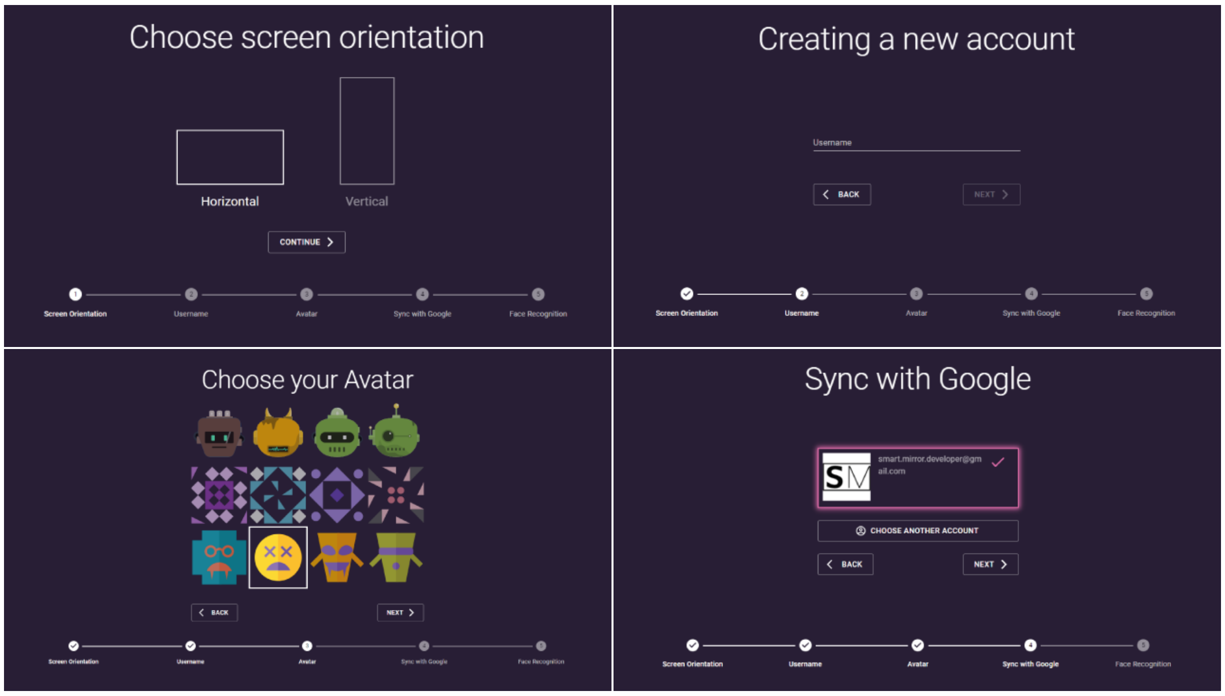


Figure 4.9: Registration process visual reference

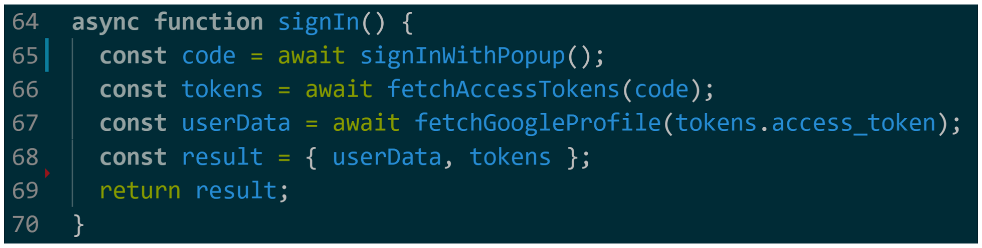
Successful registration results in saving data described in 'Database design' 3.3 in the database, and redirecting the user to the main screen of the application (see 4.7) as it was shown in the state diagram 3.2.

## 4.4 Google integration

As mentioned in the introduction, the idea of the Smart Mirror project is to create software that runs on a specific device (see chapter 5), making it join the family of smart home devices. That is why the most crucial functionalities of the system were based on Google services, which requires an authorized user to make requests. Therefore the user must authenticate with the use of a Google account, as it not only allows the system to be integrated with other devices by consuming common resources but also provides a higher probability that the user is real.

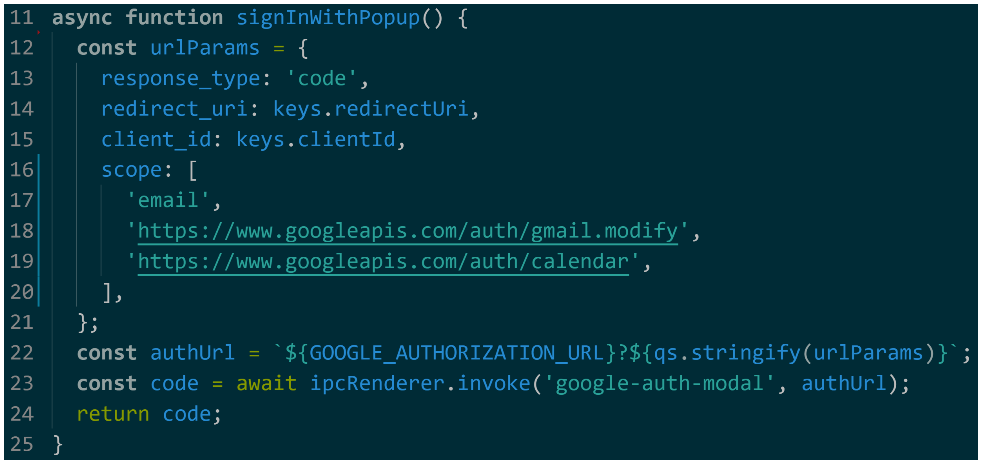
### 4.4.1 User authorization sequence

The authorization sequence begins when the main application window calls the following Middleware API function: `window.middleware.google.signIn()` (see figure 4.10). Then `signInWithPopup()` function with the use of IPC (see 4.6) informs the main process that a new renderer process should be created. As soon as the main process receives this information, it creates a new object of `BrowserWindow` class (see 4.2.2), which results in displaying a modal window. The newly opened chromium instance allows user to authenticate by automatically loading Google authorization URL. The URL, among other values, includes query parameters that indicate the type of access being requested from the user (see figure 4.11) [22, 'Installed apps']. Content of the 'Google sign in' modal window is presented in figure 4.12.



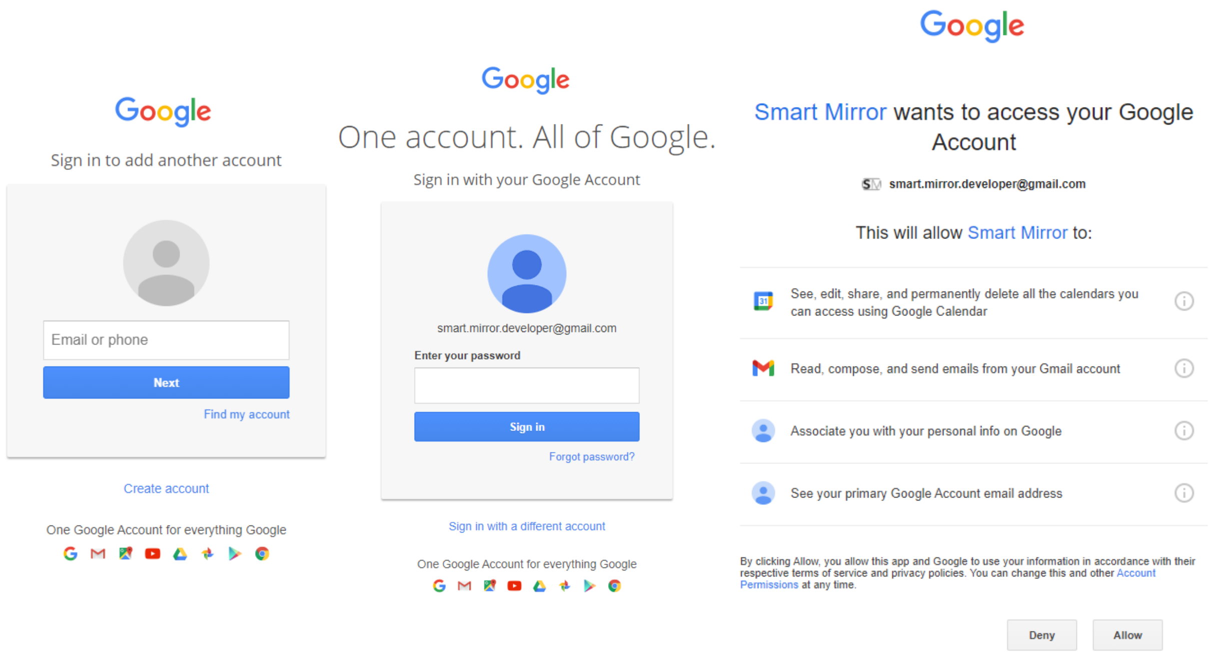
```
async function signIn() {
  const code = await signInWithPopup();
  const tokens = await fetchAccessTokens(code);
  const userData = await fetchGoogleProfile(tokens.access_token);
  const result = { userData, tokens };
  return result;
}
```

Figure 4.10: Function responsible for authorization sequence (for complete source code see `/src/middleware/google/signIn.js`)



```
async function signInWithPopup() {
  const urlParams = {
    response_type: 'code',
    redirect_uri: keys.redirectUri,
    client_id: keys.clientId,
    scope: [
      'email',
      'https://www.googleapis.com/auth/gmail.modify',
      'https://www.googleapis.com/auth/calendar',
    ],
  };
  const authUrl = `${GOOGLE_AUTHORIZATION_URL}?${qs.stringify(urlParams)}`;
  const code = await ipcRenderer.invoke('google-auth-modal', authUrl);
  return code;
}
```

Figure 4.11: `scope` parameter describes what permissions should be granted by the user



Google

Sign in to add another account

Email or phone

Next

Find my account

Create account

One Google Account for everything Google

Google

One account. All of Google.

Sign in with your Google Account

smart.mirror.developer@gmail.com

Enter your password

Sign in

Forgot password?

Sign in with a different account

One Google Account for everything Google

Google

Smart Mirror wants to access your Google Account

smart.mirror.developer@gmail.com

This will allow Smart Mirror to:

See, edit, share, and permanently delete all the calendars you can access using Google Calendar

Read, compose, and send emails from your Gmail account

Associate you with your personal info on Google

See your primary Google Account email address

By clicking Allow, you allow this app and Google to use your information in accordance with their respective terms of service and privacy policies. You can change this and other Account Permissions at any time.

Deny Allow

Figure 4.12: Modal window for Google authentication

Google handles the user authentication and user consent (see Figure 4.12) - the user is asked whether they are willing to grant all required permissions that the application is requesting [22]. After successful completion of the process[10] the application retrieves an authorization code which is immediately exchanged for access and refresh tokens (see Figure 4.10 line 66).

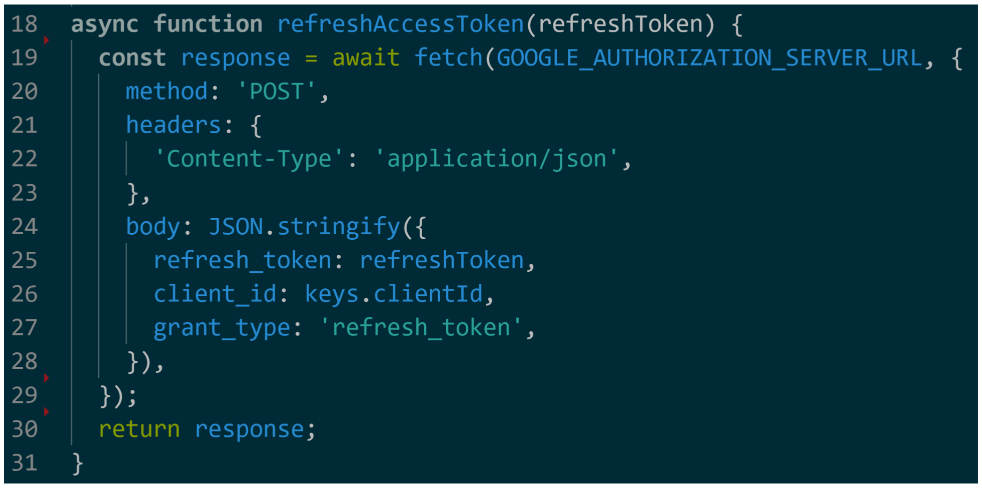
The last step of the authorization sequence is saving the received refresh token in the database, which might happen at various times depending on the authorization context (e.g. it will not happen immediately after authentication during registering, as it is required to successfully complete all steps of the registration flow before the user is saved in the database - see 4.3). Thanks to this mechanism, the user does not have to authenticate with Google credentials every time in order to obtain a valid access token (see 4.4.2 and 4.4.3).

### 4.4.2 Making requests to Google APIs on users behalf

In order to access private data using Google APIs, the application must obtain an access token [22, 'Obtain an access token from Google Authorization Server'], which is a credential that attached to an API request allows to do so. It is important that a single access token can grant varying degrees of access to multiple APIs. A variable parameter called `scope` controls the set of resources and operations that an access token permits [22] and is defined in the initial access-token request (see 4.11, line 16). Access tokens periodically expire and become invalid credentials for making Google API requests. However, it is possible to refresh an access token without prompting the user for permission [15] (see 4.4.3).

### 4.4.3 Refreshing an access token

Whenever the user logs in to the application, a new access token has to be obtained (see 4.4.2). This can be achieved by sending a `POST` request to the appropriate Google API (see Figure 4.13).



```
async function refreshAccessToken(refreshToken) {
  const response = await fetch(GOOGLE_AUTHORIZATION_SERVER_URL, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      refresh_token: refreshToken,
      client_id: keys.clientId,
      grant_type: 'refresh_token',
    }),
  });
  return response;
}
```

Figure 4.13: Function responsible for refreshing an access token (for full source code reference see `/src/middleware/google/refresh.js`)

A body of the request must contain a refresh token which can be found in the database (see table 3.3). As long as the refresh token is valid, the Google authorization service response will contain a new access token, which will serve as a valid credential for a limited time[11].

[10] In current project state it is considered successful only when user gives permission to all of the Google scopes (see 6.2)
[11] In current application state access token is valid only for an hour, however this can be improved (see 6.3)

## 4.5 Signing in

Users of the Smart Mirror application can log in to the system either by using the default face recognition approach or by authenticating with their Google account credentials. While both ways require an internet connection in order to fetch a valid access token (see 4.4.3) the first one requires also a web camera.

### 4.5.1 Using face recognition

As shown in the application state diagram (see diagram 3.2) when no user is signed in, the application by default redirects to the face recognition state, which is equivalent to displaying a `<LoginScreen />` React component (see figure below for visual reference).

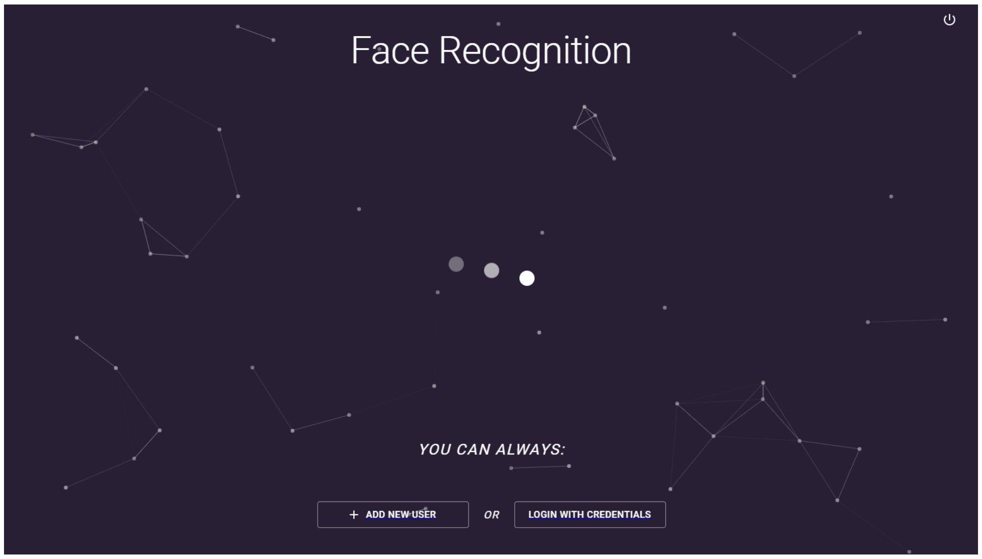


Figure 4.14: Login screen visual reference (see `/src/renderer/components/loginScreen/LoginScreen.jsx`)

Whenever this component is mounted on the screen, it informs the face recognition module (see figure 4.6) that attempts to recognize the user should be made. It does so by invoking:

`window.middleware.faceRecognition.recognize()`

which is a middleware API function from the `faceRecognition` responsibility slice (see 4.7). It sends `RECOGNIZE` command to the C++ module via IPC (see 4.6). On successful recognition, the application (also with the use of IPC 4.1) receives a Google ID of a recognized user, which is used to retrieve his refresh token from the database, as a new access token has to be obtained (see 4.4.3).

However, there are scenarios, in which the refresh token is not valid anymore, and these are as follows [22, 'Refresh token expiration']:

- The user has revoked application's access
- The refresh token has not been used for six months
- The user changed passwords and the refresh token contains Gmail scopes (this is the case, see `scope` in figure 4.11)
- The user account has exceeded a maximum number of granted (live) refresh tokens
- The user belongs to a Google Cloud Platform organization that has session control policies in effect

In these cases new, valid refresh and access tokens have to be obtained, which can be done by once more authenticating the user with Google account credentials. That is why the user will be redirected to the appropriate screen to do so (see 4.5.2).

### 4.5.2 With Google credentials

There are only two scenario types in which user would have to interact with `<LoginWithCredentials />` component (see `/src/renderer/components/loginWithCredentials/LoginWithCredentials.jsx`), which allows to login to the application using Google account credentials instead of face recognition. They are as follows:

- Technical or external environment issues (e.g. broken web camera, very poor lighting, etc.)
- Log in with face recognition attempt was not successful due to Google authorization service response containing errors (especially invalid or expired refresh token cases)

For the first type scenarios, the user needs to manually navigate to the 'Login with credentials' screen (see figure 4.14 for visual reference), while for the rest it will happen automatically. After clicking the 'Sign in' button, authorization sequence 4.4.1 begins. When it is successfully completed

```
window.middleware.db.users.updateUsersRefreshToken()
```

is called, in order to replace the old refresh token in the database with its new value.

From the fact that authentication is completely handled by Google, the user might try to log in using a Google account that was not registered in the Smart Mirror application yet, so there is a need for a users validation mechanism. During registration, along with other values (see `userData` in figure 4.10), the Google ID of the user is fetched (see `/src/middleware/google/profile.js`) and saved in the database. It allows to uniquely identify a user in both: Google services and Smart Mirror application. This means that mentioned user validation comes down to a simple check whether the ID of the user that has just authenticated is assigned to any of the existing documents in the `users` collection (see `googleData` in table 3.1). If it is not, then the user will be prompted that his account was not registered yet and a registration proposal will be displayed (see `/src/renderer/components/loginWithCredentials/LoginWithCredentials.jsx` for exact logic implementation).

## 4.6 Face recognition module integration

Face recognition module is an external, standalone application written in C++ language, which allows to register and then recognize users by analyzing face images retrieved from the web camera. It was built as a part of the Smart Mirror system (see 'System architecture' 3.1), however, its implementation is beyond the scope of this thesis. It is consumed by the Electron application which utilizes IPC protocol for this purpose, namely sockets.

During booting, the Electron application establishes a connection by finding two available ports and launching the C++ module with obtained port numbers passed as parameters (see `/src/middleware/faceRecognition/launcher/launcher.js`). One of the ports serves as a sender and the other one as a receiver, meaning the application can both: send commands to the module and receive its feedback. This two-directional data flow approach is required so that the application can inform the module that certain actions should be taken and react accordingly to the C++ module's responses (e.g. showing registration progress in the UI, finishing successful registration flow, logging in recognized user).

Feedback received from the module is firstly tokenized (for tokenizing function implementation reference see `/src/middleware/faceRecognition/tokenizer.js`) into command and payload, and then fully handled by `faceRecognition` responsibility slice (see figure 4.7), which includes not only functions responsible for communication but also connection establishment (see `/src/middleware/faceRecognition/index.js`).

String commands presented in table 4.1 can be exchanged between two system components (face recognition module and desktop application) under the condition of successful connection establishment.

| Command | Functionality |
|---|---|
| From Electron to face recognition module | |
| `REGISTER <userID>` | Tells the module to switch to registering mode and try to register user identified by the given `<userID>` |
| `RECOGNIZE` | Tells the module to begin recognition procedure |
| `SLEEP` | Tells the module to switch to "paused" state |
| `WAKEUP` | Tells the module to go back to work |
| `STOP` | Shuts down the module completely (exits the application) |
| From face recognition module to Electron | |
| `PROGRESS <percentage value>` | Continuously sent to the Electron application during registration process |
| `RECOGNIZED <userID>` | Sent to the Electron application after successful recognition, `<userID>` is ID of recognized user |
| `REGISTERED <userID>` | Sent to the Electron application after successful registration, `<userID>` is ID of registered user |
| `ERROR <message>` | Sent to the Electron application if an error occurred, `<message>` contains additional error information |
| `FATAL <message> <exitCode>` | Sent to the Electron application if a fatal error occurred, `<message>` contains additional error information, `<exitCode>` contains module's exit code |

Table 4.1: Face recognition module and desktop application communication protocol (see also `/src/middleware/faceRecognition/commands.js`)

## 4.7 Main screen

Main screen is the most important piece of UI displayed in the Smart Mirror desktop application, as it provides access to all functionalities mentioned in chapter 2 in a form of widgets (see 4.7.3) or separate renderer processes (see 4.7.1). However valid access token is required in order to display it, as some of the accessible widgets depend on private data from Google APIs, meaning only authenticated users can reach this screen[12] (see diagram 3.2).

Similar to other complex screens of the desktop application, to implement a React component for this part of the UI, many sub-components had to be distinguished, implemented, and enclosed in one parent component. For this purpose `<MainScreen />` was implemented (for full source code reference see `/src/components/mainScreen/MainScreen.jsx`). Its two core sub-components are:

- `<TopBar />` - serves as a holder for additional options of the main screen (see figure 4.15 for visual reference)
- `<GridLayout />` - responsible for containing widgets arranged in a certain configuration (see 4.7.2)

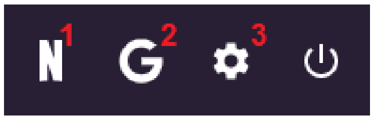


Figure 4.15: Top bar options (`/src/renderer/components/mainScreen/TopBar.jsx`)

[12]In the current application state, the user account has to be integrated with their Google account even if they do not wish to use Google dependant widgets (see 6.2)

Main screen is capable of operating in two distinct modes:

- Widgets usage [default] (see 4.7.3)
- Layout edition (see 4.7.2)

Current mode is determined by one of the state variables of the `<MainScreen />` component.

### 4.7.1 Web browsing

One of the functionalities provided by the Smart Mirror application is web browsing. It has been implemented as a response to users' needs, as even though by registering the user gains access to all widget-related functionalities (see 4.7.3) there is sometimes a need to find very specific information or perform steps that could not be predicted by the pre-built UI of the application. That is why each user can view any website.

Since the application was built with the use of Electron.js framework (see 4.1.1) there was no need to use a third-party web browser for this purpose, as each renderer process is actually a Chromium instance (see 4.2.3), which can be used as a normal web browser by simply loading remote content via HTTPS protocol.

In order to begin web browsing, from the top bar options, the Google icon has to be selected (see figure 4.15 option 2). When it happens the following function is called:

```
window.middleware.web.openBrowser()
```

which comes from the `web` responsibility slice of the middleware API (see figure 4.7). Similarly to opening Google modal window in the user authorization sequence (see 4.4.1) the main process is informed via IPC that a new renderer process should be created, and it does so by again creating new `BrowserWindow` instance and immediately calling its `loadUrl` method with Google search engine URL passed as a parameter. From now on the user can browse web pages as he would normally do in a standard web browser. There is also the option of selecting the Netflix icon (see figure 4.15 option 1) which causes the same course of events, with one major difference, namely URL loaded in the Chromium instance simply points to the Netflix web application.

It is worth noticing that because web browsing is handled by a renderer process separate from the main application window process (see figure 4.6), even if browsed website crashes or user encounters malicious code it will not affect or harm the application as a whole, as these processes are independent and isolated from each other (see 4.2.1).

### 4.7.2 Layout

Layout describes what widgets are visible and where are they placed on the main screen. It has a form of three by three grid, as during AB testing of the layout prototypes, displaying more than 9 highly engaging widgets turned out to be too overwhelming for the users. It is represented by an array of objects in the following manner:

```
const layout = [
  { i: 'clock', x: 0, y: 0, w: 1, h: 1, isResizable: false },
  { i: 'weather', x: 2, y: 0, w: 1, h: 1, isResizable: false },
  { i: 'news', x: 1, y: 1, w: 1, h: 1, isResizable: false },
  { i: 'calendar', x: 0, y: 2, w: 1, h: 1, isResizable: false },
  { i: 'radio', x: 1, y: 2, w: 1, h: 1, isResizable: false },
  { i: 'mailbox', x: 2, y: 2, w: 1, h: 1, isResizable: false },
];
```

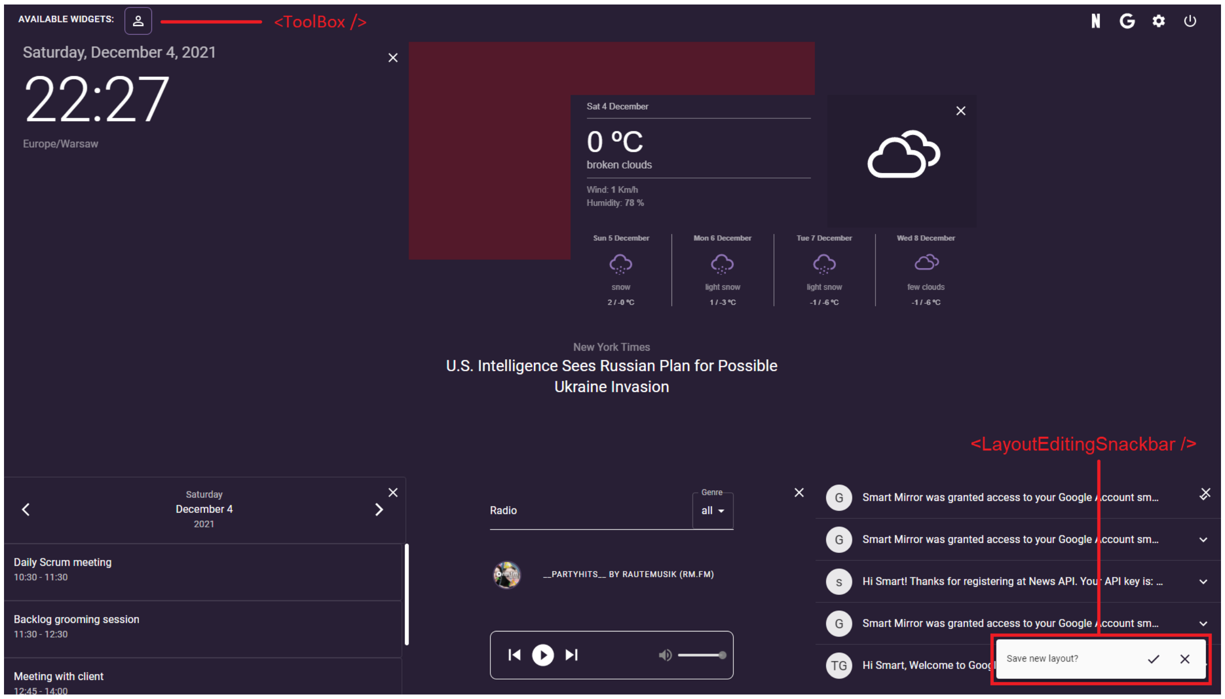
Figure 4.16: Default layout of the main screen (`/src/middleware/database/DEFAULTS/defaultLayout.js`)

where `i` field refers to an id of a widget, `(x, y)` pair means its placement, and `(w, h)` pair describes its horizontal and vertical spanning respectively. This form of representation is a requirement of `React Grid Layout`[13] which was the main pillar of the layout feature implementation.

The layout can be modified, and in order to do so, the main screen has to operate in 'Layout edition' mode (see 4.7), which can be achieved by selecting the `Edit layout` option from the settings drawer (see `/src/renderer/components/mainScreen/SettingsDrawer.jsx`), accessible under the gear icon (see figure 4.15 option 3). Users can perform the following activities to modify the layout:

- Change the set of displayed widgets
- Alter the position of already visible widgets

Position altering is done by 'drag and drop'[14], in a very similar way as on modern mobile phones (see Figure 4.17 for visual reference).



U.S. Intelligence Sees Russian Plan for Possible Ukraine Invasion

Daily Scrum meeting
10:30 - 11:30

Backlog grooming session
11:30 - 12:30

Meeting with client
12:45 - 14:00

G Smart Mirror was granted access to your Google Account sm...
G Smart Mirror was granted access to your Google Account sm...
S Hi Smart! Thanks for registering at News API. Your API key is: ...
G Smart Mirror was granted access to your Google Account sm...
TG Hi Smart, Welcome to Goog

Figure 4.17: Main screen in 'Layout edition' mode, during weather widget dragging

To add a widget to the layout (make a widget visible) appropriate icon has to be clicked from 'available widgets' toolbox (see `/src/renderer/components/mainScreen/ToolBox.jsx`). And for the sake of removing widgets from the layout, an 'X' icon inside a particular widget has to be clicked (see figure 4.17). Whenever the set of displayed widgets gets updated, both: state variable controlling toolbox contents and state variable holding current layout are modified accordingly (see `onTakeItem()` and `onPutItem()` functions implementation[15], as they are responsible for this logic), however, when the widget is moved only layout state variable is updated. Users can discard any changes they have made to the layout at any time while being in 'Layout edition' mode, by simply clicking an 'X' inside `<LayoutEditingSnackBar />` (see figure 4.17). This is possible because before entering 'Layout edition' mode current layout is memoized, so it can be restored later on. If the user decides to rather save the layout, the following function is called:

```
window.middleware.db.users.updateUsersLayout(userId, layout)
```

[13]Library providing grid layout system for React-based applications
[14]'Drag and drop' functionality was not implemented from scratch, it was one of the React Grid Library features
[15]These functions can be found in the following file: `/src/renderer/components/mainScreen/MainScreen.jsx`

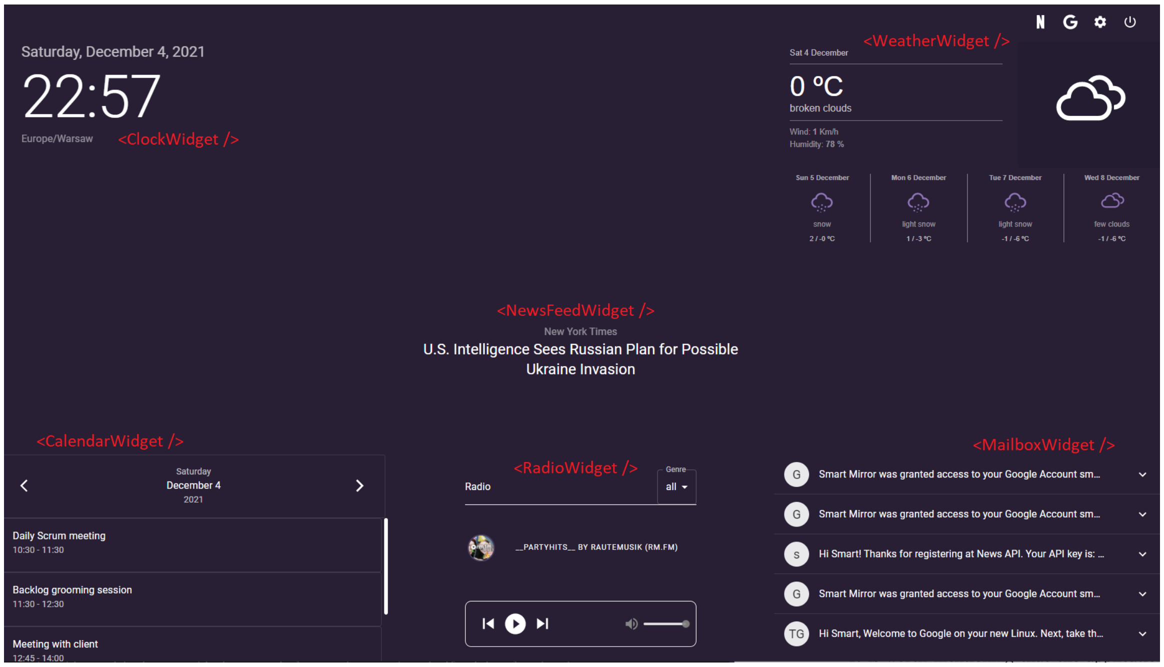
which is a Middleware API function, from `db` responsibility slice, that updates the value of `layout` field in the user document with the given id. The only way to exit 'Layout edition' mode is to either save or discard the newly created layout.

Whenever the user logs in, his personalized layout is restored by being fetched from the database and assigned as an initial value to the state variable holding the current layout of the main screen. Also the initial content of 'available widgets' toolbox has to be computed by comparing user's restored layout with a collection of all available widgets (see `computeToolBoxItems()` function from `/src/renderer/components/mainScreen/MainScreen.jsx`).

Two breaking issues occurred during layout functionality implementation, namely, it turned out that the `React Grid Layout` library is not swapping items during horizontal drag-over, rather it moves the items into a new row. This means that in a case where a widget was dragged into a column with all rows already occupied, a new row was added for the dragged widget, creating a vertical overflow. In order to fix this issue `fixLayout()` function was implemented (see `/src/renderer/components/mainScreen/MainScreen.jsx`). It detects whether the mentioned violation occurs and fixes the layout by iterating through coordinate combinations until it finds the first (x, y) pair that is not occupied, and then places there the overflowing widget.

The other issue was related to the responsiveness[16] of the application. `React Grid Layout` library only supports setting the height of rows in pixels. Meaning that it is not possible to set the height of the row to 1/3 of the current screen height (which is a must in this case), as both % and vh CSS units are not supported. This also meant that the main screen grid layout would not adjust after changing the screen orientation (which is a special case of resizing a window). In order to fix this issue `useWindowDimensions` function is called (see `/src/renderer/hooks/useWindowDimensions.js`), which always returns current screen dimensions in pixel units, and causes the main screen grid to rerender by setting its row height property to the current screen height divided by 3, so the grid layout always fills the whole screen, making it fully responsive.

## 4.7.3 Widgets



Saturday, December 4, 2021
22:57
Europe/Warsaw

Sat 4 December
0 °C
broken clouds
Wind: 1 Km/h
Humidity: 78 %

Sun 5 December snow 2 / -0 °C
Mon 6 December light snow 1 / -3 °C
Tue 7 December light snow -1 / -6 °C
Wed 8 December few clouds -1 / -6 °C

New York Times
U.S. Intelligence Sees Russian Plan for Possible Ukraine Invasion

Saturday
December 4
2021

Daily Scrum meeting
10:30 - 11:30
Backlog grooming session
11:30 - 12:30
Meeting with client
12:45 - 14:00

__PARTYHITS__ BY RAUTEMUSIK (RM.FM)

G Smart Mirror was granted access to your Google Account sm...
G Smart Mirror was granted access to your Google Account sm...
S Hi Smart! Thanks for registering at News API. Your API key is: ...
G Smart Mirror was granted access to your Google Account sm...
TG Hi Smart, Welcome to Google on your new Linux. Next, take th...

Figure 4.18: Accessible widgets

[16]Adjusting to the screen size and ratio

Widgets are compact elements of the desktop application user interface that allow users to access information or perform defined actions. They provide access to different functionalities that were described in the system requirements 2.2.1. This section focuses on widgets implementation, operation, and resources consumption.

Each widget is implemented as a separate stateful React component, meaning it is responsible for managing its own state, during its lifecycle. Since all widgets are layout elements, they should behave similarly. However, the composition is superior to the inheritance in the React framework [3], meaning in order to make certain components share the same behavior, each of them should e.g. consist of the same top scope component. That is why before injecting them into the main screen layout, `generateDOM()` function is called (see `src/renderer/components/mainScreen/MainScreen.jsx`). This method, with the use of `widgetIdMap` objects[17], maps the current layout array (see Figure 4.16) to a set of visible widget components, simultaneously wrapping every one of them in the `<WidgetContainer />` (see `src/renderer/components/widgets/WidgetContainer.jsx`), in order to unify their behavior inside of the grid layout. This not only makes the widgets draggable while the application is in the 'Layout Edition' mode (see 3.6) but also allows them to be responsive, as it forces each widget to fit its cell's size.

It is worth mentioning that the application utilizes a global context approach, which allows providing data like e.g. access token or Gmail address to the components that need it, without additional calls to the database. This is because most of the data required by the widgets is injected into so-called `userContext`, after successful login. Any widget is capable of consuming the context by simply calling react built-in function `useContext()`, which returns mentioned data (for more details on contents held in the `userContext` object see `src/renderer/context/UserContext.js`). This approach is for example used by Google-dependent widgets that require an access token to request data they want to display.

In the current state of the system, seven different widgets are accessible by the users, but there will be more of them in the future (see 6.3). The table below summarizes a list of attainable widgets with a brief description of how they operate and what external resources they consume (see table 4.2).

[17] Objects mapping ids of widgets to appropriate React components (see `src/renderer/components/widgets/widgetIdMap.js`)

| Widget name | Source code file | Description |
|---|---|---|
| Calendar | `CalendarWidget.jsx` | Displays Google Calendar events for a selected day. Events are fetched from the Google Calendar API via `getCalendarEvents()` middleware function located in `google.apis` responsibility slice (see 4.7), which utilizes the user's access token and Gmail address from `userContext` for this purpose. Each event is represented as a simple bar containing its starting and ending hour. Whenever the user selects a different day, events from this day are requested. |
| Mailbox | `MailboxWidget.jsx` | Displays five most recent, unread messages from user's Gmail inbox. Emails are fetched from the Gmail API via `getMessages()` middleware function located in `google.apis` responsibility slice (see 4.7), which utilizes the user's Google ID and access token from `userContext` for this purpose. Each message is represented as an accordion, which expanded provides full message content. Whenever the user marks a message as 'read', a new unread message is fetched from their mailbox. |
| Digital clock | `ClockWidget.jsx` | Can operate in one of two modes: clock and timer, which is determined by a stateful variable of this component. By default displays a 24-hour digital clock, which updates every second, date in the following format: `dddd, MMMM D, YYYY` and current timezone. Tapping on the clock opens up a dialog that allows choosing another mode of the widget, which provides typical timer functionalities (start, stop, restart, lap). It uses moment.js library in its implementation. |
| Radio player | `RadioWidget.jsx` | Allows playing music by accessing different radio stations, which stream URLs are fetched from the radio browser API, and then passed to the HTML audio tag as a source attribute. Whenever the user selects a different music genre new API call is created with an appropriate filter parameter. API returns a set of accessible stations including their logo and name. Users can easily switch between the stations using the widget's buttons, which simply update a stateful variable holding the current station index. For the ease of querying the radio browser API, an open-source API wrapper was used in a form of an NPM package. |
| Weather forecast | `WeatherWidget.jsx` | Displays current weather together with the forecast for the next four days. Utilizes Open Weather API for this purpose, which besides raw data provides UI icons of current weather. However, in order to request weather for the exact user's location, their geographic coordinates are needed. Ipdata.co API is utilized for this purpose. |
| News feed | `NewsFeedWidget.jsx` | Displays news header fetched from newsapi.org, which changes periodically. For each header, the user can select 'Read more' option. Doing so opens up a new browser instance (see 4.7.1) with the URL of a news which the header originated from, allowing to read a whole article. |
| Logged in user indicator | `LoggedInUser Indicator Widget.jsx` | Serves as an indicator displaying username, avatar, and Gmail address of currently logged-in user. It does not consume any external resources nor database, as all required information is taken from the `userContext`. |

Table 4.2: Widgets accessible in the current state of the application. Files containing widgets code can be found in the following directory `src/renderer/components/widgets`

# Target environment

In order to use this software to empower a smart home device, the target appliance has to be built first. Any user with minimal technical knowledge can do so, as it is not more challenging than connecting a monitor to a computer. However, several components have to be gathered.

This chapter describes what components are essential and how to connect them to assemble the environment, which can act as an intelligent mirror, by utilizing the Smart Mirror software.

## 5.1 Environment composition

The environment should be composed of five elements as shown in figure 5.1.

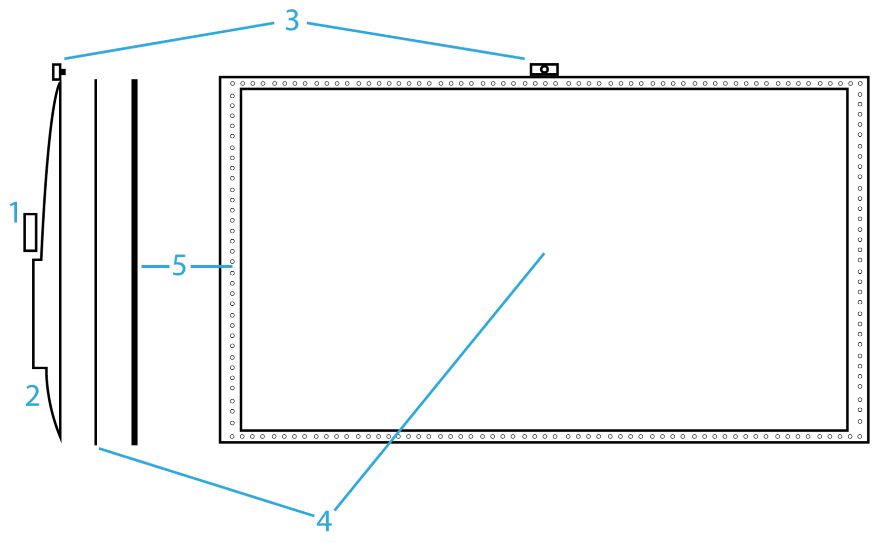


Figure 5.1: Side and front views of a smart mirror device

Each element has its responsibility and recommended parameters:

1. `Raspberry Pi`
   is a computer that runs the Smart Mirror system. The software was tested on a 4B version with 8GB of RAM, and that is a recommended unit. One of its Micro HDMI ports is used for connecting the display.

2. `Display`
   it can be any display with an HDMI port or any other Raspberry Pi-compatible interface. Recommended minimum screen resolution is 1920 × 1080 (full HD). Other resolutions are acceptable, as the UI is responsive, however, screens less than 700 pixels wide are not supported. The system was tested with the use of a 32 inch full HD TV and a 4K monitor.

3. `Web camera`
   a typical web camera, preferably at least HD or full HD resolution, to ensure the best face recognition performance. It has to be connected to one of the Raspberry Pi USB ports. It is required for registration and login purposes (see 4.3 and 4.5.1). The system was tested with both recommended resolutions.

4. `Two way mirror`
   it is an optional component, as the assembled device can be used without it. Its purpose is to protect the screen and act as a reflecting surface, which at the same time lets through just enough light produced by the display so that the UI of the system is visible to the user. It gives the device a mirror-like look, making the user feel like interacting with an actual mirror. It is recommended to use a glass mirror with 70% reflectiveness and 30% transparency as it ensures the best user experience and can be used in wet environments such as bathrooms. Acrylic solutions can be applied when the used display is rather small, however, it is not encouraged as the mirror coating is very fragile and scratches easily.

5. `Infrared touch frame (IR Frame)`
   it is a frame equipped with infrared sensors allowing to map user screen touches to mouse click events, making a simple display a touchable one. Thanks to it the user can interact with the assembled mirror by touching the screen. IR frames come with inbuilt software, making them plug and play ready, so the only thing the user has to keep in mind is choosing the right size - the frame should be as big as the used display.

## 5.2 Software installation

The installation process of the Smart Mirror system on the target device is very simple. It comes down to inserting a properly prepared microSD card in the Raspberry Pi card slot and running the device.

The SD card needs to contain an image of an already prepared and set up system, which will be downloadable from the web. The only thing that the user has to do is simply place this ready image on a fresh SD card, and then insert it into the device. After booting, the Smart Mirror system is ready to use.

The system which image is used in this solution is a Raspberry Pi OS[1] equipped with pre-installed packages, one of which is the Smart Mirror software. It is configured in a way, that after booting, automatically and immediately runs the Smart Mirror desktop application in kiosk-like mode[2], allowing the user to interact with and use all features of the application, but no more than that. Prepared system among others contains a package providing an on-screen keyboard so that the user does not need to use any external devices to provide input, however, they can if they want to, by simply connecting it via Raspberry Pi USB port.

[1]System was also tested on Ubuntu
[2]Mode restricting user actions

# Conclusion

With the increased adoption and use of smart home appliances, home automation builds are getting more complex and convenient, including a more significant number of devices. The Smart Mirror system for a relatively small amount of money, since only components of the target device, have to be bought, contributes to this expansion providing a multi-purpose solution that can help users enlarge the volume of their smart home infrastructure, even more improving the standard of their life.

## 6.1 Summary

The objective of this thesis was the design and implementation of the vast majority of the Smart Mirror system logic along with its user interface and data storage solution. All of these elements were enclosed in an Electron-based desktop application, which integration with other system components, such as face recognition module or external APIs, has also been realized. The application's primary purpose comes down to enclosing many everyday-use functionalities inside easy-to-use widgets.

Compared with other solutions of this kind, the implemented system introduces several advantages including editable layout, built-in face recognition, and a modifiable set of accessible widgets. However, the scope of this paper covers only the first release of the software, meaning it will be further improved and maintained, bringing not only new widgets but also other functionalities, which are discussed in the 'Future work' section (see 6.3).

## 6.2 Discussion

During the implementation phase, it was decided to make Google integration a mandatory condition to register to the system. It may be questionable, as, in the group of potential system users, there might be people that do not wish to or do not use Google services for personal reasons, but they would like to simply use widgets not dependent on Google. This issue could be solved by providing an alternative way to register to the system, skipping the Google integration step. However, this issue was postponed, as the majority of potential users of this kind of appliance usually own similar devices, and depending on the vendor, they also require this kind of integration (e.g. it is not possible to use a Google Home speaker without Google account). It means it is highly probable that potential users of the Smart Mirror system already use these services and are willing to use them on another device to keep all of them in synchronization.

## 6.3 Future work

As mentioned in the summary, this thesis covers only version alpha of the Smart Mirror system, meaning many features will be implemented in the upcoming releases. This section briefly describes some of them.

Future improvements were divided into two segments: widget-related and general functionalities. The first one refers to improvements of existing widgets and the implementation of entirely new ones. The second one describes what functionalities can be enhanced or included in the system in general.

1. Widget-related:

   - **Spotify Music widget implementation**
     will allow users to listen to music from Spotify service, as it provides more flexibility than radio and other smart home devices like e.g. Google Home or Amazon Echo speakers are integrated with this service. This feature would require additional implementation of authorization mechanism (e.g. embedding Spotify authorization web page inside of this widget), as to make requests to Spotify API a credential in a form of an access token is required. However, the mechanism for storing and refreshing the tokens is already implemented on behalf of Google integration and can be also used in this case;
   - **Currencies widget implementation**
     will allow users to view current exchange rates of selected currencies, including cryptocurrencies. It will simply fetch data from an external APIs and present it to the user in an orderly manner (similar to the news feed widget);
   - **Google Tasks widget**
     will allow users to view existing and add new tasks from and to a selected list. This widget will be based on Google Tasks API, so that it is automatically synchronized and integrated with appropriate mobile and web applications provided by Google;
   - **Calendar widget improvement**
     events caching will be implemented, which means that events from a selected day will be fetched only when selecting the date for the very first time. This will optimize the application by reducing the number of requests sent to the Google Calendar API;
   - **Weather widget improvement**
     an option to toggle between metric and imperial units will be added to the weather widget (for now only metric units are supported). Open Weather API will be utilized for this purpose, as it allows to specify units as a request parameter;
   - **Radio widget improvement**
     users will be able to add stations to favorites, which will be remembered in the database;
   - **News feed widget improvement**
     users will be able to select both: source and category (technology, sport, general, etc.) of the news appearing in the news feed widget;

   As the number of widgets increases, so does the amount of data that needs to be stored in the database for each user, meaning the NeDB (see 4.1.3) database may be replaced with the MongoDB DBMS in the future.

2. General functionalities:

   - **Auto logout**
     after some time of inactivity from the currently logged in user, they will be automatically logged out;
   - **Standby**
     by default, the mirror will be in standby mode. Users will be able to wake up the system by touching the screen. In the standby mode screen will be blacked out, so that the device will look like an ordinary mirror from the outside;
   - **Google services access token lifetime enhancement**
     a new access token required for Google services will be requested after each hour of continuous use of the system;
   - **Controlling external LED lights**
     LED lights hidden behind screens and mirrors is a modern design trend. Mechanism supporting external LED lights will be implemented, allowing users to manipulate their color and intensity from the level of the Smart Mirror system's UI;

- **Support for connecting Bluetooth devices**
  in order to play music in better quality, users might want to use some external speakers, which can be connected via Bluetooth. Mechanism supporting connection with this type of devices will be implemented, allowing users to choose the device they want to connect to via user-friendly UI;

# Bibliography

[1] ”Application Architecture”, Electron Documentation. Available: http://man.hubwiz.com/docset/electron.docset/Contents/Resources/Documents/docs/tutorial/application-architecture.html.

[2] Chrome Comic. Available: https://www.google.com/googlebooks/chrome/big_04.html.

[3] ”Composition vs Inheritance”, React Documentation. Available: https://reactjs.org/docs/composition-vs-inheritance.html.

[4] Database of databases - NeDB. Available: https://dbdb.io/db/nedb.

[5] ”Deep dive into Electron’s main and renderer processes”. Available: https://cameronnokes.com/blog/deep-dive-into-electron’s-main-and-renderer-processes/.

[6] Electron Documentation. Available: https://www.electronjs.org/docs/latest/.

[7] ”Front-end Frameworks”, State of JS. Available: https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/.

[8] ”Hosted Projects”, OpenJS foundation. Available: https://openjsf.org/projects/.

[9] Magic Mirror project. Available: https://github.com/MichMich/MagicMirror.

[10] MongoDB. Available: https://en.wikipedia.org/wiki/MongoDB.

[11] ”Multi-process architecture”, Chromium Documentation. Available: https://www.chromium.org/developers/design-documents/multi-process-architecture.

[12] NeDB - The JavaScript Database. Available: https://github.com/seald/nedb.

[13] NeDB - The JavaScript Database. Available: https://github.com/louischatriot/nedb.

[14] NeDB, NPM repository. Available: https://www.npmjs.com/package/nedb.

[15] ”OAuth 2.0 for Mobile & Desktop Apps”, Google Documentation. Available: https://developers.google.com/identity/protocols/oauth2/native-app.

[16] ”Process Model”, Electron Documentation. Available: https://www.electronjs.org/docs/latest/tutorial/process-model.

[17] React home page. Available: https://reactjs.org/.

[18] ”React (JavaScript library)”, Wikipedia. Available: https://en.wikipedia.org/wiki/React_(JavaScript_library).

[19] ”React Without JSX”, React Documentation. Available: https://reactjs.org/docs/react-without-jsx.html.

[20] ”Secure your site with HTTPS”, Google Documentation. Available: https://developers.google.com/search/docs/advanced/security/https?hl=en&visit_id=637728468854793762-3789451689&rd=1.

[21] ”Security, Native Capabilities, and Your Responsibility”, Electron Documentation. Available: https://www.electronjs.org/docs/latest/tutorial/security.

[22] ”Using OAuth 2.0 to Access Google APIs”, Google Documentation. Available: https://developers.google.com/identity/protocols/oauth2.

[23] ”Webpack concepts”, Webpack Documentation. Available: https://webpack.js.org/concepts/.

[24] ”Who uses React?”, Stackshare. Available: https://stackshare.io/react.

[25] ”Writing your first Electron App”, Electron Documentation. Available: http://man.hubwiz.com/docset/electron.docset/Contents/Resources/Documents/docs/tutorial/first-app.html.

[26] M. Dryka, O. Gierszal, B. Pluszczewska. ”Electron Development: A Quick Guide”. Available: https://brainhub.eu/library/electron-development/.

[27] C. Griffith, L. Wells. *”Electron: From Beginner to Pro”*. Apress.

[28] P. Hunt. ”Why did we build React”. React Blog. Available: https://reactjs.org/blog/2013/06/05/why-react.html.

[29] J. Lord. ”Electron 1.0.”, Electron Blog. Available: https://www.electronjs.org/blog/electron-1-0.

[30] F. Rieseberg. ”Electron joins the OpenJS Foundation”, Electron Blog. Available: https://www.electronjs.org/blog/electron-joins-openjsf.

[31] K. Sawicki. ”Atom Shell is now Electron”, Electron Blog. Available: https://www.electronjs.org/blog/electron.

[32] R. Stevens. *”Unix network programming”*. Prentice-Hall, Inc.

# CD contents

Attached CD contains the following folders:

1. `thesis` - electronic version of this document
2. `smart-mirror` - source code of the project