Designing an escape room sensory system

Bachelor thesis

Issa Hanou

Gwennan Smitskamp

Marijn de Schipper

Delft University Of Technology

TUDelft

Designing an escape room sensory system

Bachelor thesis

by

Issa Hanou
Gwennan Smitskamp
Marijn de Schipper

Project duration:	November 11, 2019 – January 31, 2020	
Thesis committee:	Ir. O. W. Visser,	TU Delft, coordinator
	Dr. E. Isufi,	TU Delft, coordinator
	Ir. T. V. Aerts,	TU Delft, coach
	J. W. Manenschijn Bsc,	Raccoon Serious Games, customer

Preface

This is the report of the Bachelor End Project (in short, BEP) created by Marijn de Schipper, Gwennan Smitskamp and Issa Hanou, for obtaining their bachelor degree in Computer Science at the Delft University of Technology.

For ten weeks, we have worked to create a system for running escape rooms for small groups of people. The system consists of three parts: back-end, front-end, and a client computer library and was commissioned by Raccoon Serious Games. The system and development process will be explained in this report.

We would like to extend our gratitude to the complete team at Raccoon Serious Games for offering us the chance to work on a very inspiring product and providing their support during the project. Furthermore, we want to thank Jan-Willem Manenschijn in particular for his guidance during the project and trusting us with building a system for his company and actually putting it to use.

Finally, we want to thank our coach, Ir. T. V. Aerts, a teacher at the Delft University of Technology, for assisting us and finding the time to guide us during the project.

Issa Hanou
Gwennan Smitskamp
Marijn de Schipper
Delft, January 2020

Executive Summary

Raccoon Serious Games develops different kinds of gaming experiences, including escape rooms. In an escape room, a group of players, usually between 2 and 20 people, are locked in a room, where they have to find clues and solve puzzles to escape. When such a room is played, there is always an operator, monitoring the progress of the players and keeping an eye on their safety. Modern escape rooms are quite technologically advanced, where all components of the room interact with each other.

To run these games, Raccoon Serious Games needs a system which should manage all technical aspects within the room and provide a way for the operator to keep track of the players. Furthermore, the system must be flexible enough to handle many different escape rooms, which are hardcoded in configuration files.

To meet these needs of Raccoon Serious Games, a team of three developers has developed a new system, called S.C.I.L.E.R.. In ten weeks, they have created a system from scratch. First, they researched technologies and requirements that would be necessary for the system and afterwards implemented the system based on their findings.

S.C.I.L.E.R. allows an operator to monitor a complete escape room. The system connects a user interface to all the devices in the room, which are controlled by the user as well as the configuration of the escape room. This configuration is generated from a JSON configuration file, containing all information for the escape room. The system allows the operator to send hints to the players, control the time and manage their progress through the status of devices in the room and the puzzles they have to solve. It also contains the feeds of cameras in the room to actually see the players. Furthermore, it provides the user with a way to check a configuration file and put it to use.

The system will be used by Raccoon Serious Games in the near future for the escape rooms that they will be hosting.

Contents

1 Introduction

In a rapidly changing world, technology has made quick advancements, while at the same time, issues like security and sustainability become ever more pressing. Teaching people about security, how to protect themselves and what the dangers are when handling technology, is very important [1]. Although many different methods to do so exist, a new surging way is gamification, which is often implemented in serious games. These are described as 'a game which is designed with a goal other than just entertainment' by Raccoon Serious Games [7].

These games are designed to encourage problem-solving and cooperation while stimulating participants to think about certain issues, for example, climate change. Serious games exist in a number of different formats: large escape events (where 200 people play at the same time in small groups), board games, digital games (for example, mobile games), and physical escape rooms. The latter can be thought of as one or several rooms in a building in which the goal is usually to escape but can also focus on finding a certain object. These escape rooms are usually played in teams of 2 to 20 people, depending on the format of a specific escape room.

Raccoon Serious Games is a company that focuses on designing serious games, in all different forms. While they already had a system to host escape events for large groups, called M.O.R.S.E., they also wanted to have a system to host escape rooms for smaller groups. The system should facilitate users to play escape rooms, with all components working together, while also allowing an operator from Raccoon Serious Games to monitor the players, see how they are doing, give hints to help them or intervene when necessary. Although Raccoon Serious Games already owned a system for this purpose, this system was outdated and no longer maintainable, they wanted a new system.

For this reason, a new system called Sensory Communication Inside Live Escape Rooms, hereafter referred to as S.C.I.L.E.R., was developed. This was developed as a Bachelor End Project (BEP) for the Computer Science & Engineering program at Delft University of Technology.

In this report, the BEP design and development process of S.C.I.L.E.R. is documented. First, the problem addressed by this BEP is defined and analysed in Chapter 2, upon which research was done, documented in Chapter 3. Then, the research is used to come to a design of the system, explained in Chapter 4. The implementation of the design is documented in Chapter 5 and the testing of the system in Chapter 6. The ethical implications of the developed product are mentioned in Chapter 7. Furthermore, the process of the development is examined in Chapter 8 upon which conclusions are drawn in chapter 9. Finally, the results are discussed and recommendations are made in Chapter 10.

2 Problem definition and analysis

The problem that this thesis attempts to solve is the automation of operating an escape room. This problem is further defined in this chapter by the challenges that this thesis wishes to address and the needs and motivation of the client. First, the context of the problem will be sketched.

2.1. Context

An escape room, as defined by this thesis, is an isolated location, consisting of one or more physical rooms, in which a game is played by a team of players who want to escape from the room or reach another type of goal by finding and solving puzzles.

The puzzles inside the room are linked in such a way that when they are solved in order, the team wins and escapes. Furthermore, some puzzles might be dependent on others, before they can be solved. For example, a button can only be pushed when a flickering light is on; or, only when an object is found, can the players figure out what code to enter on a keypad.

When a team is playing an escape room, there is always an operator monitoring their progress. An operator will start the game, keep track of how the team is doing, give hints when deemed necessary and can stop the game if the team calls for it.

2.2. Client

The client, Raccoon Serious Games, wants to have a product that automates as much as possible in operating an escape room. The system must be robust, guaranteeing that it does not crash in the middle of a game run containing many devices giving input in an interval of deciseconds. The number of devices used in an escape room can easily rise to 20 to 50 units that need to be controlled.

Raccoon Serious Games has several escape rooms that they want to be able to run. It should be possible for multiple rooms to run simultaneously.

Finally, the product must be agile and should allow for new escape rooms to use the system. The configuration of escape rooms should be flexible enough to allow for new rooms to be created and designed.

The client operates in the serious games business, which is where this product will be used. Although the techniques used are similar to a home automation system, more on this in Chapter 3, this system will be personalized for escape rooms, as used by Raccoon Serious Games.

2.3. Challenges

During a game run, there are several challenges that this system must deal with, which can be categorized to be for player, operator or configuration.

2.3.1. Player

When a team is playing an escape room, they must not notice any delays in the game and visual effects inside the room. When a puzzle is solved, the players must immediately know they did, as there are also puzzles that give different outputs for wrong and right answers as a hint to solving the puzzle.

2.3.2. Operator

From the operator's perspective, the biggest challenge comes with the updating of the room's status. The operator should be notified of the current status of all devices in the room, in the sense of both the network connection and what the current value of a device is. Additionally, the operator should know the status of all puzzles which the team needs to solve. As some devices might change their status frequently when players try to solve a puzzle, the updating must happen quickly, so no delay can be noticed.

Furthermore, all members of the Raccoon Serious Games team should be able to handle the operator interface, so it should be user-friendly and not require a lot of explanation to manage an escape room.

2.3.3. Configuration

Configuration concerns the settings of an escape room and specifies everything from the time players have, to the sequence of events and mood light settings. The configuration will be read into the system, upon which game runs can be played. The configuration faces two main challenges: flexibility and usability.

First of all, the format in which the configuration is read should support all possible events that could happen in an escape room. It should be able to handle many different devices with different input en output types.

Secondly, the format in which configuration is entered should be easy to create for members of Raccoon Serious Games team. It should not take too much time to write a configuration and it should be easy to understand how to translate an escape room to a configuration.

2.3.4. Components

As the system should handle configurations and communicate with many clients, one back-end should handle all the communication in the system and make sure the proper actions are taken at the right time. Furthermore, the operator must be able to manage the room, so a user interface, also referred to as front-end, should be created where the operator can do everything named in Section 2.3.2. Finally, the devices in the escape room are controlled by client computers. These are mini-computers, which can be a Raspberry Pi or Arduino, for example, which work from a library that belongs to the S.C.I.L.E.R. system.

These three components together with the configuration should make sure that the escape room runs smoothly. In the research, Chapter 3, more explanation is given on how these components interact together and how they will run the system, as well as the communication protocol used to handle communication between the components.

2.4. Conclusion

In conclusion, the system should support the operator of an escape room by automating as much as possible, while ensuring quick communication with devices in the room such that there is no delay noticeable to the players or operator. Finally, the system should be easy to handle and understand both during a game run and when writing configurations.

3 Research

To meet all the requirements that follow from the problem description, see Chapter 2, research was needed. The Product Plan, which can be found in Appendix I, describes the goal of the research phase was to find out more about what the system would need and how it should be built (Section 3 of the product plan) and includes a glossary that applies to this report as well (Appendix A of the product plan). To conduct this research, the following research questions were formulated:

- In what way is the old system of the customer not suitable for future use?
- What are the most important features to have in an escape room system?
- What is the best set-up to use to communicate between different components of an escape room system?
- What type of hardware is small enough to hide in the room, while it is also suitable as a reliable backend, communicating with multiple client computers?
- What configuration format or template contains all necessary information, can be written by a member of the Raccoon Serious Games team and can be interpreted by the system?
- What is the latency threshold that the system can allow to avoid hindrance to the players?
- What are the best tools to use for keeping a consistent code base?

The full report on the findings can be found in Appendix J, including a conclusion on the decisions made based on the research. In this chapter, the research will be summarised and the major conclusions repeated.

3.1. Old system from Raccoon Serious Games

Raccoon Serious Games already owned a system for running an escape room. It was developed about four years ago, in a short amount of time. As part of the research conducted for the development of S.C.I.L.E.R., the old system was studied. It turned out this system was outdated, undocumented, unreliable and limited in its options. Although it performs strongly for certain features, extending it with new features was hard, as found by the client. Therefore, it was decided early on in the project that building a new system from scratch would be better.

In the first meetings with the client, it was already agreed that the new product would function differently from the old system. There would no longer be any authorisation required, as one instance of the system would just handle one escape room. Furthermore, the entering of configurations through a front-end was also found to be a nice feature, but not a necessary one and not the focus of the new system.

So, the old system was mostly studied in terms of features that might be of use to S.C.I.L.E.R.. Most importantly, the library that the old system used for its client computer seemed well-structured and the idea of a main class with specific implementations for different devices was taken as a basis for the new library. However, as more discussed in Section 3.3, the communication method used in the old system is not used in S.C.I.L.E.R., so the implementation of the new library was done from scratch.

3.2. Features of an escape room system

Raccoon Serious Games owns another system, called M.O.R.S.E., which is used for managing large scale escape events. Some responsibilities of this system are similar to what S.C.I.L.E.R. will be doing, for example, configuring escape rooms/events and communicating with many clients. Therefore, this system was studied in-depth. As both systems are used by the same company, it seemed preferable to keep some functionalities and technologies the same, so members of the Raccoon Serious Games team would be able to easily handle both systems. However, it is important to note that the M.O.R.S.E. system was only used an example of how to handle configurations and to see what languages and frameworks it uses; while the code base of S.C.I.L.E.R. was started from scratch and is not based on M.O.R.S.E..

The most important conclusions drawn from M.O.R.S.E. are to keep the configuration simple, yet flexible, and some nice-to-have features on the front-end for both players and operators that should also be included in S.C.I.L.E.R..

Additional to M.O.R.S.E., two escape rooms systems on the market were also studied for interesting features. The two analysed systems are Houdini[1] and QUEEN[2]. Both companies' software can be bought online for anyone to easily build escape rooms themselves. Although these systems are very elaborate and offer many features, it was decided that building a new system from scratch would be a better option to fit the wishes of Raccoon Serious Games. This was decided mainly because building a new system would allow the client to have a product that meets all of their needs exactly. Furthermore, it was found that both existing systems had limitations, not allowing users to work with custom hardware or not supporting audio, which eliminated the systems as possibilities for the client.

To develop the new system to meet all of Raccoon Serious Games's wishes, members of their team were interviewed to find out what features they would want to see in the new system. These were all translated into requirements for building the new system. The final list of requirements can be found in Section 4 'Product backlog' of the Product Plan, in Appendix I.

Finally, a few home automation systems were studied as the communication techniques used is similar to what S.C.I.L.E.R. should be able to do. However, the new system will be more elaborate and specific to escape rooms created by Raccoon Serious Games.

3.3. Set-up of the communication in an escape room system

As the set-up of the system is a complicated and crucial part, this was the biggest research point. Because the system has three main components, it was important to find out the best set-up for each component as well as how these would be connected. Therefore, there were four sub-questions to this research question:

- What is the best technique to use for communication between the back-end and client computers?
- What language and framework are most suitable for maintaining communication with different client computers while managing the escape room?
- What language can be used to code the library for the client computers to ensure proper instruction handling and easy to control the connected device?
- What language can generate a user-friendly interface on the mobile device of the operator, while communicating with the back-end?

The three components and their requirements were studied separately to come to the best structure for each component. The communication technique was also researched independently of the languages used, to make sure it was the best technique. In the end, all conclusions were put together to find a final structure of the system.

3.3.1. Communication technique

Two main techniques can be used: request-response or publish-subscribe. The difference lies in the matter of when messages are sent. For request-response, answers are only sent upon request; while in a publish-subscribe model, clients subscribe to topics and receive all messages related to those topics. For an escape room system where clients constantly update the back-end on their status, the publish-subscribe model was chosen as a more efficient one.

[1]https://houdinimc.com/
[2]`https://escaperoomdoctor.com/queen`

The Message Queue Telemetry Transport Protocol, in short MQTT, is a well-supported publish-subscribe protocol, with a library in almost every programming language. Moreover, this protocol was found to have the least latency of several compared protocols, see Appendix 3, Section 3.1. Therefore, it was decided to use the MQTT protocol for S.C.I.L.E.R., as the latency is a great concern in guaranteeing minimal delay.

3.3.2. Back-end

Several main languages were considered: JavaScript, TypeScript, Go, Python and Java. The full comparison can be found in Appendix 3, Section 3.2. In short, all languages are well-documented and widely used, so to come to a decision, they were mostly compared in abilities and how they handle real-time updates and concurrency. Furthermore, the experience of the team members was taken into consideration. The conclusion was drawn that Go would suit the back-end best, as it handles concurrency well, allows for object-oriented programming, is strong in parallelism and one team member already had a good experience with it.

3.3.3. Client computers

The library used for client computers should be able to easily communicate with the device itself. The full explanation can be found in Appendix 3, Section 3.3. As most devices will be controlled by a Raspberry Pi, Arduino or other micro-controller, the language used for the client computer library must support these. Python is widely used in programming Raspberry Pis, it can also communicate well with the other options, and several libraries for MQTT in Python exist. Finally, Raccoon Serious Games already coded several devices in Python, so a Python library would easily allow for this code to be reused.

3.3.4. Front-end

Several frameworks were studied, which were all found to be pretty solid options for a front-end in this system. Angular, React, Vue and Django were compared in terms of documentation, and their ability to support mobile-friendly websites. Django can be used to create a front-end but is mostly used when a database is used in the system. As early on, it was decided to not use a database, Django was quickly rejected. The complete report of this can be read in Appendix 3, Section 3.4. TypeScript is a strongly-typed superset of JavaScript and object-oriented programming would support the system well, so would be preferable to JavaScript. Furthermore, the M.O.R.S.E. system is written in Angular Typescript, so to support consistency amongst the products of Raccoon Serious Games, the final choice went to Typescript Angular.

3.4. Hardware

The hardware on which the back-end and front-end are run should be reliable to ensure that the system does not crash in the middle of a game run. The full design of the system running on the hardware can be found in Section 4. Another factor that was considered is the availability of the hardware that will be selected. The choice was made to initially run the system from a Raspberry Pi. This set-up will be thoroughly tested a little past halfway the project when there is a minimal working product, and if these tests conclude that the hardware does not meet the requirements and expectations of the client, another option that came up in the research, see Appendix 3 Section 4, will be selected.

3.5. Configuration

After closely studying the configuration structure of the M.O.R.S.E. system and the Hass.io[3] home automation system, it was decided that the configuration would be entered in JSON format. The entering of the configuration through a front-end was set as a nice-to-have feature, where the user's input would be translated to a JSON file, which would then be entered into the system.

The RuleSets as used in M.O.R.S.E., seemed very efficient to work with. As M.O.R.S.E. has a similar type of configuration in the form of a game with a duration, puzzles and hints, the structure of the rules was used as a model for the configuration format of S.C.I.L.E.R.. However, in the brainstorming phase, it was decided to simplify the configurations of S.C.I.L.E.R. more and create a more versatile tree structure for conditions.

[3]`https://www.home-assistant.io/hassio/`

3.6. Latency

The latency in the system will be the delay that could be noticeable to players. Since the client does not want players to notice any delay at all while playing, a few similar systems were studied and their delays compared. After some analysis, the threshold for S.C.I.L.E.R. was set at a maximum delay of 100ms for round-trip communication.

3.7. Tools

Finally, some research was done on tools to support a clean codebase and ensure consistency. Several tools were chosen for different languages, and in the end, a code formatting tool and code style checker were selected for each of the three system components. Finally, Travis was selected to be used as a maintenance tool for the GitHub repository.

3.8. Conclusion

So, researching on how to set up the system and the most important features, a plan can be drawn on how to develop S.C.I.L.E.R.. First off, the system will be run from the following set-up: a publish-subscribe protocol, implemented by an MQTT broker, will manage the communication in the system; the back-end will be written in Go and will handle all communication and the configuration file; the Angular TypeScript front-end will provide a user interface; the client computer library to control devices will be written in Python.

Furthermore, the system should fulfil the following requirements: the latency for a round-trip communication should stay below the threshold of 100 milliseconds and the configuration file will be written in JSON and consist of RuleSets to define the escape room puzzles.

Finally, from the research, a few points of testing can be concluded. The latency threshold can be tested by measuring the round-trip communication time. The user interface can be tested whether it meets the Raccoon Serious Games team's expectations, by doing a user test. And, the set-up can be tested on Raspberry Pi, to see if the hardware holds up to meet the performance requirements. The results of these tests can be read in Chapter 6.

4 Design decisions

In this chapter, all design choices are explained. First, the system itself is analysed and the different components are elaborated on. Then, the design choices for the three components are justified. Accompanying visualisations are included to explain the different system components and set-up.

4.1. System

The system will consist of three components: back-end, front-end and client computers, which implement the S.C.I.L.E.R. library. In Figure 4.1, the set-up of the system can be seen. In this section, the choice for including and excluding certain general elements of a software system is explained.

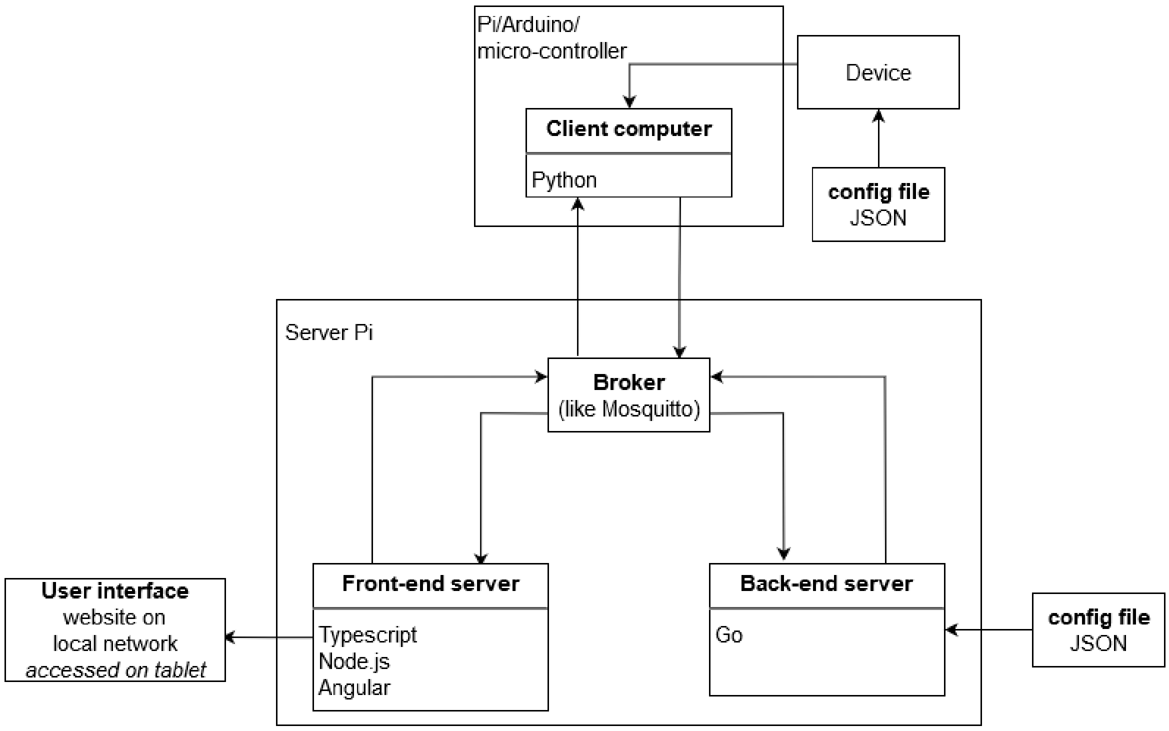


Figure 4.1: System structure, showing all the components in the system and how they interact together.

4.1.1. Server

The system will be run from a server on a Raspberry Pi, which powers the MQTT broker, the back-end and the front-end. This Raspberry Pi will be a model 4, which is better than a model 3 in several ways [5]. It has improved CPU performance and more RAM, which will facilitate faster communication from the serve Pi to client computer Pis. Furthermore, the increased working memory compared to the model 3 will allow the server to handle large escape rooms. Therefore, this is the hardware chosen to run the server.

The front-end that will be run from the server Pi, will publish its website on a predetermined link. This link can then be accessed on a tablet, mobile phone or computer, so the operator can manage the escape room.

4.1.2. Broker

The MQTT broker will manage all the receiving, filtering and publishing of messages. The choice for MQTT is explained in Section 3.3. As MQTT is a publish-subscribe protocol, it works by sending messages to a certain topic, and all clients subscribe to topics they are interested in. These messages are always structured in JSON. Furthermore, MQTT is implemented with a keep-alive, to keep checking if certain clients are still connected.

As all clients to the broker will be subscribed to topics, the broker will make sure that all the messages it receives are then published to the appropriate topics. The following topics will be used:

- **client-computers**: the topic for communication to all client computers
- **back-end**: the topic for all communication to the back-end
- **front-end**: the topic for all communication to the front-end
- {**client-name**}: the topic for communication to a specific client computer
- {**label**}: the topic for communication to all clients which correspond to the label, for example, "hint"

Each client to the broker has a client id, which ensures that no two clients with the same name exist. Especially for the back-end, this is very important. However, the system does allow multiple front-ends to connect, which all receive the same information.

The communication that happens across the system is documented in the manual, found in Appendix D. In Figure 4.2, a visual overview of the communication can be seen, in a simple system with back-end, front-end, a 'display' client computer and a 'lights' client computer. It shows the startup of the system, sending of statuses, starting the game, sending a hint and a puzzle involving players answering.

4.1.3. Database

There will be no database, as the system will be run on a local network for the duration of one escape room. Although the escape room should be possible to be set up again elsewhere, this should not require a full database. These were the wishes of Raccoon Serious Games, so the choice was quickly made to no run form a database, but a file with information.

Therefore, an instance of the system will be run by configuration files. These files are written in JSON format and include all the necessary information for an escape room. These files are saved locally and entered in the system, either by hardcoding the file name or entering a new file through the front-end. JSON was chosen as it provides a clear structure within a file, and every language contains JSON parsers to easily retrieve data from the file. Finally, as the MQTT messages are also in JSON, this allows for easy communication within the system.

A concern could be a system crash in the middle of a game run, for example, during a power outage. In this case, the system can no longer communicate with its components. Furthermore, escape rooms of Raccoon Serious Games are implemented with electromagnetic doors, which need power to keep them closed. The moment the power goes out, all doors open as a safety measure and the game can no longer continue. In this case, it does not matter whether the system remembers any data from the game run because the players cannot continue anyway.

4.1.4. Configuration

The main configuration file will include the following properties:

- `general`: general info about escape room: name, duration, IP address of the MQTT broker host
- `devices`: information about the devices which are used in the escape room
- `puzzles`: the different puzzles in the escape room, with their name, what needs to be done to solve them, and what happens after

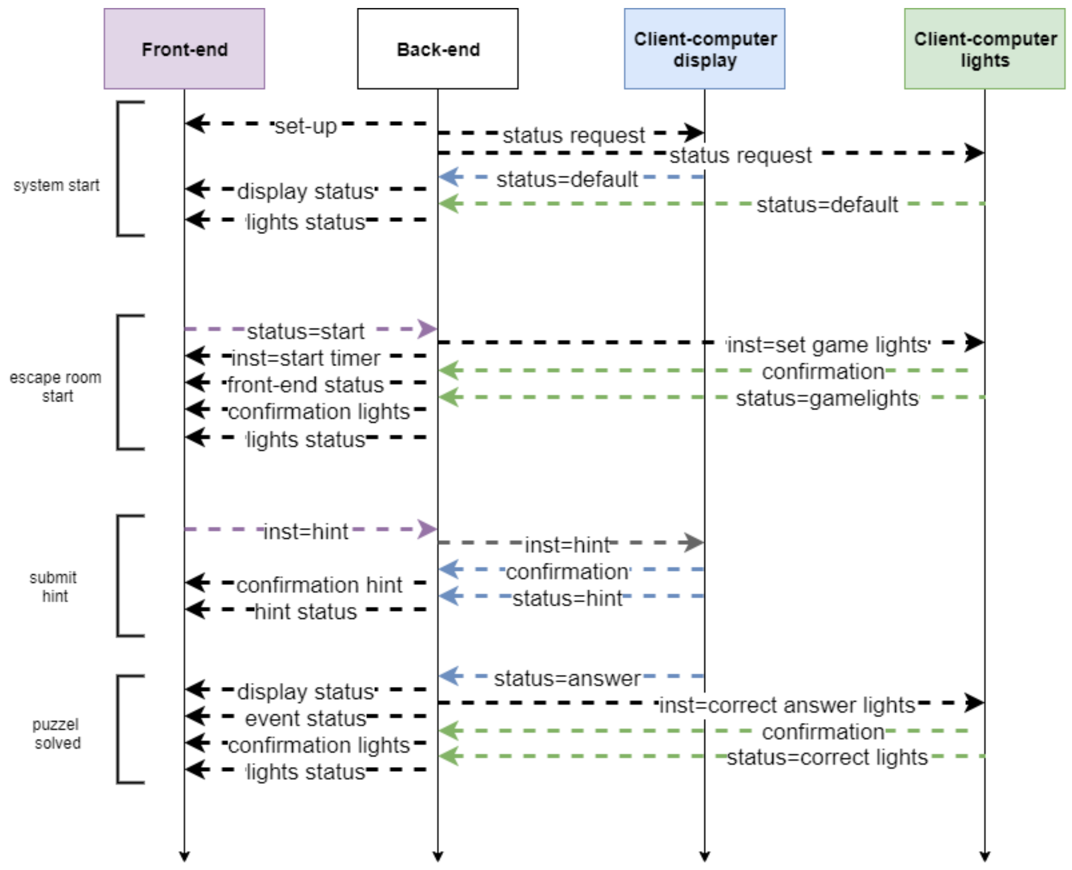


Figure 4.2: Messaging in system, showing all messages in a scenario where the system is started for an example escape room, a hint is sent and a puzzle is solved.

- `general_events`: these are events that will happen in the escape room

The handling of this configuration is explained in Section 4.3.2. Furthermore, each device will have its configuration, which specifies information custom to the device, more on this in Section 4.4.

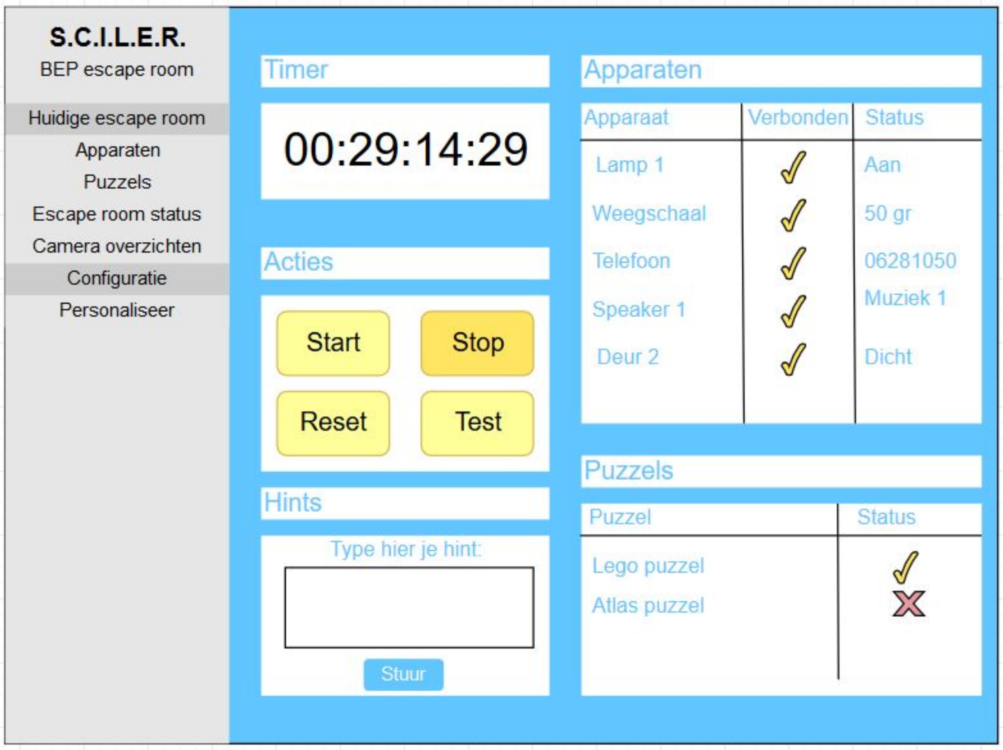
4.2. Front-end

The design of the front-end will be very basic and user-friendly, though still including all necessary components. The front-end will be used by the escape room operator, using a tablet, so the design must be suitable for smaller screens. In the research phase, the members of the Raccoon Serious Games team were interviewed to find out how they would like to see the operator screen. From this, the initial design was drawn, which can be seen in Figure 4.3. It shows the interface for desktops, but for smaller screens, like tablets and mobile phones, the menu on the left will be collapsed by default.

The most important parts are the timer keeping track of how much time the players have left; the status of devices and puzzles, so the operator can see how the team is doing; the buttons to control the room (start, stop and room moods); and the hint section, to send hints into the room. As these components will be required to run an escape room, these are deemed the most important and will be implemented first.

Besides the home screen with the above-mentioned components, nice-to-have additions to the front-end will be camera screens and the submitting of a configuration. Therefore, the menu will allow the user to switch screens, although the main functionality for running the escape room will be on the home screen. These features were discussed with the client to be very favourable, but a well-working system the main components was more important.

The front-end will be easy to use and require no additional explanation to what the data means. Furthermore, it will update with live data to always show the real-time data to the escape room operator. Other nice-to-have features, like pre-defined hints that can be selected and more buttons that can be generated from the configuration, are not in the initial design but might be added during development.



Apparaat	Verbonden	Status
Lamp 1	✓	Aan
Weegschaal	✓	50 gr
Telefoon	✓	06281050
Speaker 1	✓	Muziek 1
Deur 2	✓	Dicht

Puzzel	Status
Lego puzzel	✓
Atlas puzzel	✗

Figure 4.3: Initial front-end design, showing a menu on the left side and the main functionalities of the home screen.

4.3. Back-end

The back-end is responsible for handling all communication and working with the configuration. The latter includes reading, type checking, implementing logic and handling puzzle rules.

4.3.1. Communication

The back-end will communicate with all clients and the front-end through the MQTT broker. The JSON messages sent will be structured with ClientID, TimeStamp, Type and Contents. The ClientID and TimeStamp are from the sender, the type defines how the Contents is structured. The type of messages that the back-end will receive are: `connection, status, instruction` and `confirmation`.

The back-end will have a handler to handle all different messages it receives, including the different instructions it can manage. The handler will then publish messages to topics that are already defined to make sure the instruction or other data is received by the proper client. The front-end might receive a wide range of data, published to the front-end topic, while client computers receive instructions on the topic `client-computers`, their topic or a general labelled topic.

4.3.2. Configuration

The configuration will be read in from a JSON file, a visual representation of the data in the file can be seen in Figure 4.4, in the form of a UML diagram. A config object will then be created, which has type-checked all constraints and inputs. Type-checking is important so that once a configuration is loaded, the system can ensure that no crashes happen during a game run.

The most important part of the configuration is the condition logic. The objects used to define all the logic of events happening in the room are puzzles and general events, which both implement the `Event` interface containing a list of rules.

The actual handling of events happens when a status message from a client computer comes in. The back-end will check whether any rules exist of which conditions can now be resolved, and if so, that rule's actions will be executed. Because the maps created in the configuration handler cache a lot of data, most actions can be done by lookup, which will speed up the system.

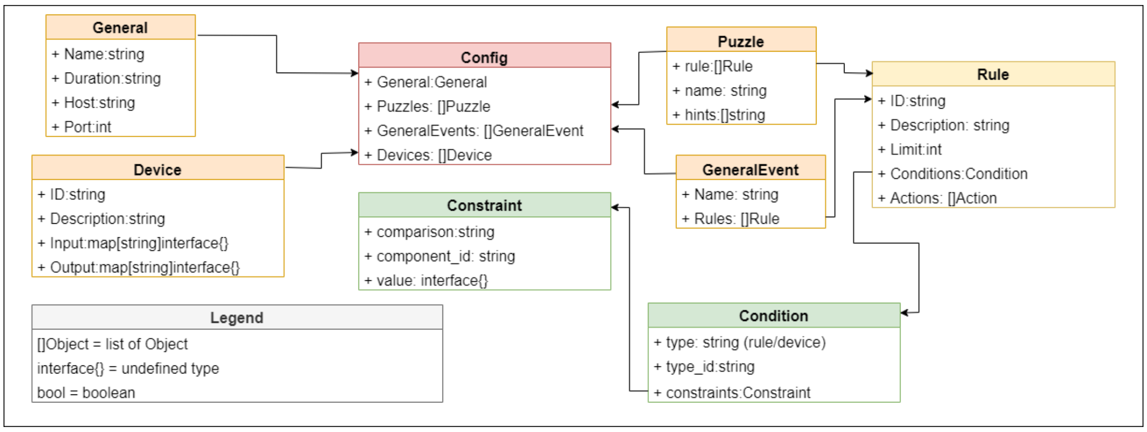


Figure 4.4: UML of the configuration file, showing what entries need to be in there.

The exact handling will now be explained, and its pseudocode can be found in Algorithm 1. Here, the `statusMap` already exists, which returns a list of rules, called `ruleList`, in $O(1)$. The lookup for the status value of a component from its id is also a map lookup that can be done in $O(1)$.

Algorithm 1 Pseudocode of how events are handled. When a device's status comes in, all rules that have conditions based on this device are checked and their actions executed when the conditions resolve.

```
type Message: (device_id, time_sent, type, contents)
type Operator: (operator, list)    ▷ Both conditions and constraints are handled as operator, with recursion
type Condition: (id, type_id, constraints)
type Constraint: (comparison, component_id, value)
procedure RESOLVE(tocheck)
    if tocheck tyepof Condition then
        return Resolve(tocheck.constraints)
    else if tocheck typeof Constraint then
        return compare tocheck.value with tocheck.component_id's value using tocheck.comparison
    else
        if tocheck.operator == OR then
            result = false
            operatorInUse = OR
        else if tocheck.operator == AND then
            result = true
            operatorInUse = AND
        for object in tocheck.list do
            result ← result operatorInUse Resolve(object)    ▷ The right operand is evaluated conditionally
        return result
message = message received from any client
ruleList ← statusMap[device_id]
for rule in ruleList do
    if rule.executed < rule.limit then
        result ← Resolve(rule.conditions)
        if result then
            for action in rule.Actions do
                action.execute()                              ▷ Handled in parallel
            rule.executed + = 1
```

Analysis When studying the pseudocode for event handling, as seen in Algorithm 1, the runtime of the algorithm can be derived. First of all, the actual algorithm consists of a for-loop over the rule list, which takes $O(r)$ time, where r is the number of rules. As each rule will be checked, this is both the upper and lower bound run-time. Only if the conditions resolve to true, will the actions be executed. As all actions have to be executed, a loop is used. However, the different actions will be executed in parallel, which the Go language is very fluent in.

The most important logic is the `Resolve` procedure. To start, the input size is defined as the tree size, i.e. the number of nodes in the tree. Note that it is not a binary tree, but just a tree, where the width can vary per depth.

For a leaf of the condition tree (which is just one condition), the constraint tree should be resolved. This is handled in the same way as a condition tree.

Finally, a leaf of the constraint tree is handled by evaluating the value from the message using the comparison and comparing the value of the constraint. This comparison can be done in $O(1)$ for all cases except `contains`, which is only possible on array values. In that case, it is $O(s)$, where s is the length of the array. In all other cases, it is number comparison or equals (also for strings and booleans).

For the traversing of the condition tree, as it is not a binary tree, this process does not have a 2-base log complexity. Furthermore, lazy evaluation will be used, so the best-case scenario should differ considerably from the worst case, as, for example, AND conditions will terminate to false with the first false result. Finally, different conditions in the same depth of a tree will be handled concurrently, so breadth-first search is applied.

When taking the condition tree, the maximum depth is defined as $d1$ and the maximum width as $w1$, for the constraint tree these are $d2$ and $w2$, respectively. Bread-first search can be solved in $O(n)$ where n is the number of nodes the tree has, as everything needs to be evaluated, there is no win in run-time here.

Conclusion This amounts to a total worst-case run-time complexity of $O(n*m*s)$ for the `Resolve` method, where n is number of conditions, m is the largest number of constraints from any condition in the first tree and s is the size of the largest array that needs to be searched (if any exist). Because `Resolve` is called on every rule, the full event handling method has a worst-case run-time complexity of $O(r*n*m*s)$.

In general, most units (rule/timer/device) will not appear in all rules, but only a few (around 5), so r will be relatively small. Furthermore, arrays are hardly used, so s will usually be 1. Therefore, this worst-case run-time seems high but is not a limiting factor to the system. It is important to keep in mind that the run-time complexity can easily run high when creating rules which depend on five different units ($n = 5$), where each unit has ten components to be checked ($m = 10$)

4.4. Client computers

The library that is used for implementing a client computer will handle all communication to the back-end. It will include an application file and an abstract Device class. The device class should be extended by a custom device class which handles a certain device.

The application will be implemented to handle all the communication of a client computer, such that it is the same for all client computers. This includes the handling of messages and connecting to the back-end. It should send connection messages to the broker, handle instructions accordingly, send confirmations about performed instructions and send its status whenever it detects a change or its status is requested.

The abstract device class will force the devices to make sure that implements methods called from the application file. These methods will facilitate the custom test functionality of the device, the reset status that the device should return to on instruction and the implementation of any other custom instructions that the device should listen to. Furthermore, the device should be forced to implement how to return its status, because the format differs per device and should include all information useful to the operator.

To create a client computer, the device class should be extended, and a configuration file needs to be written to define the client computers input and output components and the type of data they handle. Furthermore, it defines the name and description of the device and provides the IP address of the MQTT broker host.

Finally, the library should be clear to read and properly documented, as it should be published to a package installer, so it will be available to all developers that will implement devices for the S.C.I.L.E.R. system. As the library is written in Python, the package installer will be pip[1].

[1] https://pypi.org/

4.5. Conclusion

To sum up, the front-end design will allow the user to operate the escape room through a user interface accessed from a mobile device, usually a tablet. It will include all functionality necessary to properly run the escape room, which will be tested through usability testing, more on this in Chapter 6. Furthermore, additional features to improve the user experience will be added to the system if there is enough development time.

The back-end will handle a configuration file and afterwards communicate with all client computers and the front-end. The configuration file will store all the data in a JSON format, for easy access. By creating maps of the data on start-up, the system ensures a fast event handling when statuses arrive from client computers and can quickly execute actions based on the events. Type-checking will ensure that once the system is set up, no errors occur during run-time.

Finally, the client computer library will force the user to implement client computers in such a way that they are compatible with S.C.I.L.E.R.. This will make sure that no errors occur during run-time due to an ill-implemented client computer.

5 Implementation

This chapter will shed light on the implementation of the design that is described in Chapter 4. Changes to the original design plan are elaborated and the three components are also completely analysed individually, as well as the system as a whole.

5.1. System

The implementation of the system still complies with the original design. The set-up with running a back and front-end from a single Pi, as well as the broker which connects the whole system, was a good choice. The system has been implemented on a Raspberry Pi model 3 but is also compatible with a model 4. However, as model 3 was made available to the team earlier on, that was used first. Because running the system from a model 4 Pi had less priority than improving the system itself, and there was not enough time to do both, the system has only been run on a model 3.

One change that was made to the original system design, was the addition of a JavaScript library which the client requested for better compatibility with JavaScript programmed devices. An updated system set-up can be seen in figure 5.1

The Mosquitto broker[1] was chosen to handle the MQTT protocol. Each programming language has an MQTT library, which all connect to a broker. Mosquitto was chosen as it is a widely used broker, which can be started both from a Raspberry Pi as well as a Windows terminal. So, this was a good choice for the developers, as they all have Windows laptops and this enabled them to run the system locally.

5.1.1. Logging

The complete system keeps logs of events that happen and message that are received and sent. The back-end logs everything to its terminal, which can only be accessed when reading the terminal on the serve Pi, and to a file which is saved locally on the Pi. The file can be accessed after each system run, whether it ends naturally or because of a crash, and is named with the date and time of the system start in the format `02-01-2006-15-04-26`, such that no two files have the same name. The choice for local files was made so that if a system error would occur and the system breaks down, the file is saved and can be accessed on the Pi later on for debugging purposes.

Client computers that implement the Python library have the same implementation of logging as the back-end to both their terminal and a local file, again for the same reason as for the back-end. Client computers that implement the JavaScript library take in a logging function in their constructor, so the user can define how to log their data. More on this in Section 5.4.2. The front-end keeps logs in its console on the client-side because writing logs to a file on the client-side does not serve any purpose. However, as it uses one main logger this can be altered to log to the server-side.

Logging throughout the system uses three levels: info, warning and error to inform the user about the severity of logging. Logging messages also include a complete timestamp in the format `2020-01-24 13:11:57`, also including a timestamp, and the logs include a JSON message.

[1] `https://mosquitto.org/`

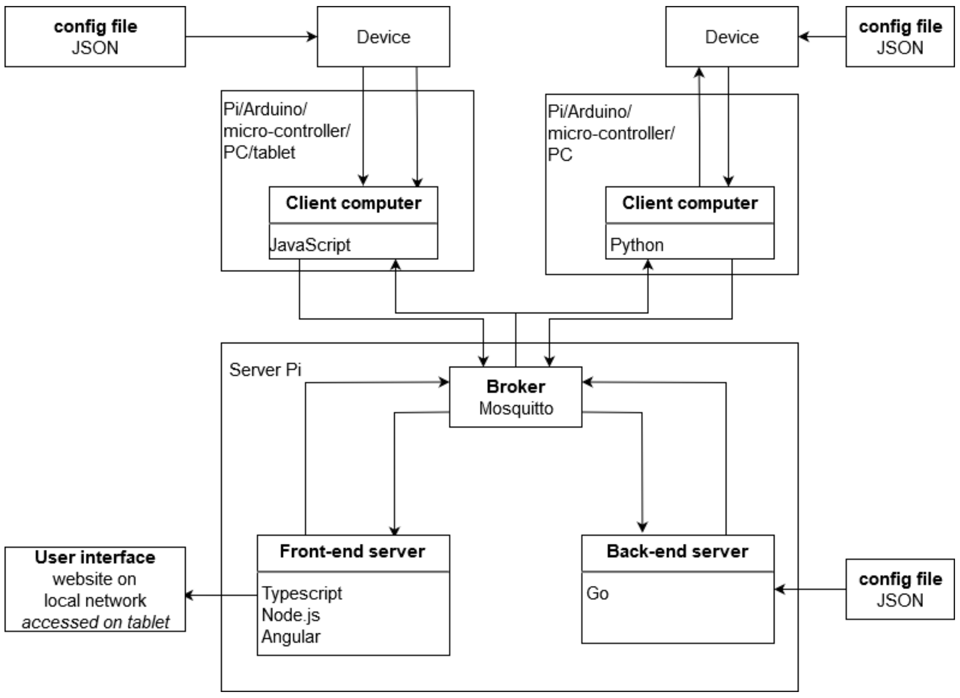
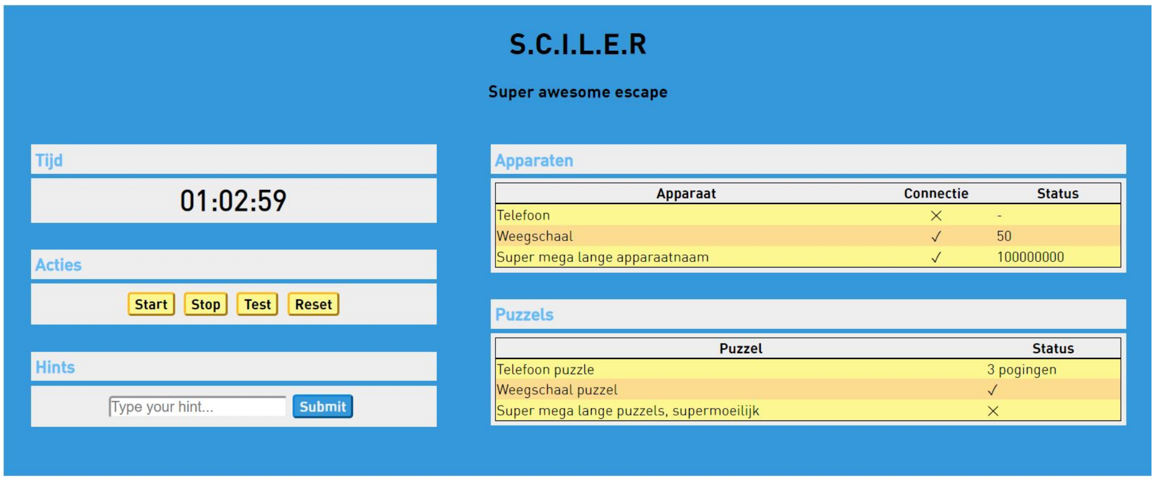


Figure 5.1: System structure, showing all the components in the system and how they interact together.

5.2. Front-end

The first design of the operator screen, seen in Figure 4.3, was implemented early on in development, which resulted in the home screen seen in Figure 5.2. Over the course of implementation, it was improved many times with newly added features and improved styling which looks more professional. Major improvements were marked by using the Material Design framework[2] for general styling and including a menu in the front-end to allow the operator to see the camera feed and enter a new configuration file, which is also checked in real-time. Because the team does not have a great feeling for styling, Material Design was integrated. It allows users to have consistent styling throughout their application by providing built-in types that can be used and ’fit together’ visually. This way, the team no longer had to think about which colours to use or how round to shape a button for it to look nice.


S.C.I.L.E.R

Super awesome escape

Tijd

01:02:59

Acties

Start Stop Test Reset

Hints

Type your hint... Submit

Apparaten

Apparaat	Connectie	Status
Telefoon	×	-
Weegschaal	✓	50
Super mega lange apparaatnaam	✓	100000000

Puzzels

Puzzel	Status
Telefoon puzzle	3 pogingen
Weegschaal puzzel	✓
Super mega lange puzzels, supermoeilijk	×

Figure 5.2: Front-end implementation in the first week of development

The full functionality included in the front-end, as it exists in the final version of the product, will now

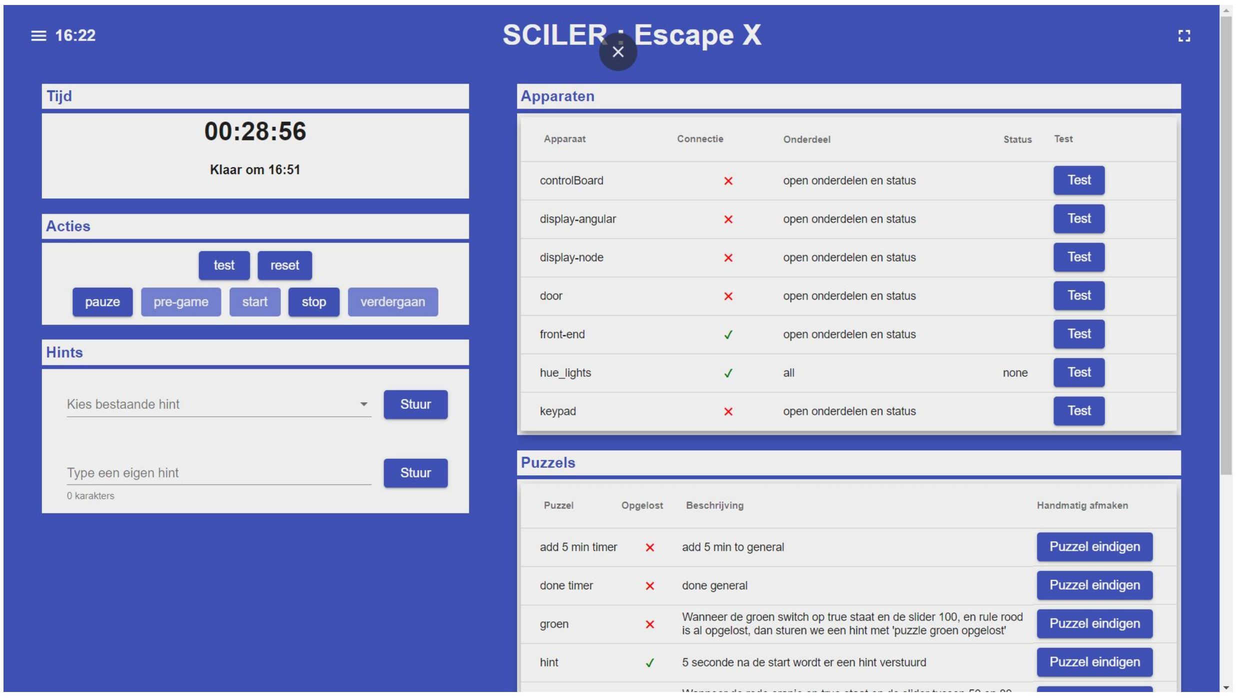
[2]`https://material.angular.io/`

be further explained. The operator GUI starts up on the home page. Most of the information shown comes from the configuration. All this information is communicated by the back-end and continuously updated. The operator GUI is completely in Dutch, as this was requested by the client. However, as most information is retrieved, this can still be in a different language, depending on the information in the configuration file. The Dutch names in the front-end are translated into English in this report.

All pages include a header in the top of the screen with the current local time, the name of the product, the name of the escape room that is currently running and a button to put the web page in full-screen mode.

5.2.1. Home page

The home page has five sections: Time, Actions, Hints, Devices and Puzzles, which are either shown in grid form or listed, depending on the size of the screen, see Figures 5.3, 5.4 and 5.5a. These are all styled in the same way, with a header box and contents box close together, to provide consistency across the user interface.


16:22

SCILER : Escape X

Tijd

00:28:56

Klaar om 16:51

Acties

Hints

Apparaten

Apparaat	Connectie	Onderdeel	Status	Test
controlBoard	✗	open onderdelen en status		Test
display-angular	✗	open onderdelen en status		Test
display-node	✗	open onderdelen en status		Test
door	✗	open onderdelen en status		Test
front-end	✓	open onderdelen en status		Test
hue_lights	✓	all	none	Test
keypad	✗	open onderdelen en status		Test

Puzzels

Puzzel	Opgelost	Beschrijving	Handmatig afmaken
add 5 min timer	✗	add 5 min to general	Puzzel eindigen
done timer	✗	done general	Puzzel eindigen
groen	✗	Wanneer de groen switch op true staat en de slider 100, en rule rood is al opgelost, dan sturen we een hint met 'puzzle groen opgelost'	Puzzel eindigen
hint	✓	5 seconde na de start wordt er een hint verstuurd	Puzzel eindigen

Figure 5.3: The home screen of the front-end on a big screen, game is started

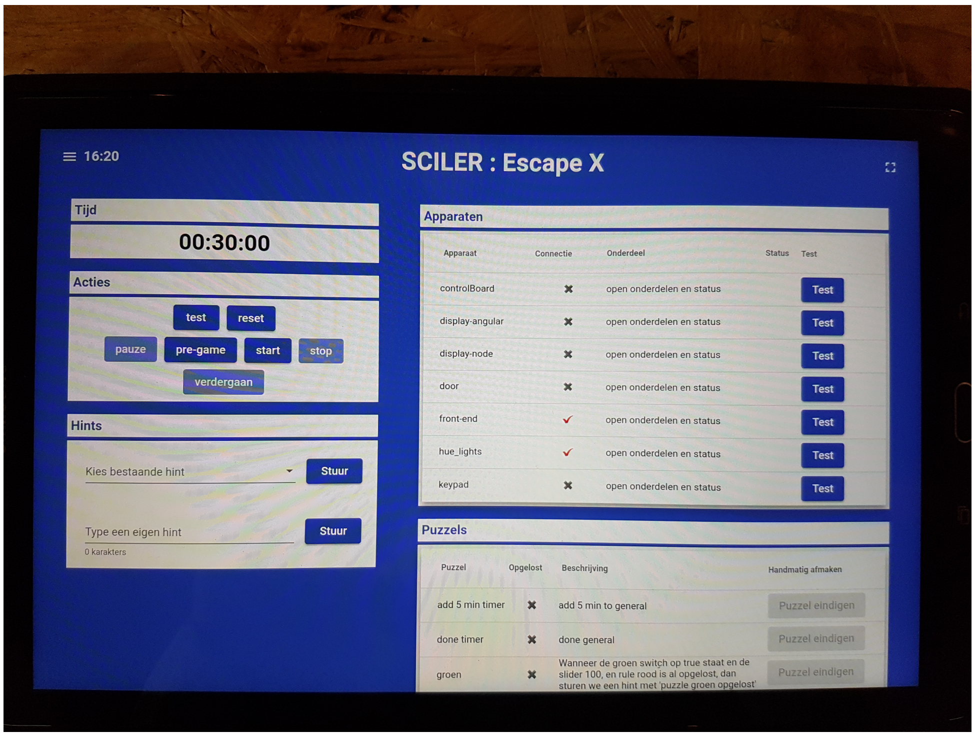
To start, the time is set to the duration of the escape room retrieved from the configuration and shown with hours, minutes and seconds. When the start button is pressed, the timer starts counting down to 0. Below the counter, the end time of the escape room will also appear. Milliseconds were considered but found to not be very important to the user, although they are still recorded in the code.

The action section contains all buttons to manage the escape room. There are two default buttons: test and reset. These instruct the back-end to let all client computers test their functionality or reset their status, respectively. Furthermore, other buttons can be generated from information in the configuration file. It is up to the user to implement these buttons, although, without implementing a start button, the game cannot be started. In Figure 5.3, the implemented buttons are start, stop, pause, continue and pre-game.

In the hints section, operators can send hints to the players in two ways: either by selecting a hint which is retrieved from the configuration or by typing a custom hint. When typing, the count will show the number of characters, though no limit is imposed. When selecting a pre-defined hint, the user can choose from hints that belong to a certain puzzle. The selection drop-down contains headers with puzzle names and below each, the hints that belong to that puzzle are shown.

Devices are displayed in the device table with their name, connection status and value status. The type of value status depends on the device and is shown per component of the device. These statuses are collapsed by default but can be extended by clicking a row in the table. This way, if a device contains many components, the whole screen is not immediately filled with this information. Finally, each row has a test device button, to only test that specific device.

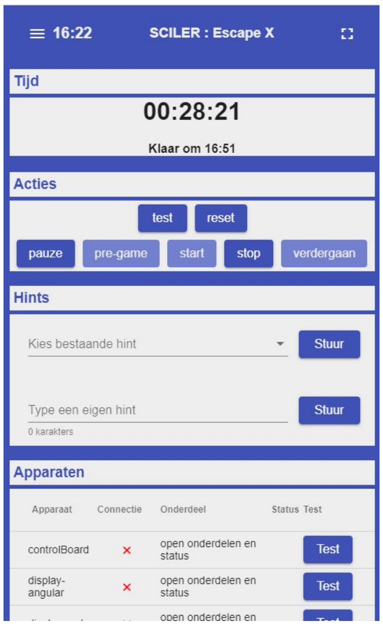
The final section of the front-end is the puzzle table, which contains a row for every rule in the configuration with their name, status, description and a button. The status lets the operator know whether or not this puzzle has been solved. This can be done either by the players solving the puzzle or the operator clicking the done button for that puzzle, which is only enabled. In either case, the actions belonging to that rule are executed.



Apparaat	Connectie	Onderdeel	Status	Test
controlBoard	✖	open onderdelen en status		Test
display-angular	✖	open onderdelen en status		Test
display-node	✖	open onderdelen en status		Test
door	✖	open onderdelen en status		Test
front-end	✓	open onderdelen en status		Test
hue_lights	✓	open onderdelen en status		Test
keypad	✖	open onderdelen en status		Test

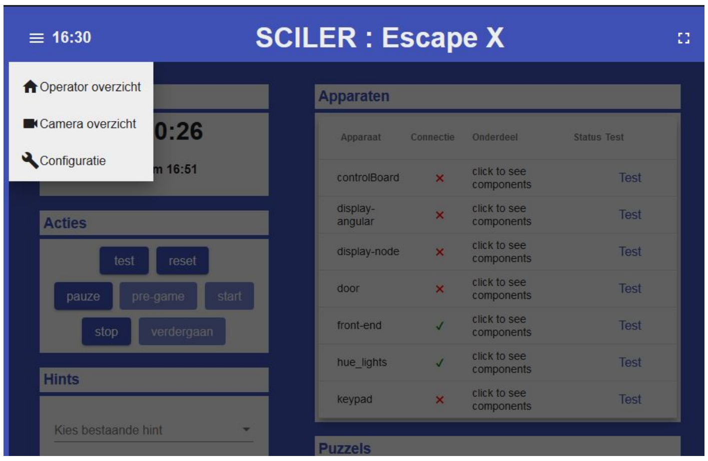
Puzzel	Opgelost	Beschrijving	Handmatig afmaken
add 5 min timer	✖	add 5 min to general	Puzzel eindigen
done timer	✖	done general	Puzzel eindigen
groen	✖	Wanneer de groen switch op true staat en de slider 100, en rule rood is al opgelost, dan sturen we een hint met 'puzzle groen opgelost'	Puzzel eindigen

Figure 5.4: The home screen of the front-end on a tablet device, game is not started



Apparaat	Connectie	Onderdeel	Status	Test
controlBoard	×	open onderdelen en status		Test
display-angular	×	open onderdelen en status		Test

(a) The home screen of the front-end on a mobile screen, game is started



Apparaat	Connectie	Onderdeel	Status	Test
controlBoard	×	click to see components		Test
display-angular	×	click to see components		Test
display-node	×	click to see components		Test
door	×	click to see components		Test
front-end	✓	click to see components		Test
hue_lights	✓	click to see components		Test
keypad	×	click to see components		Test

(b) The unfolded menu in the front-end to navigate the application

Figure 5.5: Screens of the home page on mobile and with menu

5.2.2. Camera page

When using the menu in the upper-left corner, see Figure 5.5b, the user can switch between the three screens of the application: home, camera and configuration. The camera page, see Figure 5.6, allows the user to see live feeds from cameras in the escape room, which are rendered from the IP address entered in the configuration. The user has the option to open a second camera screen, and both feeds can separately be selected from the drop-down menus. Finally, the user can open the camera in full screen, seen in Figure 5.7, which only shows the header and the IP address of the camera feed. The choice was made to not put the camera on the home screen, as this would fill up most of the screen if the video should be clearly visible.

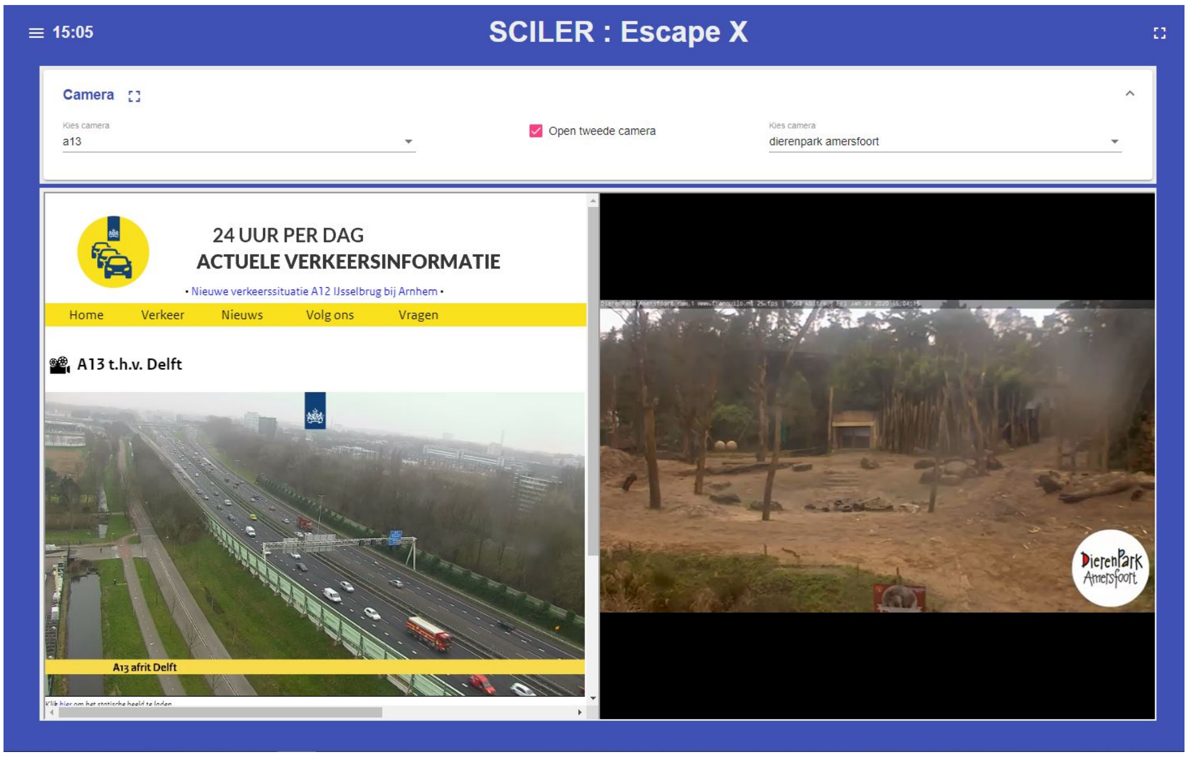


Figure 5.6: The camera screen of the front-end, with two feeds, which are currently websites

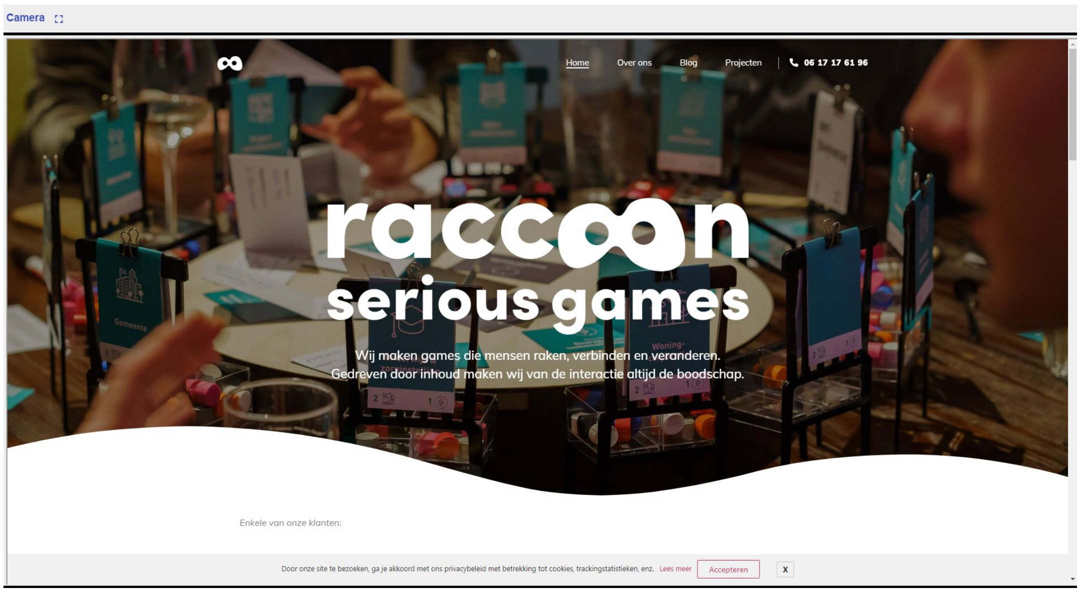
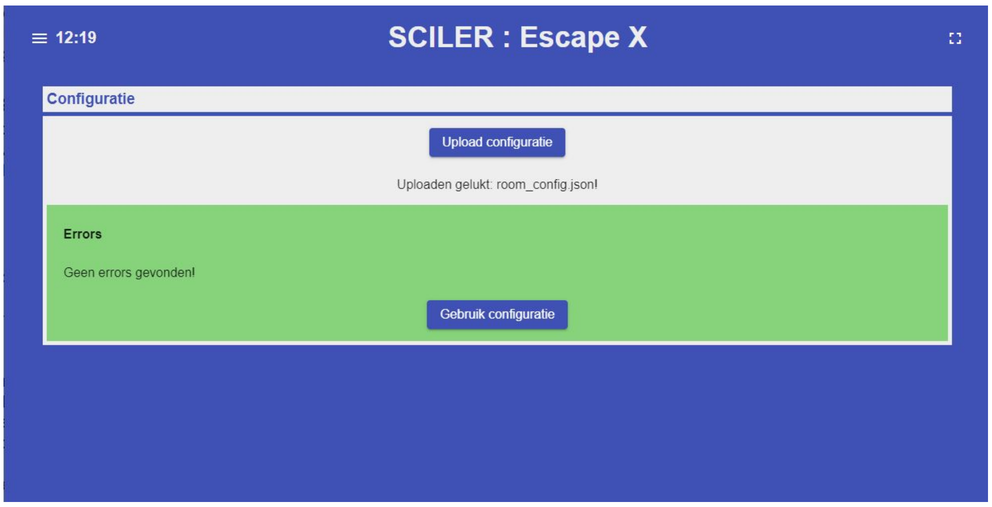

Camera
Home
Over ons
Blog
Projecten
06 17 17 61 96
raccoon
serious games
Wij maken games die mensen raken, verbinden en veranderen.
Gedreven door inhoud maken wij van de interactie altijd de boodschap.
Enkele van onze klanten:
Door onze site te bezoeken, ga je akkoord met ons privacybeleid met betrekking tot cookies, trackingstatistieken, enz. Lees meer
Accepteren
x

Figure 5.7: The camera screen of the front-end, with one feed in full screen mode. The feed is currently a website

5.2.3. Configuration

The final part of the front-end is the configuration page, where users can submit their configuration file, which is then checked for any errors. The errors are returned in four phases: JSON errors, formatting errors, detailed errors and system errors, because if there are errors in a previous phase, then the next part of the configuration examination cannot be executed. Only for system errors can two levels be shown, as these are only based on the host in the configuration, whether it matches that of the current configuration. These errors are also formed in the `MessageHandler`, instead of all other errors, which are generated from the `ConfigHandler`. Furthermore, an expansion panel with information about the different types of errors appears on the screen with the errors.

If no errors are found, then a button appears, so the user can restart the system with the configuration, after which the user will be redirected to the home screen. Screenshots of the configuration page can be seen in Figures 5.8 and 5.9, both with errors and with no errors found.

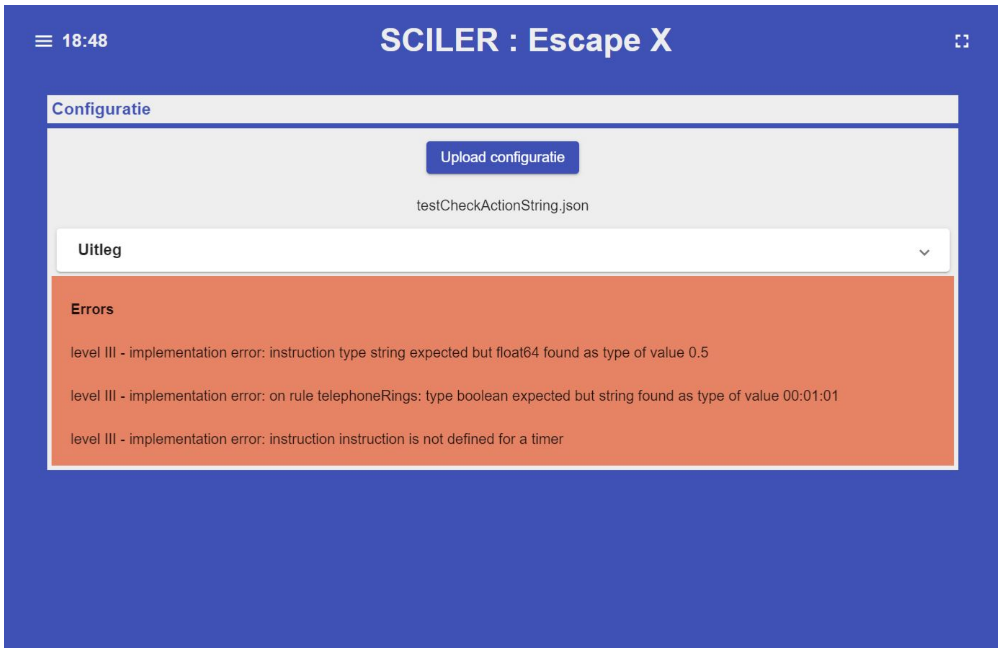


Uploaden gelukt: room_config.json!

Errors

Geen errors gevonden!

Figure 5.8: The configuration screen of the front-end, when no errors are found



testCheckActionString.json

Errors

level III - implementation error: instruction type string expected but float64 found as type of value 0.5

level III - implementation error: on rule telephoneRings: type boolean expected but string found as type of value 00:01:01

level III - implementation error: instruction instruction is not defined for a timer

Figure 5.9: The configuration screen of the front-end, when errors are found in stage three and error explanation has been collapsed.

5.3. Back-end

The back-end is implemented to handle all communication with the devices and is fully powered by the configuration file. The back-end is implemented in Go and contains three modules: `config`, `communication` and `handler`, as well as a main file. The configuration file design is updated to contain all the information the back-end ended up needing to properly function, see Section 5.3.1.

The resulting back-end implementation is an efficient system thanks to the thorough checking of the configuration and generation of lookup maps at startup.

5.3.1. Configuration

The `config` module contains a `ConfigHandler` and supporting types which make up the full `config`. It starts by loading the JSON file into the Go struct called `ReadConfig`, see the UML structure in Figure 5.10. Then, the data is processed to create a `WorkingConfig` Go struct, see the UML structure in Figure 5.11, which is the object that the rest of the back-end depends on. This process mostly focuses on checking the data in terms of proper data types and necessary information. The UML's can also be found in full size in Appendix A.

Configuration structure The configuration was updated compared to the design (described in Section 4.1.4) to contain, besides the general info, devices, puzzles and general events, also the following attributes:

- `cameras`: the cameras that exist in the room, with their name and the link to their feed
- `timers`: the different timers that are used in the escape room which are used by different puzzles or events
- `button_events`: these are events that are triggered by a button in the front-end

The output of a component of a device has also been updated to contain all information needed to process and type-check the inserted configuration.

A manual including visualisation and example configuration can be found in Appendix D and E, respectively.

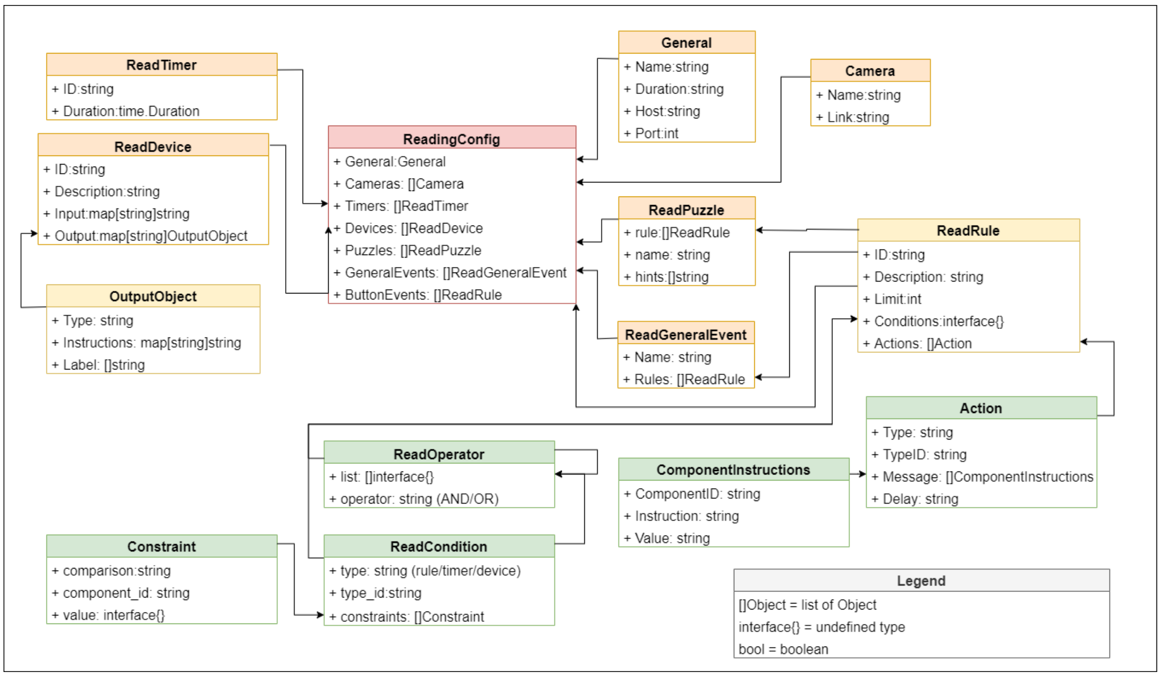


Figure 5.10: UML of the `ReadConfig` object after the first type checking.

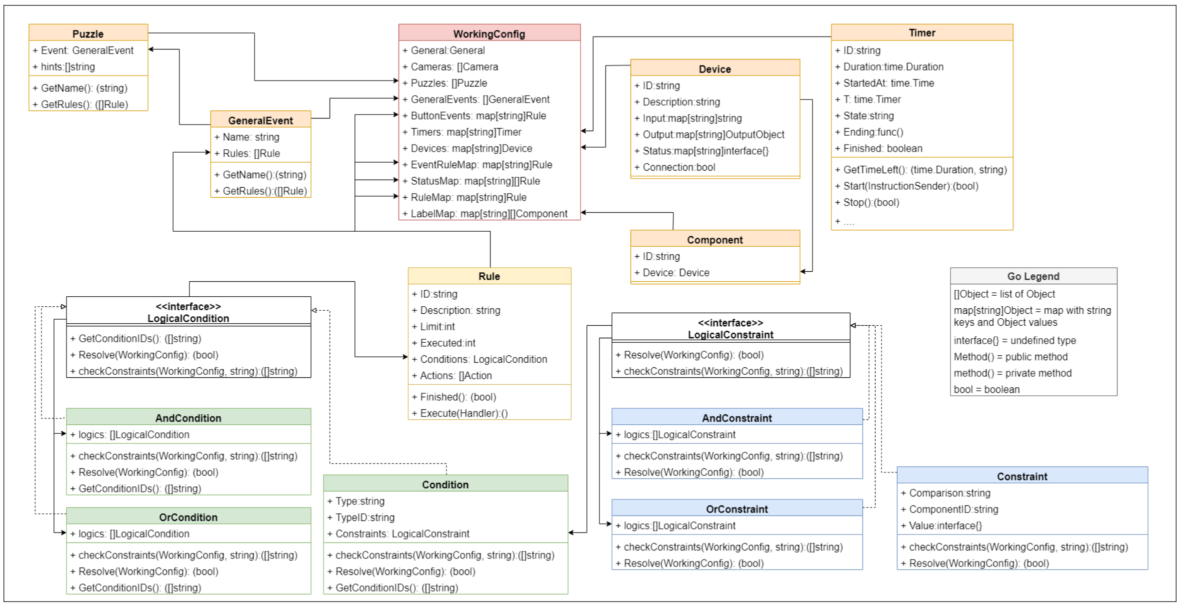

Puzzle
+ Event: GeneralEvent
+ hints:[]string
+ GetName(): (string)
+ GetRules(): ([]Rule)
GeneralEvent
+ Name: string
+ Rules: []Rule
+ GetName():(string)
+ GetRules():([]Rule)
WorkingConfig
+ General:General
+ Cameras: []Camera
+ Puzzles: []Puzzle
+ GeneralEvents: []GeneralEvent
+ ButtonEvents: map[string]Rule
+ Timers: map[string]Timer
+ Devices: map[string]Device
+ EventRuleMap: map[string]Rule
+ StatusMap: map[string][]Rule
+ RuleMap: map[string]Rule
+ LabelMap: map[string][]Component
Device
+ ID:string
+ Description:string
+ Input:map[string]string
+ Output:map[string]OutputObject
+ Status:map[string]interface{}
+ Connection:bool
Timer
+ ID:string
+ Duration:time.Duration
+ StartedAt: time.Time
+ T: time.Timer
+ State:string
+ Ending:func()
+ Finished: boolean
+ GetTimeLeft(): (time.Duration, string)
+ Start(InstructionSender):(bool)
+ Stop():(bool)
+ ....
Component
+ ID:string
+ Device: Device
Rule
+ ID:string
+ Description: string
+ Limit:int
+ Executed:int
+ Conditions: LogicalCondition
+ Actions: []Action
+ Finished(): (bool)
+ Execute(Handler):()
Go Legend
[]Object = list of Object
map[string]Object = map with string keys and Object values
interface{} = undefined type
Method() = public method
method() = private method
bool = boolean
<<interface>>
LogicalCondition
+ GetConditionIDs(): ([]string)
+ Resolve(WorkingConfig): (bool)
+ checkConstraints(WorkingConfig, string):([]string)
<<interface>>
LogicalConstraint
+ Resolve(WorkingConfig): (bool)
+ checkConstraints(WorkingConfig, string):([]string)
AndCondition
+ logics: []LogicalCondition
+ checkConstraints(WorkingConfig, string):([]string)
+ Resolve(WorkingConfig): (bool)
+ GetConditionIDs(): ([]string)
OrCondition
+ logics: []LogicalCondition
+ checkConstraints(WorkingConfig, string):([]string)
+ Resolve(WorkingConfig): (bool)
+ GetConditionIDs(): ([]string)
Condition
+ Type:string
+ TypeID:string
+ Constraints: LogicalConstraint
+ checkConstraints(WorkingConfig, string):([]string)
+ Resolve(WorkingConfig): (bool)
+ GetConditionIDs(): ([]string)
AndConstraint
+ logics:[]LogicalConstraint
+ checkConstraints(WorkingConfig, string):([]string)
+ Resolve(WorkingConfig): (bool)
OrConstraint
+ logics:[]LogicalConstraint
+ checkConstraints(WorkingConfig, string):([]string)
+ Resolve(WorkingConfig): (bool)
Constraint
+ Comparison:string
+ ComponentID:string
+ Value:interface{}
+ checkConstraints(WorkingConfig, string):([]string)
+ Resolve(WorkingConfig): (bool)

Figure 5.11: UML of the processed configuration, showing the class hierarchy used to work with. Some Go stucts are missing, these are already in 5.10

Process of loading the configuration in the system The first process when running S.C.I.L.E.R. is reading and checking the configuration file created by the user to generate a `WorkingConfig` datatype. As this is a critical part of the system without which S.C.I.L.E.R. would be useless, all errors that occur in the handling of the system's default file cause the system to shut down. It will display any errors thrown in its console and a locally created file.

This process starts by parsing the JSON object into a `ReadConfig` object. This is a process done by an external JSON library. In this stage, only JSON errors can occur, based on the tags that the `ReadConfig` object expects and formatting errors, see Section 5.2.3, are handled.

Next, the `ReadConfig` object is used to generate the `WorkingConfig` object. The `WorkingConfig` has additional data structures to keep track of the data but to put all the data in, it first needs to be checked. Therefore, the `ReadConfig` is first created directly from the JSON file, and only afterwards is the `WorkingConfig` created.

In this stage, all objects of the UML in Figure 5.11 are generated by first checking if all types are valid. This can render formatting errors, which cause the system to break because the necessary information is missing. During this process, the default objects are also created. These are objects which exist in every escape room, regardless of the configuration. The default objects are a `front-end Device`, with a `gameState` output component and `set state` instruction; and a `general Timer`, to manage gameplay.

When the data is converted to the `WorkingConfig` structure, extra data maps are created. These can only be created based on properly structured information and the process will generate implementation errors, as this depends on references to other parts of the configuration. Type-checking is a very important part of the process since it ensures that once the configuration is processed, no errors should occur during run-time.

Constraints are checked whether the specified component exists in the corresponding `Device` object. Furthermore, the comparison is checked to be relevant for the component's type. Finally, the constraint's type is checked to be correct corresponding to the component's. This is done by a lookup for the type of component in the `Input` map of the `Device` object involved in the constraint and the value type of the constraint itself.

Actions need to be checked by comparing the instruction for a component with its list of possible instructions. This is done by a lookup for the possible instructions and values of an `OutputObject` in the `Output` map of the `Device` concerning the action. The instruction and value are retrieved from the `Component-Instruction` of the action itself.

The operator will not be able to start the system if the configuration it uses contains errors. Therefore, the system will output these errors in its logs when it handled a configuration containing invalid conditions and actions that are not compatible with the implemented devices defined in the `Device` component of the configuration.

The generated maps in the `WorkingConfig` are used to save pointers for a quick lookup between related devices, components, labels, events and rules. For example, the `WorkingConfig` searches all rules containing a condition on a specific device only once, and saves these `Rule` pointers under key `Device.ÍD` in the `StatusMap`. Then, when all rules regarding a device need to be checked, `StatusMap[Device.ID]` can return these rules immediately instead of searching through all rules again.

After this process, the system is set up and ready to run the escape room.

Usage The JSON configuration file should be written to include all parameters mentioned in the manual found in Appendix D. When seen for the first time, it can look like a complex structure with a lot of different parameters. However, more experience can be gained and the manual and configuration checker in the front-end help users construct a proper configuration file.

One of the developers wrote a configuration file for one of the escape rooms that will be hosted by Raccoon Serious Games in February. She, of course, has a lot of experience with the system and knows the configuration structure well, but the contents were very new to her. It took her half a day to write the configuration for the first version, which was updated in a couple of minutes every time a puzzle was worked out in more detail by Raccoon Serious Games. So, for someone with less experience with the structure, but more knowledge and understanding of the contents, the time required to write an initial configuration file should be similar.

The format of the configuration was thought out thoroughly in the design phase, see Section 4.3.2. When implementing the `ConfigHandler`, there were some struggles with how to properly put the information from the file into objects that the back-end can handle. Therefore, some additions and changes were made, as was just explained.

However, in the end, the team is satisfied with the format chosen with the addition of a couple of components, as it allows for all possibilities of conditions and actions to be implemented. During the final weeks of development, the client gave the team information about the escape rooms they will be running in February and there was nothing that could not be implemented with S.C.I.L.E.R..

5.3.2. Communication

The `communication` module handles all `communication` with the broker. It contains one main `Communicator` object which has all the methods necessary to set up the connection to the broker, reconnect when necessary, startup its communication and publish messages at its disposal.

In the design phase, the structure of message handling was not thought out as much as the configuration, apart from the actual messages that could be sent. When implementing the back-end, the developer chose to separate the `Communicator` from the `MessageHandler`. The `Communicator` uses the MQTT library to do the actual communication and the `MessageHandler` only looks at received messages and handles them based on their content. This functionality was split, because that would keep the MQTT messaging separate from the custom S.C.I.L.E.R. implementation of the message handling.

On startup, the system uses the host and port given in the configuration file to set up an MQTT client to connect to the broker. Functions in the `communication` module set all the needed options for the MQTT client to work properly. The client subscribes to all messages for topic `back-end` and specifies that all these messages should be handled by the implemented `MessageHandler`.

The `MessageHandler`, implemented in the `handler` module, processes all messages through the S.C.I.L.E.R. system. It divides its contents up in the `msgMapper`, which, depending on the type of message received, calls the appropriate helper methods. Appendix C contains the message manual specifying all the messages that are sent within the system between the different components.

Instruction messages Instruction messages are messages received from the front-end, and the possible instructions are:

- `send setup`: The back-end will send a message to the front-end, containing all information from the configuration that will not be updated throughout the run.
- `reset all`: The back-end will set its `WorkingConfig` back to its initial state by reading in its configuration file again, and the back-end sends all devices an instruction message to reset itself.
- `test all`, the back-end will send all devices an instruction message to execute its test protocol.

- `test device`: The back-end will send the device specified in the massage an instruction message to execute its test protocol.
- `finish rule`: The back-end will call execute all actions on the rule provided in the message regardless of its conditions.
- `hint`: The back-end will send the hint instruction and value to the topic `hint` for all devices subscribed to this topic to execute the hint protocol.
- `check config`: The back-end will check the configuration provided in the message as described in section 5.2.3
- `use config`: The back-end will use the configuration provided in the message to reset the S.C.I.L.E.R. system according to the new configuration by going through the configuration process again, which is described in the previous section.

Connection messages Upon a received connection message from a client computer, the back-end updates the connection status of the `Device` in the `WorkingConfig` according to the value of the message. This new status of the `Device` is then sent to the front-end.

Confirmation messages Confirmation messages are sent by client computers which performed an instruction. The message is forwarded to the front-end if that was the original instructor.

Status messages After receiving a status message from a client computer, the handler will first update the status data of the corresponding `Device` in the `WorkingConfig` accordingly. This is done only when the components and its types correspond to the components in the device's `Input` or `Output` maps. This new status is then sent to the front-end.

Next, it checks conditions of the rules which involve the device from which the status was received. The system quickly finds the rules containing conditions with this device via the `StatusMap`. All rules in this set are checked one by one on its `Limit` and its `Conditions`, in that order. This is not done in parallel since the status of a rule can influence the `Resolve` result of the conditions of another rule.

If the `Rule.Executed` is smaller than the `Rule.Limit`, then `Resolve` is called on `Rule.Conditions`. `Conditions` is a logic tree of conditions with constraints as leafs, as shown in Figure 5.11. Branches are split with an operator `AND` or `OR`, and the tree's leaves will put their result in a boolean value. Constraints are evaluated in a somewhat lazy matter, as each constraint is looped over, but if an `AND` operator has already seen a false result, the constraints that are evaluated afterwards will no longer be resolved. Finally, `Resolve(Rule.Conditions)` returns a boolean value, and if this is false, nothing happens. When the `Rule.Executed` is smaller then `Rule.Limit`, and `Resolve(Rule.Conditions)` returns true, that rule's actions are executed. This way, the implementation of the event handling, still follows the methods of Algorithm 1, as described in Section 4.1.4.

Actions of a `Rule` are executed in parallel since these do not influence each other. This is implemented using a goroutine, which is built-in Golang functionality to execute parallel instructions [6]. An action can contain instructions for devices, timers or labels. An action can have a delay, in this case, the action will send its corresponding messages after this time has passed.

An action instruction for a device is sent to the appropriate client computer, without checking its contents again. Because the contents have already been type-checked when the `WorkingConfig` was generated, this can happen safely. An action for a timer is processed in the back-end, where all the timers are managed.

When an action with a label as its target is processed, the instructions will be sent to all client computers of which the device contains a component with that label. First, these devices are found using the generated `LabelMap`, which has label keys and lists of its components as values. Then, all detected client computers receive separate messages for each instruction in the list of `action.Instructions`. Because the label instructions do not contain `component_id`'s, these need to be retrieved from the configuration and added to the `ComponentInstruction` messages before they can be sent. If in a later stage of development (see also Section 10.1), S.C.I.L.E.R. is to be optimized to reduce the number of messages sent, the label instructions would be an implementation to refactor by already setting the components in the `WorkingConfig`, for example.

The last steps of processing status messages are sending the event status to the front-end, as these might have updated if a rule's conditions were met and it was executed; and the sending of the front-end's button status, which tell the front-end which buttons should be disabled based on the current `gameState`, which is the front-end's output status.

5.4. Client computers

The implementation of the client computer library was done early on in the development process, with only minor improvements added later on. The design had already been thought out thoroughly in advance, and the team stuck to the design closely, with some improvements to communication and instruction handling.

5.4.1. Implementation Python library

The abstract device class forces the devices to implement the methods: `main`, `test`, `reset`, `get_status` and `perform_instruction`. By forcing these methods, the user cannot forget to implement a certain function, thereby crashing the system. The `main` method of the custom class should be implemented to set up the different components of a device and call the device's `start` method. The implementation of the `test` and `reset` method is required to ensure that each device can be tested and reset on instruction from the operator. The `get_status` method returns the status of the device, containing all components and properly formats their status. This method is used to update the back-end of the status, on which events might be executed. Finally, `perform_instruction` handles an instruction, not being `test` or `reset`, but custom to the device. This allows the client computer to implement functionality for showing a hint or setting the lights, for example.

Furthermore, the library provides all devices with a `log`, `start` and `status_changed` method. `log` allows devices to log messages about what they receive, do and what might go wrong. `start` is called from the `main` method and starts the device. `start` allows for a loop function to be defined which will then be used instead of the built-in loop in the library. This is done to better facilitate asynchronous behaviour in client computers. `status_changed` sends a new status message to the back-end, which is implemented through the application file in the library.

5.4.2. JavaScript Library

After finishing the minimal working product, the client indicated that besides the Python library, which works well with Pis, he also wanted a JavaScript library to control devices which are written in JavaScript, for example, a website. So, a JavaScript version of the library was implemented. The design was kept mostly the same, with both libraries having an abstract device class which should be implemented.

The difference with the JavaScript version is the absence of the `main` method and a different `start` method implementation. Since the JavaScript library runs in the browser, there is no need for a loop to keep the script running. The `start` method in this version of the library takes a callback function which will be called once a connection is established by the `start` method. This version of the library also handles logging differently from its Python counterpart. Since the JavaScript library runs in the browser, writing logs to a file was not possible. Besides, it would be better to store these logs on the server which serves the web page, instead of the client storing them. For implementing a device, a logger function must now be defined which will be called every time something has to be logged. This way, a developer can choose to only print to the console, to send these logs to a server which stores them or to do both.

5.4.3. Library availability's

Finally, both library implementations were published to the package installers for the respective languages. The JavaScript library is published as `sciler` and can be found on npm [3]. The Python library is published as `sciler` and can be found on pip [4]. This way they can easily be imported for any device that will implement the library.

Programmers working for Raccoon Serious Games have already implemented client computers themselves, using the published libraries. Once they studied the manual for client computers, they were able to implement the devices to behave as expected. Furthermore, the client has already tried out running S.C.I.L.E.R. with self-implemented client computers, and this worked according to expectations. So, the library appears to provide all the functionality needed for client computers to communicate with the system and Raccoon Serious Games can perform all actions as they need to for their escape rooms.

6 Testing and quality control

Testing is an important part of every software product, to ensure a well-working system. This chapter will explore all the testing done over the course of development and how quality control is ensured both during the process and in the future.

As mentioned in the research in Chapter 3, some tests were thought of based on the research performed. So besides the automated testing, testing was also performed on the system, the latency testing and a user test was done.

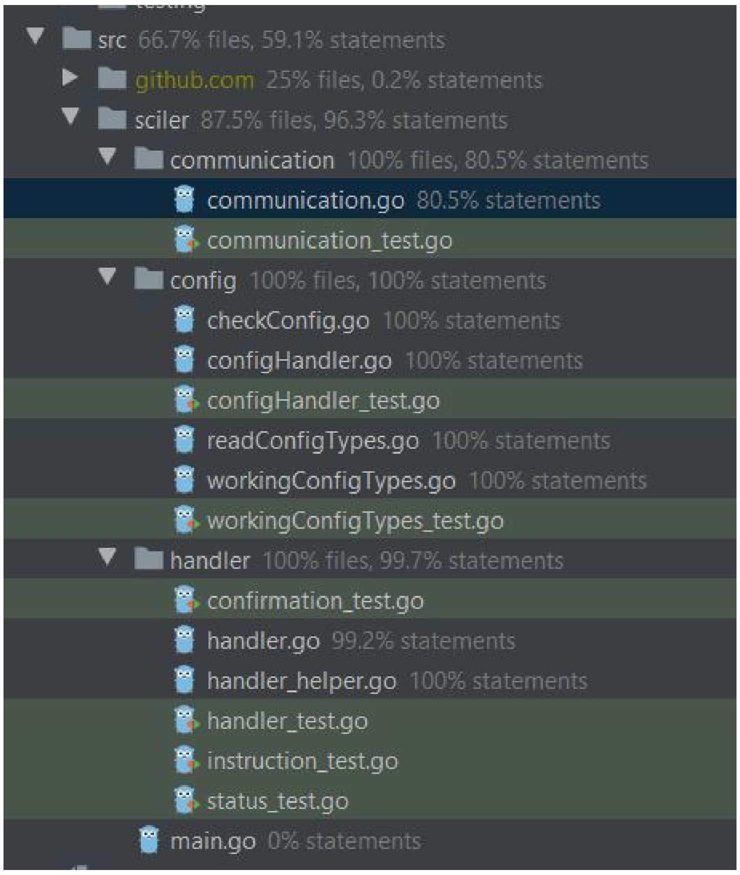
6.1. Automated testing

As there are three components, they each should be tested individually. However, for the client computer library, this turned out impractical. The library consists of an abstract device class and an application. The library is tested manually but does not have any automated testing, because this did not receive a high priority, so was not finished in time. Looking back, this should have received a higher priority, because creating a test where an implementation of the abstract device is tested and the MQTT communication is mocked, would have been doable. Initially, a test file was written which allows the user to test a device implementation whether it fulfils all requirements of the implementation. This test file can be run manually, by copying the test file to a local repository where the implemented Device class is located. Then, the name of the device in the test file should be changed to the correct Device class. Unfortunately, this test was not updated throughout the project, so it is not fully accurate anymore. The test file is still present in the Python library, to be updated after the project deadline.

For the front-end, the Angular framework offers support for browser testing, using Karma[1] and Jasmine[2]. At the beginning of development, tests were created to check whether all parts of the front-end were present. However, as the front-end was further implemented, the hardcoded data in the front-end was removed and all data is now retrieved from the back-end. Therefore, the tests were no longer accurate, as Jasmine tests are not in connection to a broker. Although an MQTT client could be mocked to test whether data is sent correctly, the team did not succeed in mocking data that is displayed in the front-end. So, the Jasmine tests only remain to check that components are initialized properly. This could be improved in the future when studying the Jasmine framework extensively, more on this in Chapter 10.

The back-end is fully unit tested. As most functionality of the back-end comes from the configuration and message handling, this can easily be tested. For the configuration handler, all possible scenarios are tested to make sure that any errors in the configuration file are caught and reported. The message handler also tests all possible message that it can receive, and what messages are published based on the received message. To do so, the MQTT client is mocked, which checks what messages it sends and how many are sent. The test coverage of the back-end can be found in Figure 6.1, which is generated from the JetBrains GoLand[3] coverage, using statement coverage. This can be run locally in GoLand with the `go test sciler` command using coverage, however, as the Testify framework does not support integration with GitHub, there is no badge in the `ReadMe`.

[1]`https://karma-runner.github.io/latest/index.html`
[2]`https://jasmine.github.io/`
[3]`https://www.jetbrains.com/go/`



```
▼ src 66.7% files, 59.1% statements
  ▶ github.com 25% files, 0.2% statements
  ▼ sciler 87.5% files, 96.3% statements
    ▼ communication 100% files, 80.5% statements
        communication.go 80.5% statements
        communication_test.go
    ▼ config 100% files, 100% statements
        checkConfig.go 100% statements
        configHandler.go 100% statements
        configHandler_test.go
        readConfigTypes.go 100% statements
        workingConfigTypes.go 100% statements
        workingConfigTypes_test.go
    ▼ handler 100% files, 99.7% statements
        confirmation_test.go
        handler.go 99.2% statements
        handler_helper.go 100% statements
        handler_test.go
        instruction_test.go
        status_test.go
      main.go 0% statements
```

Figure 6.1: Back-end test coverage

6.2. System testing

After finishing the minimal working product for the midterm meeting, the team also wanted to start testing the system running from a Raspberry Pi. This was set up on a model 3 Pi and the system was run for longer periods and analysed whether it worked sufficiently and according to expectations. A few crashes were noted, which could all be explained, as this usually meant that two back-ends were running at the same time, which the system does not allow. Furthermore, a bug was found in the reconnecting of clients to the broker upon a disconnect, so this was resolved.

Since the start of the project, the team realised that system testing would be needed to automate the testing of the full system. Although the issue was on the issue board since week 2 of development and it was given some thought throughout the project, it was not implemented till week 7. After writing scripts for the Raspberry Pi to automatically start up a broker, back- and front-end, it became easier to grasp the idea of a bash script that could start a test device, broker, back- and front-end on localhost and write some basic tests for the communication within the system. In the end, only a latency test was written, which does test the full system setup, but is very basic in its messages and is not in the format of a bash script. More on this in the next section.

6.3. Latency testing

The latency of the communication in the system should be tested to be below the threshold of 100 milliseconds, which was set after researching, see Section 3.6. By starting a broker, back-end and device, and sending an instruction to the device, a confirmation message should be sent back. The time it takes before receiving the confirmation should then be below 100 ms, plus maybe additional time for executing the instruction.

A test was designed where a special instruction was sent a hundred times to a client computer which was configured to update its status and then send its status. Whenever an instruction was sent, the time was recorded. When the corresponding status update was received, the latency was calculated by taking the difference between the current time and the time of sending the instruction. Then, the test checked if the latency of every single message was within the threshold and an average was calculated. The result of the test was an average of 66.4 ms and all individual messages stayed below the 100ms. This test was performed over WiFi, however, it was not done on the (new) router that will be used in the upcoming escape room. The team expects that this latency can be decreased further by using the new router. For the full result see Figure 6.2.

```
100 messages were sent with a mean latency of 66.37 ms
with a minimal latency of 11 ms
with a maximal latency of 95 ms
--- PASS: TestLatency (6.65s)
PASS
```

Figure 6.2: Latency test results.

6.4. User testing

Because the system should be user-friendly and easy to handle, the user interface was tested. The set-up and results of the tests are described here. The test was drawn up to be taken by a member of the Raccoon Serious Games team, who has operated escape rooms before and will also be using S.C.I.L.E.R. in the future.

An instance of the system was started up on the local network. A configuration was written that included some basic elements and a couple of devices were set up to facilitate a working system. It included hue lights, that were set with different values for pre-game, start, pause, stop and puzzles solved; a control board that was used to create a small puzzle; and a display screen to show hints. Furthermore, the system ran from the server Raspberry Pi and the tester was given a tablet for operating. Finally, a script was written for what the tester was expected to do, this can be found in Appendix B. Afterwards, the following questions will be discussed:

- Could you find why the room was not ready to start?
- Was it intuitive to find out what you had to do to prepare and start?
- Is it clear how you can test?
- Is it clear how the status works?
- Is it clear that you can override a rule to be executed?
- Is it clear that you can switch screens?
- Do the buttons perform as expected?
- Do you recommend any optimisations for use on a tablet?

Results During the test, the team made notes of the tester's comments. Not all questions were answered specifically, but the general takeaway from the test is described here.

When doing the test, the tester immediately noted a lot of things about the design, mainly the colours, button forms and organisation of the home screen. A lot of questions arose when she tried to prepare the game for playing, and also during play, the interface was not clear. These are most important things that were noted: the puzzle overview was very confusing; there was no consistency in the use of buttons; there was little consistency between the different pages of the application; the home screen was not practical with a lot of scrolling.

The tester herself is a designer, so she has a good eye for an efficient and consistent design. She pointed out many points that she would see differently and told the developers her view of why a certain and element should exist and what she would then expect from that element. This was very helpful advice. The team took notes of all her tips and comments and will improve the front-end based on that. These changes will be evaluated on how they can be implemented and whether they should be, then the useful improvements will be developed for the demo given in the presentation of the project, so after this report has been submitted, see Chapter 10. This will also be done to benefit Raccoon Serious Games, as they will put the system to use shortly thereafter.

6.5. Quality control during development

During development, several tools and rules were used to guarantee quality control. This included the use of GitHub merge requests, code coverage testing, automated testing, code formatting tools and a deployment tool.

The master and development branches on the GitHub repository were protected against direct pushing, to force the team to only submit code that would first be reviewed. All code was subjected to a full code review, consisting of both reviewing the code changes and testing the system. This was always done by at least one, but usually two team members.

Code formatting tools were used for all three components of the system, to ensure consistency throughout the codebase, following rules set at the beginning of the project, like white space use and line lengths. Furthermore, the code was tested, as described in the sections above.

Besides manual checking, Travis was used as a deployment tool and was configured to always run the code style checker and testing framework per component. Additionally, the system test bash file can also be added to Travis.

Finally, the team submitted their codebase to SIG[4] twice, both after four weeks of development as well as right before submitting their codebase to the university. The Software Improvement Group checked the code on a matter of issues, like method complexity and length. The complete SIG feedback and actions taken can be read in Appendix F. In the last weeks of development, a SIG check was also added to the merge request checks, so the team would automatically be noted if their new code broke any SIG rules. If the rules are broken, it means the code is probably too complex or long to understand can, so it should be fixed to be more understandable. This a helpful check to keep a proper codebase. The final SIG results run on the master branch can be found in Figure 6.3, and the `ReadMe` of the GitHub repository contains a SIG badge.

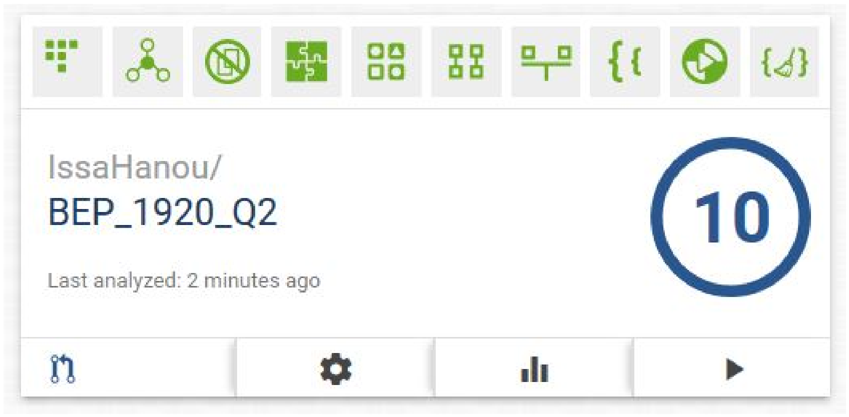


Figure 6.3: SIG results, the 10 guidelines that are checked and the final score of the codebase against these guidelines.

6.6. Quality control for the future

To ensure that the codebase will still be of high quality in the future, the team provided the codebase with documentation about the testing and tools used. This way, other developers can still use these tools to guarantee that newly added code will also be up to the standards of the codebase as it is left by the team. Furthermore, the GitHub provides checks for Travis (including testing and code style) and SIG to make sure that newly written code will still be of good quality.

A manual will be provided for writing bash files for system testing, so these can also be created in the future to automate more functionality testing.

After delivering the product to the client, it will quickly be put to use to run two escape rooms in February of 2020. Test runs will already be run at the end of the project, and the team will be involved for a few weeks after handing in their product, to help out with setting up the system for these specific escape rooms, assisting the team of Raccoon Serious Games to learn how to use S.C.I.L.E.R. and solve any bugs that might occur.

After these few weeks, when Raccoon Serious Games is more comfortable in using the system, the team will ensure that the developed escape rooms can run smoothly and then the product will be fully finished.

[4]`https://bettercodehub.com`

Furthermore, any features or improvements requested by the client will be quickly resolved by the team to provide a product meeting all the client's wishes.

7 Ethics

This chapter will discuss the ethical implications of the S.C.I.L.E.R. system.

S.C.I.L.E.R. was developed for Raccoon Serious Games, who will use it in their escape rooms. They also keep the rights to sell the product to other parties. The developers are still allowed access to their developed code base after finishing the project and can use it for personal purposes. However, they will not be allowed to sell the software.

The system runs from a configuration file. The only other data the system contains by default is the name of the product, S.C.I.L.E.R.. So, all the data used by the system is generated from a file, which will be written by the user. Whether or not to include any sensitive information is thereby completely up to the user.

Furthermore, all devices have their implementation of the client computer library, based on their configuration file. Therefore, again, all the data is written by the user.

Although configuration files are saved, these only contain information about the escape room. No data of the players is saved, although the system can be extended to save game data, like the time it took to escape, hints that were given and possible wrong answers that were given in the process. This data could then be saved under a team name, thereby not implicating any personal data.

Besides the escape room configuration data, the user interface also allows the user to see camera feeds. The links to the feeds are retrieved from the configuration, so are controlled by the user. Furthermore, the system does not save any of this data, it only streams the feed from the camera into an iframe.

Finally, as the system runs on a local network and uses that connection to communicate between devices, it is only possible to intercept any data that is being sent, when the intruder is in the network itself. The network is secured on its own, so the only way to break in is by hacking the router. By using a secure router to set up the network, there are no risks to the system or of data leaks. The network does not have any special protection, as this was not a requirement of the client.

The only personal information being sent is the camera feed, of which the link is only sent once. The rest of the information is only sensitive to Raccoon Serious Games in the sense that when it would get out, it could ruin the experience for other people, because it contains information about how to solve the escape room. However, when checking or loading a new configuration through the front-end, the complete JSON configuration file is also sent over the network. So, by ensuring a secure network, this data will remain safe.

So, S.C.I.L.E.R. has few ethical implications, as it does not handle personal data of the players or the user. All other data used in the escape room is controlled by Raccoon Serious Games.

8 Process

In this chapter, the process of how the three team members worked from the start to the final product will be described. The process is evaluated in terms of the use of scrum and git, the planning, the distribution of work and communication. Finally, the process, in general, is evaluated.

8.1. SCRUM and Git

From the first moment on, it was decided to use SCRUM as an agile management framework. It is a well-known framework for project-based learning, which is widely used in computer science [2]. Working according to the SCRUM methodology means working in so-called sprints, which divide the work.

The team worked with weekly sprints, where each week had a certain goal, called a milestone, to which all main issues that week contributed. These main issues were then divided up into smaller issues, which were then divided amongst the team members for an equal distribution of the work. The choice for a weekly sprint was made because this would offer enough work for all three team members while also allowing for a good check on whether the progress was still in line to be completed in time for the final deadline.

At the end of each day, or at the beginning of the next, the team would try to discuss what they had previously worked on if there was anything extra important that had to be done that or the next day, and whether there were any issues encountered. In practice, this would not happen on a strict day-to-day basis, but as everyone worked together closely, the whole team was always updated on the status of the sprint.

On Mondays, the team would start each week by finishing up any reviewing left from the last week, then go over all the issues from the last week and see what was done and what was not. The complete sprint was resolved, mostly by finishing all issues, with a few exceptions for issues that were not, which were discussed and then moved to the next sprint. Afterwards, the new sprint would be looked at. As the team worked from a backlog and planning, the next sprint was already drawn out, the issues only had to be split up into smaller ones if necessary and divided amongst the team members.

The team worked from a Git repository, to ensure good code cooperation and version control. Each sprint worked from its development branch and after each sprint, the development branch would be merged into the master branch and a new development branch would be created. This ensured that the master always contained a working product.

As mentioned, a sprint backlog was used during development. The backlog was created in the first weeks when the Product Plan was drawn up, which can be found in Appendix I. The GitHub Project Board was used to keep track of issues throughout the project as well as during one week. It included a main backlog, a ToDo list, an In Progress section, an In Review section and the issues that were already done. The sprint issues were put in the ToDo list each Monday, together with additional issues that were created during development, with bugs or additional functionalities.

Finally, the GitHub repository was configured to require two approvals for merge requests and secure its master and development branches from direct pushing of commits. Within the team, it was agreed that each merge request should be reviewed by everyone. As one person works on a merge request, the other two must review. It was also decided that at least one of them would have to do a very thorough review, the other could be less thorough but would still have to check the work. This not only ensured proper code reviews, but it also guaranteed that all team members would know how every part of the system works, which was important, as there are only three team members.

The team chose for SCRUM because in the context of the BEP-project, with a set deadline and a midterm meeting halfway through, it was easy to manage that all requirements would be met within the time available. The whole team already had experience with using SCRUM, and also the coach and client were familiar with the methodology.

8.2. Planning

As pointed out in the previous section, a sprint backlog was used. In the first two weeks of the project, research was conducted on what the product would and a product plan was drawn up. This plan can be found in Appendix I, it includes a sprint backlog and roadmap with the planning of what features would be implemented in what time frame.

The whole project was be spread out over ten weeks, starting with two weeks of research, then four weeks of developing before presenting a minimal working product in the midterm meeting in week 6. Finally, four more weeks of coding were scheduled to finish the product, report and presentation. Writing a road plan at the start, gave a good structure to make sure everything would be done on time.

The roadmap was detailed for the first weeks to make sure a minimal working product was ready in week 6. After that, it was not very detailed, with only some ideas for extra features to implement. This was done on purpose, and these weeks would be filled in with issues in week 6.

The planning for the first weeks turned out to be very on point. The milestones set for each week were met and a well working minimal product was delivered. After the midterm meeting, the team sat together with the client to decide which extra features and issues would get priority during the last few weeks of the project, and issues were created accordingly. The new focus was mainly on system testing and adding more functionality to the user interface. Setting up the system in a production environment and making sure everything works well with the hardware, as well as writing tests for guaranteeing a working system was very important to finish the product. Besides that, adding functionality like camera screens, configuration file submissions and improving the front-end to be more user-friendly got a high priority to make sure the system would work well for the client. Finally, the roadmap was updated throughout the project when the planning changed.

8.3. Distribution of work

As there are only three members, they had to work together very closely. The work was distributed equally amongst all team members and throughout the first week, it was ensured that all team members would work on all different system components so that everyone would be able to understand everything.

However, during the course of the project, preferences arose about certain components, so each team member had one main component he/she worked on. This was not decided in general but turned out like that when dividing the issues. Besides, each team member would still occasionally have issues that, partly, required working on a different component.

The front-end was mainly designed and implemented by Issa. She started using the Material Design framework and also added the main features, like the menu. Gwennan did most of the work for the client computer Python library. She implemented several devices that were used in testing as well. Marijn did a lot of work on the back-end, setting up the configuration handling, he also wrote the JavaScript library. However, every team member has worked on each component themselves as well.

It was good that everyone worked on all parts and had a complete understanding of the entire system. However, it also worked very well to be the expert on something. If some front-end feature had to be improved, Issa would know straight away how to do that. But if there was an issue with the system running on the hardware, Gwennan would be the person to talk to. Just like Marijn would know exactly how the configuration was handled in the back-end. Being the expert on a component would make it easier to push an improvement through or know exactly in what part to look for a bug. So, the team was satisfied with how this worked out, also because this made it easy to divide issues each week, and there was hardly any discussion about people wanting to do the same work.

8.4. Communication

Within every team, there must be communication about the work and progress. Furthermore, as there are two supervisors to this project, there must also be communication between each of them and the team. The communication during the project is further analysed in this section.

8.4.1. Internal communication

Between the team members, the communication was mainly about what work had to be done, what was already done, how issues were coming along and if any problems were encountered. Furthermore, discussions took place about the best way to implement certain features.

Every day, the team worked in the office of Raccoon Serious Games, where they had their desks. This way, they were always in the same location when working, which allowed them to easily discuss with each other. When implementing a certain issue and unsure about the cleanest or most efficient way, it was very simple to ask the other team members about their opinions and continue accordingly. As a result of always working together, encountered bugs or problems could be solved easily, since other team members were close by to ask if they might know the answer since they were more familiar with that section of the code.

This straightforward communication allowed for the work to go smoothly and issues to be fixed quickly. Furthermore, when reviewing each other's work, it was easy to immediately ask someone if they could take a look or to ask for some explanation about commentary.

The work was mainly done during office hours, as all team members were present every day from nine till five. If anything came up outside of the office, the team would communicate over WhatsApp. Additionally, practical information about someone's presence or other appointments was discussed there.

While working, the team was able to communicate constructively with each other. Discussions arose on the topic of a certain implementation, but there was never any conflict between team members. This allowed for a very nice working relationship. Furthermore, the team was also bonded with each other and the staff members of Raccoon Serious Games during the daily lunches with the complete office, and team building activities. This created opportunities to also talk to each other about topics other than work, which made the work fun.

8.4.2. External Communication

With both the client, Jan-Willem, and the university coach, Taico, the team kept in touch about their progress. Most communication was handled through e-mail, and additionally, there was communication over WhatsApp with Jan-Willem and over Mattermost with Taico, respectively.

Client Every week, on Monday there was a meeting with Jan-Willem, where the progress of the last week was discussed along with the planning of the new sprint. Furthermore, this was a good chance to ask any possible questions about accessibility to materials or question about the design or implementation. As he is very busy and has a tight schedule to stick to, this was the best moment to ask. Otherwise, questions would be asked over WhatsApp or e-mail, so he could answer when he found the time.

The meetings were usually quite short, about half an hour to an hour. Unless the team had specific questions, the process would be discussed and everyone would get back to work. As there was also a demo every Friday for the full Raccoon Serious Games team, explicitly showing progress after the weekend usually would not have any added value.

Looking back, meeting more frequently would have been more beneficial to the course of the product. In the beginning, during the research phase, the client was often asked about the findings and a lot of feedback was given to various points of research. However, once development started, one meeting per week was enough to steer in the right direction, but having more structured meetings where the team would come up with extra talking and discussion points would have allowed them to update the product more quickly.

Coach Initially, it was decided to meet every week at the university on Thursday, which for the first few weeks did happen. However, it soon turned out that as the team worked together well, and did not have any issues, the meetings were not needed for extra guidance. The team decided that the time was preferably used to work on the product. Demos were not feasible as they would require the team to bring several devices and Raspberry Pis to the university and set these up before the meeting, so there was little to discuss.

During the midterm meeting, the team was able to give a small demo for the coach and as the progress was going according to the planning, there was no reason to meet more frequently.

The supervisor was updated weekly, whether it was face-to-face or over e-mail about the progress. For the team, this worked well, because short questions could also be asked over e-mail or Mattermost, which usually concerned practical information. Finally, feedback on specific issues was given over e-mail, especially for all the document, and that was very helpful.

8.5. Evaluation

The team is very satisfied with the process. The work was fun to do and the team worked together very nicely. As the product further developed, the team got more involved in the project of Raccoon Serious Games, as the product will be used in real life when it is delivered. This was very stimulating and made it easy to work towards a goal, where the team could see their product in use.

There were no conflicts within the team about each other's work attitude or presence, which made the working environment very pleasant. Everyone worked hard, so there was no case of team members piggy-backing on the work of others.

8.6. Special circumstances

Unfortunately, the father of Issa fell very ill halfway through the project, and he was hospitalized for six weeks. This caused Issa to be out for a full week, right before the midterm. Also after the Christmas holidays, she was not able to come back to work full-time, as she was still in the hospital almost every day.

The product was well on its way, development was going very well and on schedule, so the team could bear to miss a team member for some time without the fear of not making the project deadline. However, with less manpower, the team was not able to implement as many extra features as they might have liked to do, like creating a page on the front-end where the user could more easily write configuration files which would automatically put in all the keys correctly; or improving the configuration error handling to give more informative errors.

The team still kept in touch over WhatsApp, even if Issa would not be in the office, so there would still be communication about what had to be done. At the beginning of the week, Issa would try to estimate an expected workload she could carry that week. For example, it was easy for her to work on the report, as she could easily do that from the train on her way to the hospital.

Finally, due to her absence, the merge request demand of two reviewers was reduced to one. If possible, merge requests would still be reviewed by two team members, but if this turned out impossible, the other team members would still be able to continue working.

9 Conclusion

After analysing the system and development process, the whole project will be evaluated and compared to the problem posed in Chapter 2, whether or not the developed product, S.C.I.L.E.R., meets the goal of the project.

Raccoon Serious Games wanted a system to manage an escape room and automate as much as possible, so the operator would have to do as little as possible. Furthermore, the system should provide the operator with an easy-to-use user interface, optimised for tablet viewing, and allow the operator to have full control over the escape room. The configuration of the escape room should be generated from a JSON file, which should be checked for correctness to ensure no crashes will happen during run-time. Writing the configuration should be straight-forward to someone who has written JSON files before, so manuals should be easy to grasp. Finally, the system must run smoothly for longer periods, imagine days, without crashing or causing delays in the communication, even when it receives many massages at the same time. The devices in an escape room should be able to communicate with the system, by using a custom library.

In the last few weeks of the project, the team already started drawing up configurations for the escape rooms of Raccoon Serious Games. The configuration file for a complete escape room was written by a team member in about half a day. So, for the client, who is more familiar with the contents, but less with the format, the time to write configurations should be about equal or a little bit longer. Moreover, the more configuration files one writes, the easier it becomes to translate an idea into the configuration.

The configuration handling has been extensively tested, to ensure that no possible error in the file could cause the system to crash. Instead, informative errors are returned on every part of the configuration which does not abide by the structure, as described in the manual in Appendix D.

It was found that the system was able to handle all possibilities that the escape room requires, so it is flexible enough to handle the requests of the client. During testing in the last weeks, a bug was found in running the system from a Raspberry Pi, but this was also resolved. So, the system can run reliably from a Raspberry Pi, which was the client's preference. While running the system, the team did not notice any delays once the system was fully started, and the latency tests confirmed that the system's latency stayed below the threshold of 100ms, as set by the team.

Furthermore, the library for client computers that control devices was implemented in both Python and JavaScript to allow devices to choose which programming language to use. Both are published to their respective package installers, so they can be easily accessed. The client has already used the libraries to program devices to be used in escape rooms and was able to implement everything that was required.

Finally, during demos of the system, the client indicated any improvements they would like to see for the user interface to improve the usability. These were all implemented and reviewed by the client during weekly meetings to ensure that the front-end would meet the requirements of an operator. Furthermore, the user test gave improvements for the user interface, and also to optimise it for tablet use. These were noted by the team and either resolved or are mentioned as further improvements in Chapter 10.

In the end, S.C.I.L.E.R. meets all the requirements from the client and all challenges posed in Chapter 2, so it solves the problem that it set out to solve. Therefore, it can be concluded that this project was completed successfully.

10 Discussion and recommendation

The final chapter will discuss any recommendations that the team has for the future, on how the system can be improved and maintained. Furthermore, some possibilities for additions to the systems are suggested.

10.1. Future improvements

The design decisions made during the project led to the product as it is at the moment of writing. The back-end implementation allows for many different types of puzzles to be implemented and used by the system. Although other possibilities exist of structuring the configuration, the team is satisfied with the current implementation.

However, further improvements can be made in the field of testing. Although the system is tested on latency, and thereby message sending between components, more of these tests can be written. Furthermore, extra front-end UI tests can be written using Karma, Jasmine and Protractor to test button functionality on the front-end and automate the testing of proper button initialisation. With further studying the Jasmine and Protractor frameworks, developers can find out how the data can properly be inserted into the front-end through tests, such that functionality of the front-end can actually be tested. This was not done by the team, because they gave priority to other features over intensely studying the Protractor framework. Moreover, a test for the client computer library can be written where communication with the broker is mocked and an implementation of the abstract device class is used. Finally, the test for the device implementation should be updated with the newest library version and a test for the JavaScript library should be written.

As mentioned in Section 5.3.2, the communication and message handling functionalities are split in the back-end. However, the system can be improved by restructuring the back-end to have the `Communicator` and `MessageHandler` as one object, which would simplify the complete set-up and avoid circular dependencies.

Additionally, the client computer library currently consists of an application file and an abstract device class, which should be implemented. These could be put together to simplify the library. However, the current implementation does restrict access to the communication methods, which would be changed when putting the two classes together as one. So, depending on preference and desired functionality this could be changed.

Another implementation that can be improved is the labelling of device components. As currently, the labelled component's ids first need to be retrieved before instructions can be sent. This could be improved by already including the component ids with the labelling in the `WorkingConfig`, such that both can be retrieved at once and the `ComponentInstructions` can be sent right away.

The system can be further improved by adding the comparison type 'button'. Currently, if a device has a button that can be pressed, this is a component of that device. So, if a condition checks whether a button is pressed, it should only check when the button is actually pressed and not keep a $pressed = true$ status forever. Now, the implementation first sends a $pressed = true$ and immediately a $pressed = false$ status, such that the button's actions are fired, but its status is reset again. With this implementation, the 'not' comparison, which checks whether a certain value is **not** equal to the value used in the constraint, works. However, if the $false$ status is not immediately sent after pressing a button, the 'not' comparison will keep triggering with every status update. Therefore, in the future, by implementing a 'button' type, the conditions with this type can check the conditions of the rule it belongs to, and will only trigger once until the buttons send another status update.

Finally, more improvements can be made to the user interface. It should be optimised for tablet use and need little explanation to understand. The team already has several ideas for improvements, most of which will be implemented in the week after this report has been finished, but before the product is presented. These include the splitting of the table for puzzles and general events in the front-end into two tables; the updating of buttons in the front-end (like making the component expansion button more visible); adding the option to test a device's connection while playing; creating more visual information on whether devices are properly reset; and making the puzzles table more informative to match with the hints that can be given. Finally, the camera feed could be shown on the home page in a small box in the lower corner, which could then be expanded.

10.2. Maintainability

This section advises on measures to take in the future to ensure the maintainability of the system. It is divided into four parts: testing, code style, code review and code structure.

10.2.1. Testing

Currently, the back-end is fully tested to guarantee the functionality is solid and to be sure to catch any errors in the configuration or message contents. It is recommended to keep adding more tests when new functionality is written in the future, to prevent the unintended behaviour from entering the system.

10.2.2. Code style

Travis is configured to run the code style checker for each different language and fails if any errors or warnings occur. So, by continuing to use the GitHub repository for updating the code, Travis ensures that newly written code will still comply with the current code style rules.

10.2.3. Code review

The team has kept up the practice of always reviewing merge requests with two people, as far as possible. This was almost always possible unless Issa was not available due to personal circumstances, see Section 8.6. It is advised to keep this up for any future code changes to make sure that no bugs arise when developing new functionalities.

10.2.4. Code structure

The GitHub includes a check whether the new code still complies with the guidelines set by SIG. These guidelines concern the code structure and its maintainability, as large units of code are harder to grasp and therefore maintain, for example. If the guidelines are not met, the plugin will warn the developer, so the code can still be refactored and improved to keep the code base of high quality.

10.3. Future additions

Since the back-end and client computer library offer solid implementations which can, thus far, handle any request the client had for the system, there are a few additions that can be made to the system.

However, on the front-end, a few additional features can still be added, with two main extra functionalities being the automatic configuration generation and the visual overview of the escape room.

10.3.1. Generating configuration files

As configuration files can easily mount up to more than one thousand lines of code, see for example Appendix E, it would be a great addition to have a place to enter ideas, or parts of the escape room puzzles, where the system will then automatically create a configuration file, free of errors. This was a feature that the team thought of early on in development but was also immediately set as a great nice-to-have feature which would, however, require a lot of time to implement. Thus, it was set only as a feature that would be implemented if the team was left with a week of basically doing nothing.

This feature would require a user interface, whether the user is requested to fill in information which is then put in a JSON file. For data like the name, host and description of the escape room, this would not be a difficult task, however, for the condition tree logic, this is a much more complex job. Therefore, this issue was not completed over the course of this project and is a great addition for the future.

10.3.2. Visual overview

Because writing a configuration file in JSON does not offer a visual representation or nice overview of the escape room, this would be an additional feature to S.C.I.L.E.R.. The employees of Raccoon Serious Games are all very graphically orientated. This was an actual feature request by the client but was also said to be only feasible when the team was left with a lot of extra time.

This issue would require a lot of graphical experience, which the team does not have, and might also include the generating of a configuration from a visual interface. Furthermore, the interface could allow the user to drag events over a timeline to thereby alter the configuration of an escape room.

These are all great additions to the system and would very much improve the usability of the system, also allowing users without JSON experience to write configuration files. However, the team did not have enough time within the project for this, so this is left as a possible future addition.

Analysis Both additions would greatly benefit Raccoon Serious Games, as it allows people with less experience to also easily handle the system. Both additions are about visualisations on the front-end and can, therefore, be seen as features belonging together. For that purpose, this would be a good project to take up, and could be a new BEP.

However, with only three developers the team was able to do more than they had initially anticipated. Therefore, it is hard to estimate whether this would render enough work for a new BEP. Complexity-wise it should match, especially if the event visualisation includes the dragging of components to alter the escape room without having to rewrite the configuration file.

Bibliography

[1] Education technology. Using serious games technology to educate youth about cybercrime. Retrieved from *Education technology.* https://edtechnology.co.uk/Article/using-serious-games-technology-to-educate-youth-about-cybercrime/, 12 2019.

[2] Amy Fox. Scrum in the computer science classroom. Retrieved from *Computer Science Teachers Association.* http://advocate.csteachers.org/2019/02/17/scrum-in-the-computer-science-classroom/, 02 2019.

[3] Raccoon Serious Games. sciler. Retrieved from *npm.* https://www.npmjs.com/package/sciler/, January 2020.

[4] Raccoon Serious Games. sciler. Retrieved from *PyPi.* https://pypi.org/project/sciler/, January 2020.

[5] Lucy Hattersley. Raspberry Pi 4 vs Raspberry Pi 3B+. Retrieved from textitThe official Raspberry Pi magazine. https://magpi.raspberrypi.org/articles/raspberry-pi-4-vs-raspberry-pi-3b-plus, November 2019.

[6] Tilak Lodha. Concurrency and Parallelism in Golang. Retrieved from *Medium.* https://medium.com/@tilaklodha/concurrency-and-parallelism-in-golang-5333e9a4ba64, September 2017.

[7] Jan-Willem Manenschijn. Wat is een serious game? Retrieved from *Medium.* https://raccoon.games/wat-is-een-serious-game/, 10 2019.

A

UML's in full size

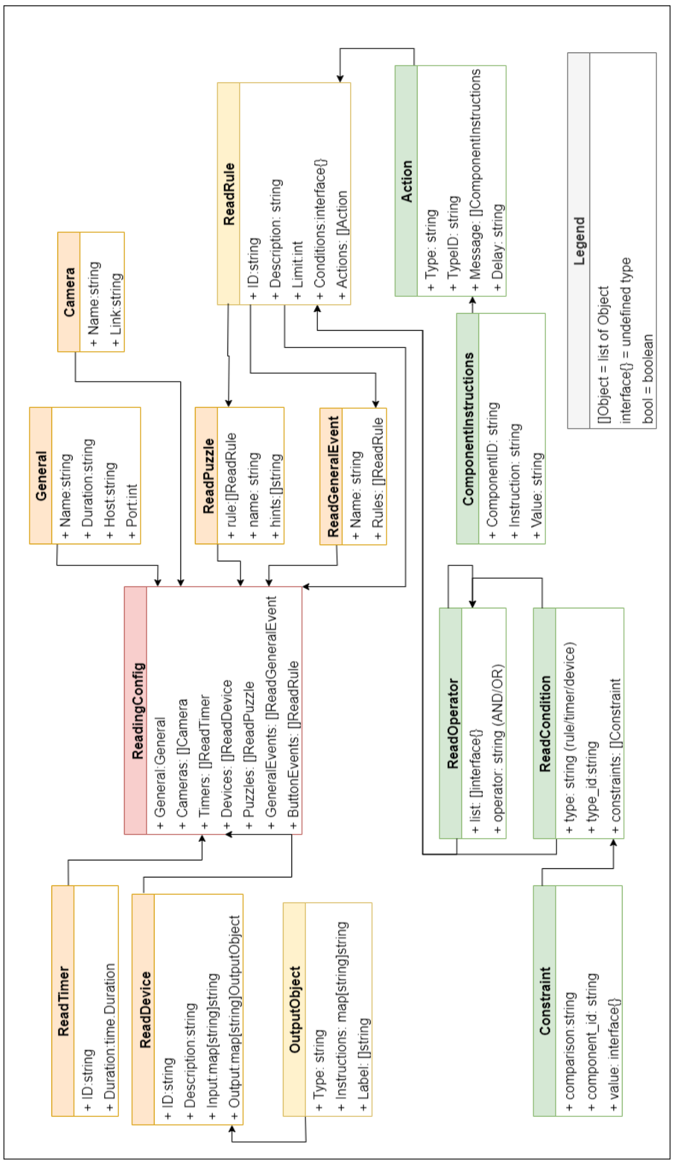


Figure A.1: UML of the `ReadConfig` object after the first type checking.

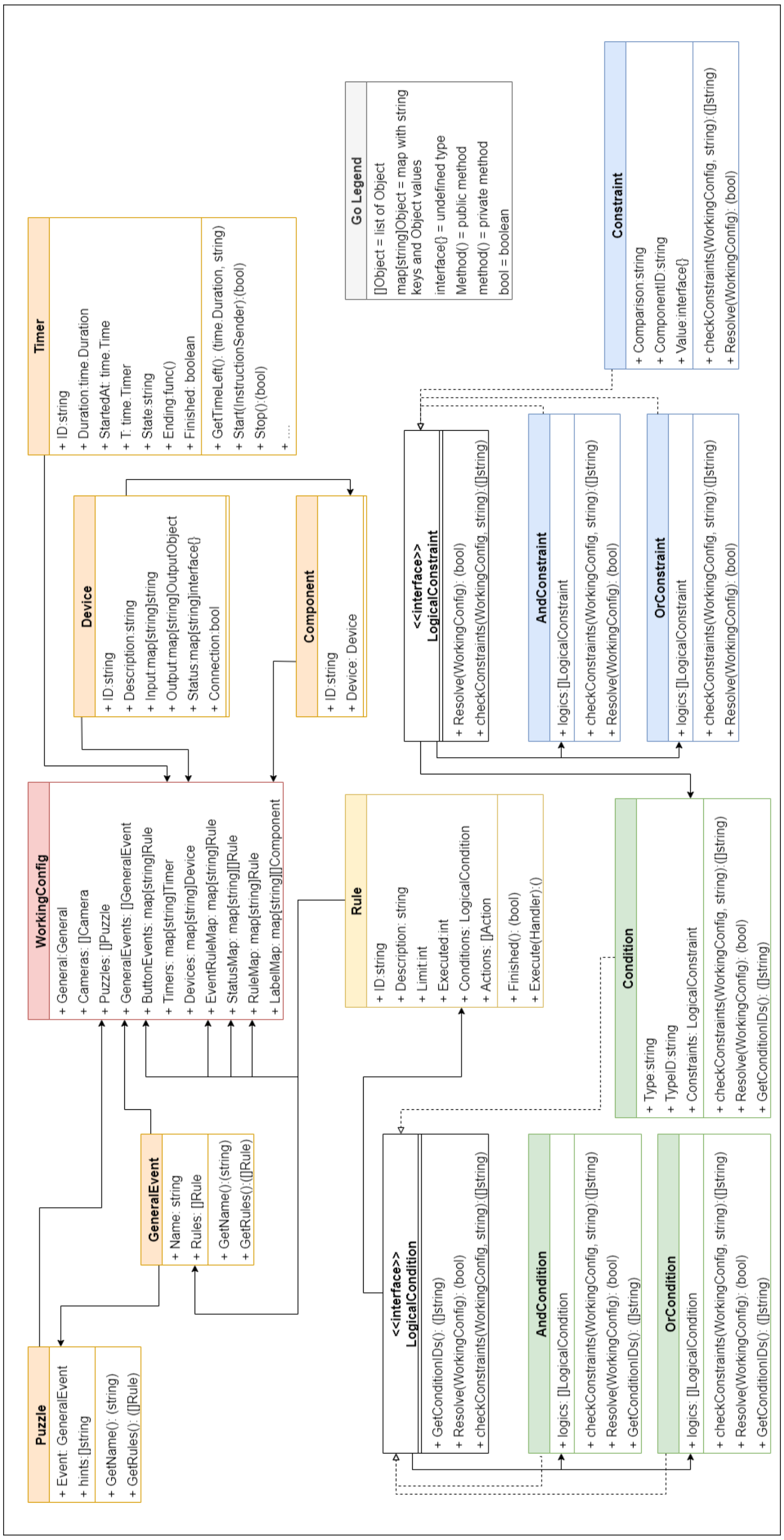

Timer
+ ID:string
+ Duration:time.Duration
+ StartedAt: time.Time
+ T: time.Timer
+ State:string
+ Ending:func()
+ Finished: boolean
+ GetTimeLeft(): (time.Duration, string)
+ Start(InstructionSender):(bool)
+ Stop():(bool)
+ ...
Go Legend
[]Object = list of Object
map[string]Object = map with string keys and Object values
interface{} = undefined type
Method() = public method
method() = private method
bool = boolean
Constraint
+ Comparison:string
+ ComponentID:string
+ Value:interface{}
+ checkConstraints(WorkingConfig, string):([]string)
+ Resolve(WorkingConfig): (bool)
Device
+ ID:string
+ Description:string
+ Input:map[string]string
+ Output:map[string]OutputObject
+ Status:map[string]interface{}
+ Connection:bool
Component
+ ID:string
+ Device: Device
<<interface>> LogicalConstraint
+ Resolve(WorkingConfig): (bool)
+ checkConstraints(WorkingConfig, string):([]string)
AndConstraint
+ logics:[]LogicalConstraint
+ checkConstraints(WorkingConfig, string):([]string)
+ Resolve(WorkingConfig): (bool)
OrConstraint
+ logics:[]LogicalConstraint
+ checkConstraints(WorkingConfig, string):([]string)
+ Resolve(WorkingConfig): (bool)
WorkingConfig
+ General:General
+ Cameras: []Camera
+ Puzzles: []Puzzle
+ GeneralEvents: []GeneralEvent
+ ButtonEvents: map[string]Rule
+ Timers: map[string]Timer
+ Devices: map[string]Device
+ EventRuleMap: map[string]Rule
+ StatusMap: map[string][]Rule
+ RuleMap: map[string]Rule
+ LabelMap: map[string][]Component
Rule
+ ID:string
+ Description: string
+ Limit:int
+ Executed:int
+ Conditions: LogicalCondition
+ Actions: []Action
+ Finished(): (bool)
+ Execute(Handler):()
Condition
+ Type:string
+ TypeID:string
+ Constraints: LogicalConstraint
+ checkConstraints(WorkingConfig, string):([]string)
+ Resolve(WorkingConfig): (bool)
+ GetConditionIDs(): ([]string)
GeneralEvent
+ Name: string
+ Rules: []Rule
+ GetName():(string)
+ GetRules():([]Rule)
Puzzle
+ Event: GeneralEvent
+ hints:[]string
+ GetName(): (string)
+ GetRules(): ([]Rule)
<<interface>> LogicalCondition
+ GetConditionIDs(): ([]string)
+ Resolve(WorkingConfig): (bool)
+ checkConstraints(WorkingConfig, string):([]string)
AndCondition
+ logics: []LogicalCondition
+ checkConstraints(WorkingConfig, string):([]string)
+ Resolve(WorkingConfig): (bool)
+ GetConditionIDs(): ([]string)
OrCondition
+ logics: []LogicalCondition
+ checkConstraints(WorkingConfig, string):([]string)
+ Resolve(WorkingConfig): (bool)
+ GetConditionIDs(): ([]string)

Figure A.2: UML of the processed configuration, showing the class hierarchy used to work with. Some Go stucts are missing, these are already in Figure A.1

B
User testing manual

You will be running a mini escape room. You have control over the escape room through buttons. Three puzzles need to be solved upon which actions should happen. You can find information about devices and puzzles below. Please make the escape room ready for use and start the game. The team will pretend to be players who do not understand what they are doing, so help them find out.

Devices

- hue lights: 4 spots in the escape room
- Control board: has three switches, three slides with lights and one main switch. default values: all sliders to 0 and switches down
- display: show messages
- keypad: numpad to enter codes
- door: magnetic door opener

Puzzles

- red: solved when the red switch is true and its slider is set 100
- orange: solved when orange switch is true and its slider is set between 50 and 80
- green: solved when green switch is true, its slider set to 100 and the red puzzle has been solved
- keyCode: solved when Raccoon's postal code is entered
- hint: after 5 seconds, a hint is send.

Manual triggers

- puzzle overview: add five minutes to time or subtract 1 minutes from run time
- button: start game
- button: pre-game phase
- button: pause game
- button: stop game
- button: continue game

C Message manual

`message`: this is the format of messages sent between all system components.

C.0.1. Base of message

- `device_id`: the id of the device of which de message is sent.
- `time_sent`: the time at which the message is sent in the format "dd-mm-yyyy hh:mm:ss".
- `type`: the type of the message
- `contents`: the actual contents the message want to pass on

`device_id` and `time_sent` is defined the same for each message, `type` and `contents` are specifically defined depending on the sender and receiver.

C.0.2. Back-end to Client Computers

- `type`: the type of the message, this can be:
  - `instruction`
- `contents`:
  - If type is `instruction`, then the message contents is **list** of instructions that have:
    - ◇ `instruction`: one of the instructions specified for this device or component or one of the following instructions: `test`, `status update`, `reset`
    - ◇ `value`: this is the value (argument) for the instruction (optional)
    - ◇ `instructed_by`: this is the id of the device which originally send this instruction (usually front-end)
    - ◇ `component_id`: this will be the id of a component in a timer or device (optional)

example

```
{
"device_id": "back-end",
"time_sent": "17-1-2019 16:20:21",
"type": "instruction",
"contents": [
        {
        "instruction":"turnOnOff",
        "value": true,
        "component_id": "led1"
        },
        {
        "instruction":"turnOnOff",
        "value": false,
        "component_id": "led2"
        }
```

```
    ]
}
```

C.0.3. Client Computers to Back-end

- `type`: the type of the message, this can be:
  - `status`
  - `confirmation`
  - `connection`
- `contents`:
  - If type is `status`, then the message contents is map of status's that have a `key` with a `value`. where `key` is a component_id, and `value` it's status in a format defined in the configurations of the device. e.g. `{redSwitch:  true, redSlider:  40, redLed:  "aan"}`
  - If type is `confirmation`, then the message contents has te following:
    - ◇ `completed` is a boolean.
    - ◇ `instructed` is the original instruction message for the device, including the `instructed_by` tag
  - If type is `connection`, then the message contents has te following:
    - ◇ `connection` is a boolean defining the connection status of the device.

Example

```
{
"device_id": "controlBoard",
"time_sent": "17-1-2019 16:20:20",
"type": "status",
"contents": {
    "redSwitch": true
    "blueSwitch": false
    }
}
```

C.0.4. Front-end to Back-end

- `type`: the type of the message, this can be:
  - `instruction`
- `contents`:
  - If type is `instruction`, then the message contents have
    - ◇ `instruction`: one of following instructions: `send setup`, `send status`, `reset all`, `test all`, `test device`, `finish rule`, `hint`

Example

```
{
"device_id": "front-end",
"time_sent": "17-1-2019 16:20:20",
"type": "instruction",
"contents": [{
        "instruction":"finish rule",
        "rule": "playlist puzzel 1",
        }]
}
```

C.0.5. Back-end to Front-end

- `type`: the type of the message, this can be:
  - `confirmation`
  - `status`
  - `time`

  - `instruction`
- `contents`:
  - If type is `confirmation`, then the then the message contents have
    - ◇ `completed`
    - ◇ `instructed`, which contains the original instruction with in the `contents`:
      - · `instruction`
      - · `instructed_by`
  - If type is `instruction`, then the then the message contents have - `instruction` with value `reset` or `status update` or `test` or `setState`
  - If type is `status`, then the message contents has
    - ◇ `id` of device
    - ◇ `status` has a map of `component_id` keys and `status` values
    - ◇ `connection` boolean
  - If type is `event status`, then the message contents has objects with
    - ◇ `id` of rule
    - ◇ `description` of rule
    - ◇ `status` of rule describes whether the rule is finished or not
  - If type is `front-end status`, then the message contents has a list of objects with:
    - ◇ `id` of button
    - ◇ `disabled` status of button
  - If type is `time`, then the message contents have
    - ◇ `id` of timer
    - ◇ `duration` has a number of the duration left in milliseconds
    - ◇ `state` sting of the timer state
  - If type is `setup`, the contents:
    - ◇ a `name` parameter carrying the name of the escape room
    - ◇ a `hints` parameter carrying a map with the name of the puzzle as key and a list of hints as value
    - ◇ an `events` parameter carrying a map with the name of the rule as key and the description as value
    - ◇ a `cameras` parameter carrying a list with camera objects with a `name` and `link` tag
    - ◇ a `buttons` parameter carrying a list of button names that should be added to the front-end

example

```
{
"device_id": "back-end",
"time_sent": "17-1-2019 16:20:20",
"type": "status",
"contents": [
        {
        "id": "controlBoard"
        "connection":true
        "status": {
            "redSwitch": true
            "blueSwitch": false
            }
        }
    ]
}
{
"device_id":"back-end",
"time_sent":"17-01-2020 15:33:28",
"type":"event status",
"contents":[
    {"id":"rood","status":false},
    {"id":"oranje","status":false},
    {"id":"add 5 min timer","status":false},
    {"id":"time up","status":false},
    {"id":"Puzzles done","status":false},
    {"id":"stop timer","status":false},
    {"id":"groen","status":false}
    ]
}
```

D
Configuration manual

D.1. Configuration file

This manual will help you write a configuration file for an escape room. The file should be written in JSON and all the tags necessary are explained here. An example can be seen in `example.config.json`. The same format

There are three main components to the file:

- `general`
- `cameras`
- `devices`
- `timers`
- `puzzles`
- `general_events`
- `button_events`
- `rules` which are defined for puzzles

D.1.1. General

This is the general information of the escape room. It includes the following tags:

- `name`: this is the name of the escape room, this is a string, e.g. "Escape room X". This can be displayed in the front-end, so should be readable and in Dutch.
- `duration`: this is the duration of the escape room, which should be a string in the format "hh:mm:ss".
- `host`: this is the IP address of the broker through which clients and back-end connect, formatted as a string.
- `port`: this is the port on which the broker runs, formatted as an integer.

D.1.2. Cameras

This will be a list of camera objects, that are set up in the room. An object has two properties:

- `name`: the name of the camera
- `link`: the link to the camera's IP address

D.1.3. Devices

This will be a list of all devices in the room. Each device is defined as a JSON object with the following properties:

- `id`: this is the id of a device. Write it in camelCase, e.g. "controlBoard". This id should be unique compared to other device ids and also the rule ids as well as the timer ids.
- `description`: this is optional and can contain more information about the device. This can be displayed in the front-end, so should be readable and in Dutch.
- `input`: defines the type of values to be expected as input. The keys are component ids and values are types of input (in string format). Possible types are: "string", "boolean", "numeric", "array", or a custom name.

- `output`: defines the outputs that can be expected, with their status and what instructions can be performed on these components
  - `type`: defines the type of values to be expected as output. Possible types are: "string", "boolean", "numeric", "array", or a custom name.
  - `instructions`: this is a map of the name of an instruction to the type of argument the instruction takes
  - `label`: this is a list of possible labels this component listens to when an action gets called on a label.

D.1.4. Timers

This will be a list of all the time-related actions/conditions. all timers have to be started in an action and be checked in a condition to be used. - `id`: this will be the id of the timer. Write in camelCase and numbers are fine, e.g. "timerHint1". This id should be unique compared to other timers ids and also the rule ids as well as the device ids. - `duration`: This will be the duration after which the timer will trigger to true and the conditions containing the timer will be checked to execute actions. The format is XhXmXs, each size optional, e.g. 1h30m30s, 40m30s, 1m30s

D.1.5. Puzzles

This will be a list of puzzle objects, which have the following properties:

- `name`: name of puzzle. This can be displayed in the front-end, so should be readable and in Dutch.
- `rules`: array of rule objects (see below)
- `hints`: array of hints (strings), specific to each puzzle. These can be displayed in the front-end, so should be readable and in Dutch.

D.1.6. General Events

General events have the following properties:

- `name`: name of event, for example, "start"
- `rules`: array of rule objects (see below)

D.1.7. Button Events

Button events are events that happen on the click of a button in the front-end. The event in the config is a rule, as defined below. The conditions can be empty or include conditions that the rule depends on, like stop can only be pressed when start is already pressed. When defined, the button is disabled until these conditions resolve.

The buttons to manage the game state are configured in this section. The front-end has output component `gameState` which keeps track of state of the game. The button event conditions can depend on the `gameState` and the actions should alter the `gameState`.

The `gameState` can have several states, which can be defined through the config. Example states are: `gereed`, `in spel`, `gepauzeerd` and `gestopt`, but more states can be used. Its default start-up status is `gereed`. The instruction for changing the `gameState` is `set state`.

An example for `start` is given below.

D.1.8. Rules

Rules are defined by:

- `id`: this is the id of a rule. Write it in camelCase, e.g. "solvingControlBoard". This id should be unique compared to other rule ids and also the device ids as well as the timer ids.

- `description`: this is optional and can contain more information about the rule. This can be displayed in the front-end, so should be readable and in Dutch.

- `limit`: this sets the number of times this rule can be triggered. If this is `0`, it means unlimited.

- `conditions`: this is either a logical operator (i) defined by `operator` (either `AND` or `OR`) and `list` which is a list of conditions or other logical operators **or** this is a condition (ii) defined by `type`, `type_id` and `constraint`

1. Logical operator
    - `operator`: this can be `AND` or `OR`
    - `list`: this is an array of conditions / logical operators
2. Condition
    - `type`: this can be `rule`, `timer` or `device`.
    - `type_id`: this will be the id of a timer, rule or device, depending on the type.
    - `constraints`: this is either a logical operator (i) defined by `operator` (either `AND` or `OR`) and `list` which is a list of conditions or other logical operators **or** this is a constraint (ii) defined by `comp`, `value` and `component_id`
        1. Logical operator
            - ◇ `operator`: this can `AND` or `OR`
            - ◇ `list`: this is an array of constraints / logical operators
        2. Constraint
            - ◇ `comparison`: this is the type of comparison and can be `eq`, `lt`, `gt`, `contains` , `lte`, `gte`, `not`. However, only `eq` will work on all types, `lt`, `gt`, `lte`, `gte` only on numeric, and `contains` only on arrays, `not` does not work on booleans.
            - ◇ `value`: this is the value on which the comparison is made. In case of `device` type, it should be in the same type as specified in the input of the device. In the case of `timer` type, it should be boolean In the case of `rule` type, it should be numeric since the comparison will be done against the times the rule is executed
            - ◇ `component_id`: in the case of "device" type, this is the id of the component it triggers. In the case of "timer" type, this is non-existent.

- `actions`: this is an array of actions:
    - `type`: this can be `device`, `timer` or `label`
    - `type_id`: the id of device, timer or label, depending on type respectively
    - `message` in case of type `device`: this defines a list of componentInstructions which have a:
        - ◇ `component_id`: this will be the id of a component in a timer or device
        - ◇ `instruction`: one of the instructions specified for this device and component
        - ◇ `value`: this is the value for the instruction of the type specified for this device and component
    - `message` in case of type `timer`:
        - ◇ `instruction`: one of the instructions for timer, e.g. `start`, `stop`, `pause`, `done`, `add`, `subtract`
        - ◇ `value`: optional, in case of `add` and `subtract` a time should be given in format XhXmXs
    - `message` in case of type `label`:
        - ◇ `instruction`: one of the instructions specified for the components with this label
        - ◇ `value`: this is the value for the instruction of the type specified for this device and component
    - `delay` in case of type `device` or `label`: This is optional, this is a duration in format XhXmXs, if an action has a delay, the message will publish after this delay.

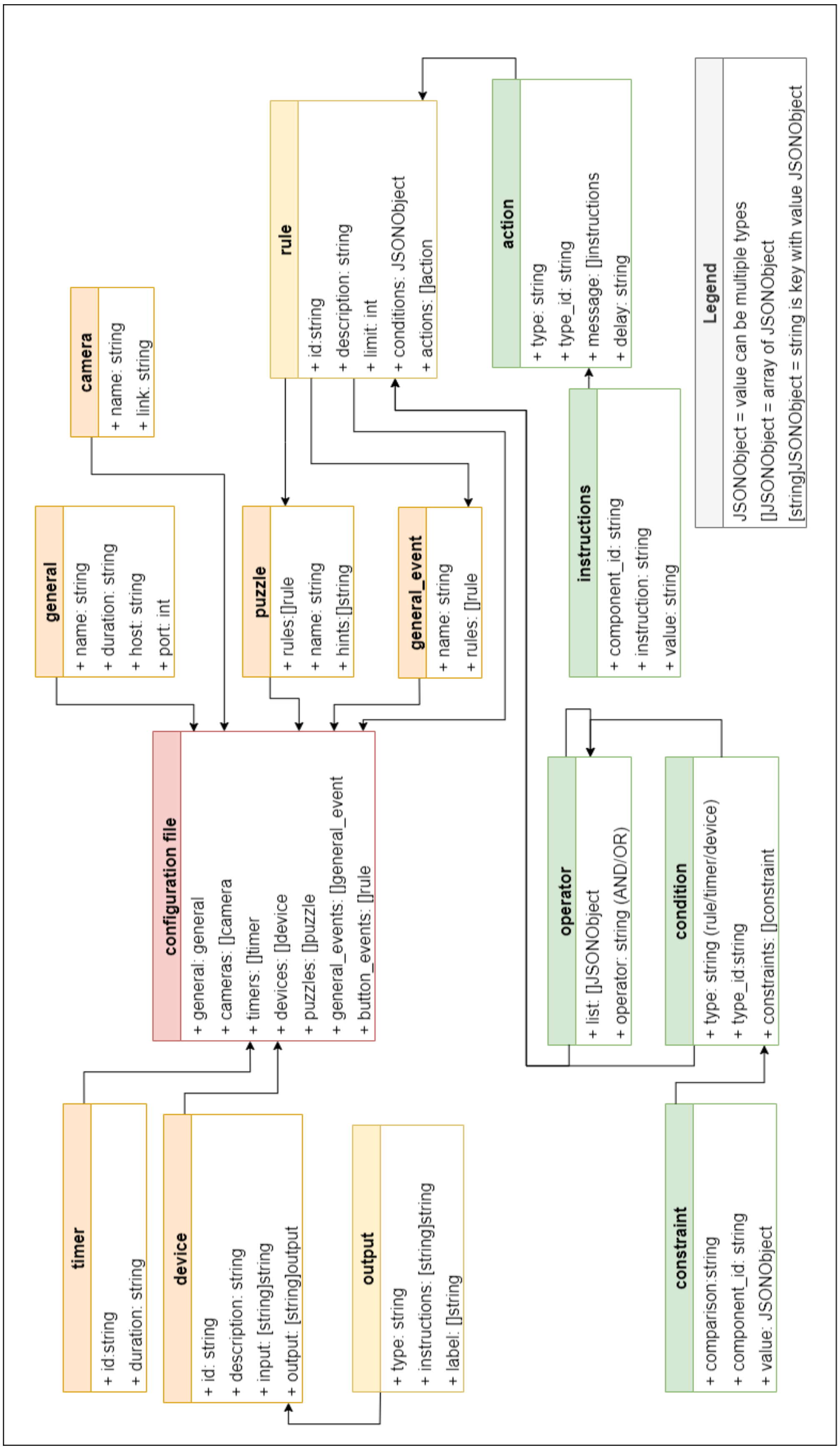


Figure D.1: The structure of the configuration json file

E
Example configuration file

```json
{
  "general": {
    "name": "Example escape room",
    "duration": "30m",
    "host": "localhost",
    "port": 1883
  },
  "cameras": [
    {
      "name": "camera1",
      "link": "localhost:2200"
    }
  ],
  "devices": [
    {
      "id": "tvScreen",
      "description": "The TV screen inside the escape roomm to display
         the time and any hints.",
      "input": {},
      "output": {
        "display": {
          "instructions": {
            "hint": "string"
          },
          "type": "string"
        }
      }
    },
    {
      "id": "door",
      "description": "The door is controlled by a door magnet via a
         relay, this client computer controls the relay to open/close
         the door.",
      "input": {},
      "output": {
        "door": {
          "instructions": {
            "open": "boolean"
          },
          "type": "string"
        }
      }
    },
    {
      "id": "keypad",
      "description": "Numpad with Enter bar to send codes, when enter is pressed.",
      "input": {
        "code": "numeric"
      },
      "output": {}
    },
    {
      "id": "spotlight",
      "description": "Spotlight that is pointed in the direction of the closet where the secret code can be found.",
      "input": {},
      "output": {
        "spot": {
          "instructions": {
            "on": "boolean"
          },
          "type": "string"
        }
      }
    }
  ],
  "timers": [
    {
      "id": "timer1",
      "duration": "10m"
    }
  ],
  "puzzles": [
    {
      "name": "ascii puzzle",
      "hints": [
        "Have you found the secret code?",
        "Have you ever heard of ASCII?"
      ],
      "rules": [
        {
          "id": "asciiCode",
          "description": "the code 157 should be entered, which is the ascii translation of the answer of the secret code found in the closet.",
          "limit": 1,
          "conditions": {
            "type": "device",
            "type_id": "keypad",
            "constraints": {
              "comparison": "eq",
              "component_id": "code",
              "value": 157
            }
          },
          "actions": [
            {
              "type": "device",
              "type_id": "door".
              "message": [
                {
                  "instruction": "open",
                  "value": true
                }
              ]
            }
          ]
        },
        {
          "id": "asciiCodeWrong",
          "description": "a different code than 157 is entered, so
             point the spotlight to the closet where the secret code
             can be found.",
          "limit": 0,
          "conditions": {
            "type": "device",
            "type_id": "keypad",
            "constraints": {
              "comparison": "not",
              "component_id": "code",
              "value": 157
            }
          },
          "actions": [
            {
              "type": "device",
              "type_id": "spotlight",
              "message": [
                {
                  "instruction": "on",
                  "value": true
                }
              ]
            }
          ]
        }
      ]
    }
  ],
  "general_events": [
    {
      "name": "helping with finding the closet",
      "rules": [
        {
          "id": "pointToCloset",
          "description": "10 minutes after the start, the players are
             helped by pointing the spotlight to the closet",
          "limit": 1,
          "conditions": {
            "type": "timer",
            "type_id": "timer1",
```

```json
            "constraints": {
              "comparison": "eq",
              "value": true
            }
          },
          "actions": [
            {
              "type": "device",
              "type_id": "spotlight",
              "message": [
                {
                  "instruction": "on",
                  "value": true
                }
              ]
            }
          ]
        }
      ]
    },
    {
      "name": "timer actions",
      "rules": [
        {
          "id": "add 5 min timer",
          "description": "add 5 min to general timer",
          "limit": 1,
          "conditions": {},
          "actions": [
            {
              "type": "timer",
              "type_id": "general",
              "message": [
                {
                  "instruction": "add",
                  "value": "5m"
                }
              ]
            }
          ]
        }
      ]
    }
    {
      "name": "Stop time's up",
      "rules": [
        {
          "id": "timeUp",
          "description": "duration has counted down to 0, players have not made it out.",
          "limit": 1,
          "conditions": {
            "type": "timer",
            "type_id": "general",
            "constraints": {
              "comparison": "eq",
              "value": true
            }
          },
          "actions": [
            {
              "type": "device",
              "type_id": "tvScreen",
              "message": [
                {
                  "component_id": "display",
                  "instruction": "hint",
                  "value": "Time is up!"
                }
              ]
            }
          ]
        }
      ]
    },
    {
      "name": "Stop when done",
      "rules": [
        {
          "id": "puzzlesSolved",
          "description": "all puzzles are solved, so stop the time.",
          "limit": 1,
          "conditions": {
            "operator": "AND",
            "list": [
              {
                "type": "rule",
                "type_id": "asciiCode",
                "constraints": {
                  "comparison": "eq",
                  "value": 1
                }
              }
            ]
          },
          "actions": [
            {
              "type": "device",
              "type_id": "tvScreen",
              "message": [
                {
                  "component_id": "display",
                  "instruction": "hint",
                  "value": "You've solved all puzzles!"
                }
              ]
            },
            {
              "type": "timer",
              "type_id": "general",
              "message": [
                {
                  "instruction": "pause"
                }
              ]
            }
          ]
        }
      ]
    }
  ],
  "button_events": [
    {
      "id": "start",
      "description": "start the game with the button.",
      "limit": 1,
      "conditions": {},
      "actions": [
        {
          "type": "timer",
          "type_id": "timer1",
          "message": [
            {
              "instruction": "start"
            }
          ]
        },
        {
          "type": "timer",
          "type_id": "general",
          "message": [
            {
              "instruction": "start"
            }
          ]
        },
        {
          "type": "device",
          "type_id": "front-end",
          "message": [
            {
              "component_id": "gameState",
              "instruction": "set state",
              "value": "in game"
            }
          ]
        }
      ]
    },
    {
      "id": "stop",
      "description": "Als het spel stopt, moeten alle lichten uitgaan",
      "limit": 1,
      "conditions": {
        "operator": "AND",
        "list": [
          {
            "type": "device",
            "type_id": "front-end",
            "constraints": {
              "component_id": "gameState",
              "comparison": "eq",
              "value": "in game"
            }
          }
        ]
      },
      "actions": [
        {
          "type": "timer",
          "type_id": "general",
          "message": [
            {
              "instruction": "stop"
            }
          ]
        },
        {
          "type": "device",
          "type_id": "front-end",
          "message": [
            {
              "component_id": "gameState",
              "instruction": "set state",
              "value": "in game"
            }
          ]
        }
      ]
    }
  ]
}
```

F SIG feedback

The team was required by the university project description to submit their codebase to the Software Improvement Group, in week 6 and 9 of the project, respectively week 4 and 7 of development. After the first submission, the team received feedback and tried to improve their codebase based on that.

F.1. First review

The codebase was submitted on Friday, December 19, 2019. On Friday, December 27, 2019, the team received feedback from SIG, which can be found here, however, it is in Dutch.

F.1.1. SIG Feedback

Let tijdens het bekijken van de feedback op het volgende:

- Het is belangrijk om de feedback in de context van de huidige onderhoudbaarheid te zien. Als een project al relatief hoog scoort zijn de genoemde punten niet 'fout', maar aankopingspunten om een nog hogere score te behalen. In veel gevallen zullen dit marginale verbeteringen zijn, grote verbeteringen zijn immers moeilijk te behalen als de code al goed onderhoudbaar is.
- Voor de herkenning van testcode maken we gebruik van geautomatiseerde detectie. Dit werkt voor de gangbare technologieën en frameworks, maar het zou kunnen dat we jullie testcode hebben gemist. Laat het in dat geval vooral weten, maar we vragen hier ook om begrip dat het voor ons niet te doen is om voor elk groepje handmatig te kijken of er nog ergens testcode zit.
- Hetzelfde geldt voor libraries: als er voldaan wordt aan gangbare conventies worden die automatisch niet meegenomen tijdens de analyse, maar ook hier is het mogelijk dat we iets gemist hebben.

[**Feedback**] De code van het systeem scoort 3.9 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code marktgemiddeld onderhoudbaar is. We zien Unit Size en Unit Complexity vanwege de lagere deelscores als mogelijke verbeterpunten.

Bij Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Dit kan verschillende redenen hebben, maar de meest voorkomende is dat een methode te veel functionaliteit bevat. Vaak was de methode oorspronkelijk kleiner, maar is deze in de loop van tijd steeds verder uitgebreid. De aanwezigheid van commentaar die stukken code van elkaar scheiden is meestal een indicator dat de methode meerdere verantwoordelijkheden bevat. Het opsplitsen van dit soort methodes zorgt er voor dat elke methode een duidelijke en specifieke functionele scope heeft. Daarnaast wordt de functionaliteit op deze manier vanzelf gedocumenteerd via methodenamen.

Voorbeelden in jullie project:

- Handler.onInstructionMsg (Go)
- generateLogicalConstraint in configHandler.go (Go)
- compareType in handler.go (Go)

- AppComponent.processMessage (TypeScript)

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Dit betekent overigens niet noodzakelijkerwijs dat de functionaliteit zelf complex is: vaak ontstaat dit soort complexiteit per ongeluk omdat de methode te veel verantwoordelijkheden bevat, of doordat de implementatie van de logica onnodig complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is, en daardoor eenvoudiger te onderhouden wordt. Door elk van de functionaliteiten onder te brengen in een aparte methode met een beschrijvende naam kan elk van de onderdelen apart getest worden, en wordt de overall flow van de methode makkelijker te begrijpen. Bij grote en complexe methodes kan dit gedaan worden door het probleem dat in de methode wordtd opgelost in deelproblemen te splitsen, en elk deelprobleem in een eigen methode onder te brengen. De oorspronkelijke methode kan vervolgens deze nieuwe methodes aanroepen, en de uitkomsten combineren tot het uiteindelijke resultaat.

Voorbeelden in jullie project:

- Constraint.checkConstraints (Go)
- checkActionDevice in configHandler.go (Go)
- AppComponent.processMessage (TypeScript)

De aanwezigheid van testcode is in ieder geval veelbelovend. De hoeveelheid testcode ziet er ook goed uit, hopelijk lukt het om naast toevoegen van nieuwe productiecode ook nieuwe tests te blijven schrijven.

Over het algemeen is er dus nog wat verbetering mogelijk, hopelijk lukt het om dit tijdens de rest van de ontwikkelfase te realiseren.

F.2. Actions taken afterwards

The complexity and size of a few units were found to be too large. This mostly included handling functions on both back- and front-end.

For the back-end, the handler was split into the main handler file and a helper file. The main handler still maps the messages based on their type and calls the appropriate methods. Each type of message calls the appropriate helper methods `on{Type}Message` to handle that type of message. The messages received need to be checked for properly structured contents, so a lot of the checking functionalities have been separated from the main handling method to simplify.

The handler test method has been split up into multiple test files, to keep a clear overview of what to test can be found where. There is now a separate `confirmation_test.go`, `instruction_test.go`, `status_test.go` and `handler_test.go`, of which the latter contains the connection tests, some general tests, and edge-case tests.

The configuration handler in the back-end also performs a lot of type checking which requires many if- and switch-statements, which add a lot of complexity to a unit. The file was split up into the `configHandler` and the `checkConfig`, to prevent one use configuration handler file, which was not structured anymore.

The big methods were split up, dividing functionality and checking of smaller methods, using as much the same method as possible. However, for checking the types of device input values, four different methods were needed for the four different types, as each needs to be checked against its type and has own restrictions on the type of comparisons allowed.

On the front-end, the problems mostly occurred in the message handling and processing to set the proper attributes and update information. Because circular dependencies had to be avoided, most of the actual updating is done in the app component, which makes that file pretty big, with well over 400 lines of code, including documentation. Most methods are split up in helper methods to keep the code readable.

With these changes and updates to the codebase, the team strives to improve its SIG score and ensure a cleaner codebase.

F.3. Second review

On Friday, January 24, 2020, the team submitted their codebase to SIG for the second time. However, the feedback was not received before the deadline of this report, so could not be included.

However, the GitHub repository includes the SIG score of the main branch, which shows a score of 10/10, see Figure 6.3 in Chapter 6.

G Project description

This is the original project description from Raccoon Serious Games. The description still contains their old company name, Popup-escape. The project is called: Live & instant communication between escape room sensory and effects.

G.1. The project

The most popular escape rooms are those who are technically advanced. A challenge for popup-escape is that all their escape rooms are mobile. Furthermore, its escape rooms change continuously.

Popup-escape has created its own software to manage escape rooms and configurations. This software is outdated and has limitations, it is hard to create new, automated puzzles and add them to a game.

Popup-escape would like to create a new software system which enables the game makers to control and connect their existing puzzles and create new puzzles.

G.2. Popup-escape offers

- A lot of fun: You are going to be tested on your puzzle skills during the project
- A informal (student-like startup) work environment and
- Working with technologies you like and think are the best
- The possibility to work on hardware such as Raspberry Pi's, arduinos, old radios and phones
- The possibility to work on games which are going to be played by real users. You can see your software in action!
- Learning from previous BEP-projects experiences and popup-escape's employees. We will be challenging you.

We have two projects lined up for which we want to use your solution! Possible research questions

What technology can best be used to manage live/direct connections over the air? What architecture fits best with the modular system of puzzles?

G.3. About Popup-escape

Popup-escape is a young company/startup started by a Delft Computer Science student four years ago. It started out as a hobby and it is now grown to a company with 6 full-time employees. We have designed and made about 70+ escape rooms over the past four years and at least 10.000 people have played our escape rooms.

Popup-escape is being re-branded to raccoon serious games. Raccoon Serious Games creates escape rooms and other games to communicate a message, to let people learn and connect.

H
Infosheet

Product Plan

Project Plan v.3.1

Bachelor End Project 2019-2020 Q2
Live communication system for escape rooms
Delft University of Technology

Issa Hanou Gwennan Smitskamp Marijn de Schipper

Supervisors:
Jan-Willem Manenschijn (Raccoon Serious Games)
Taico Aerts (Delft University of Technology)

January 6, 2020

Contents

1 Introduction

Escape rooms are a popular activity to do with a group of friends, colleagues or family. An escape room is an isolated location, consisting of one or more physical rooms, in which a game is played by a team of players who want to escape from the room or reach another type goal by finding and solving puzzles.

With the rise of technology, escape rooms have become more advanced and recently include a lot of technology to automate the running of the escape room and minimise the input of the operator while the escape room is in play. This way an unbelievable experience can be created for the user.

Raccoon Serious Games specializes in designing custom escape rooms for its clients and wants to automate these escape rooms while using state of the art technology. They developed the previous system themselves.

Raccoon Serious Games wants to have a new software system to control and operate escape rooms. This system should connect existing puzzles and let the operator manage the escape room while it is in play.

2 Product description

2.1 Customer needs

The customer needs a system to manage escape rooms in real time and configure its hardware manually before running the escape room. The current software is outdated and has limitations. The biggest problem is reliability as it would sometimes disconnect or crash and the most important customer need is that the system should not crash during the escape room.

The customer wants to implement the created system in multiple escape rooms, thus the system needs to be have a predefined format for the configuration of the hardware and events. However, this configuration should also be flexible to allow for more devices to be added easily. The configuration should be logical and easy to write for Raccoon Serious Games.

The customer plans to use the software in escape rooms at the end of February, so it needs to be ready before then. The official project ends at the end of January, from the university's perspective. So, the customer wants to work closely with the team members to have a working product for this event.

2.2 Critical attributes

The final product will contain these three parts:

- A front-end (website), connected to the back-end, which will be used to control the escape room by an operator.
- Client computers, like Raspberry Pis, which will control automated puzzles. These puzzles are handled by different devices and the client computers must be able to control devices like lights, music, special effects, doors, sensors, etc.

- A back-end, which manages all connected devices, for example Raspberry Pis, and tell them what to do as well as giving feedback to the front-end.

These components need to be able to communicate directly and fast. When connection breaks, it should reconnect within seconds. However, the system should be reliable to prevent connection breaks to the upmost extent.

3 Research for design decisions

In the first two weeks, the team members will be conducting research on a number of topics to prepare themselves for developing the software that matches the desires of the customer. To do this research, several research questions have been formulated, which can be found below. These will be answered in the research report.

3.1 In what way is the old system of the customer not suitable for future use?

First of all, the customer owns an existing system, which the team members can explore to find out which components work well already and which ones need to be improved.

3.2 What are the most important features to have in an escape room system?

Besides what currently exists, it is also important to find out what the client would like to see in the application the team develops; the features they want included and how they want to use the product.

The customer already owns a system for large so-called escape events, which are not limited to a location and include a few hundred participants all playing simultaneously. This system might contain components that can help the team members with their product. The team wants to develop a product which works similarly on the user side (front-end), such that the customer has similar products.

Furthermore, it is important to find out what the current state of the art is for escape room systems. What features are included in current systems, which are not and why not. There might also be other existing systems, using similar techniques to what the team can use for their system.

3.3 What is the best set-up to use to communicate between different components of an escape room system?

As many factors play a role, this research question has been divided into the following sub-questions:

- What is the best technique to use for communication between the back-end and client computers?
- What language and framework is most suitable for maintaining communication with different client computers, while managing the escape room?
- What language can be used to code the library for the client computers to ensure proper instruction handling and easy to control the connected device?
- What language can generate a user-friendly interface on the mobile device of the operator, while communicating with the back-end?

The product will have three main components: a front-end, a back-end and the client computers, such as Raspberry Pis. The back-end runs the application and is connected to the other components. Therefore, it is important that the team knows how to communicate with the hardware of the client computers.

Additionally, the team needs to know how to implement all planned features. For example, instructing devices to perform an action synchronously or managing a room full of connected devices at to same time. It is important to know how to communicate this fast enough and how these messages will be validated. Other features to research include: backup, failure prevention, game configuration. The security of the system is not deemed relevant as a research topic by the client, as the system will run locally.

All the components have their own language(s), which might overlap, but it is important to decide on the most suitable language for each system component. Together with deciding on languages, it is important to do some research on which frameworks might be useful and can be used with a certain language. Although, it may turn out that no framework is preferred for the implementation.

3.4 What type of hardware is small enough to hide in the room, while it is also suitable as a reliable back-end, communicating with multiple client computers?

Additionally, research is needed to find out what type of hardware is best suited for hosting the back-end. It should be a device, small enough so that can be hidden within an object inside the escape room, but powerful and reliable enough to run the escape room. As the escape rooms for which the systems will be used are in a different location each time, it is important that the back-end device can be easily set up and connected to all client computers within the room.

3.5 What configuration format or template contains all necessary information, can be written by a member of the Raccoon Serious Games team and can be interpreted by the system?

As the configurations are interpreted by the back-end, it is important to come up with a good technique to facilitate this and a proper format to enter the configurations in. For this project, it is most important that it is easy to draw up the configurations without having to write out five extra lines per setting. At the same time, the target customer is common with programming, so there is no need for a GUI.

3.6 What is the latency threshold that the system can allow to avoid hindrance to the players?

As there will always be a delay in a communication system, it is important to define what the threshold for the allowed delay, or latency, is. In an escape room system, the client does not want users to experience any of this delay. When putting the correct block in its place, the door should open pretty instantly without the player having to wonder whether it really was the correct block as nothing happened.

3.7 What are the best tools to use for keeping a consistent code base?

Finally, as the team members want our code base to be of a high quality and be able to guarantee that, they will decide on tools to use to ensure code quality, including testing and code style.

4 Product backlog

As the team wants to find out more about what the customer wants in the product, for now this is a draft, and a final product backlog will be drawn up at the end of the two weeks of research. The backlog is formulated as MoSCoW requirements per component of the system.

4.1 Glossary

- A user is an escape room operator.
- A room is a full escape room which might consist of several physical rooms.
- A client computer, like a Raspberry Pi, controls one or more devices.
- A device is a physical object in the room, like a lamp.

- The status of a device is the value which it currently holds, for a lamp this can be on/off, for a scale this can be a certain weight.
- An event specifies what happens in the escape room, by defining triggers, conditions and actions.

  * The trigger of an event is another event, a device or the time of the escape room.

  * The condition of an event is the checking of a certain value from a device or result of an event.

  * The action of an event is the changing of a device's status or the trigger of a next event.
- Input to the back-end is a message of a client computer or the front-end.
- The label of a device specifies it belonging to a certain group, like 'hint' to specify all devices that can give a hint.
- The pre-game phase is the mood set in the room when players enter, but the game has not started yet. This includes lighting and music.
- The post-game phase is the mood set in the room when players have finished the game. This includes lighting and music. Two different ones can be set, one when players have escape and one when time runs out.
- The reset phase is the default lighting of the room, in which the room can be reset.

4.2 Front-end

Must Haves

- The front-end must display the remaining time of the escape room.
- The front-end must have an overview of devices in the room with their current state.
- The front-end must have an overview of devices in the room with their current state of connection (through their client computer).
- The front-end must have a start button, initiating start event in the back-end.
- The front-end must have a general test button for the escape room to see that all devices are functioning and connected.
- The front-end must have a reset button, resetting all the devices through the back-end.
- The front-end must have a stop button, initiating the stop protocol on the back-end.

- The front-end must have the ability to type and send a hint.
- The front-end must be available to the user on a mobile device.

Should Haves

- The front-end should have pre-defined hints for the operator to give, which can be retrieved from the configurations.
- The front-end should have an overview of events that have been scheduled or finished or are still to do.
- The front-end should have a test button per device.
- The front-end should give a warning of devices that are not reset.
- The front-end should have a pause button, which allows for later continuity of the game, but temporarily makes to solving puzzles impossible.
- The front-end should display the feed of the camera in the room.

Could Haves

- The front-end could give an overview of the room's events that have been executed or not, with the time it took players to solve them.
- The front-end's overview of events could be sortable.
- The front-end could contain buttons for the operator to trigger several actions during the game. These can be predefined and a selected set of items.
- The front-end could retrieve and show logs from the current session, keeping track of time spend on events, entered device values and hints given. These logs are saved in a text file.
- The front-end could have a lost button, initiating the event that the group does not finish the escape room; regardless of the remaining time.
- The front-end could give a warning of devices that are not connected; either constant or repeated at the start of a run.
- The system could allow for players to ask for hints.
- The system could allow for multiple game modes, distinguishing modes where users can ask for predefined hints or the operator decides on hints to be sent.
- The system could have the option for the operator to ask the players if they want to receive a hint, before giving a hint.

- The front-end could allow the operator to choose options for how players are alerted of a hint, for example, only speaking out the hint or having a type writing sound.
- The front-end could have a test button, testing the sequence of the devices.
- The front-end could allow the operator to switch between the feeds of different cameras.
- The in-game screen could show a scoreboard with the current team's actual time and the times of previous teams.
- The front-end could show the in-game screen to the operator.
- The front-end could be optimisable for the operator, with custom success messages, countdown animations, etc.
- The front-end could have a visual interface to see how events relate to each other.
- The front-end could have a visual interface to enter the configurations.

Won't Haves

- The front-end will not include a sound mixer where the operator can define what audio comes from which speaker.
- The front-end will not be supported to run in Internet Explorer.

4.3 Back-end

Must Haves

- The back-end must be able to communicate directly to client computers and front-end client over WiFi and LAN.
- The back-end must be able to take in a configuration file in predefined format.

  * The configuration file must contain the settings of an escape room (duration, start event, etc).

  * The configuration file must specify the devices needed for the room.

  * The configuration file must specify the events that take place in the room.

  * Each event in the configuration file must include triggers, conditions and actions to define the order of events.

  * The configuration file must specify the reset, pre- and post-game phases.

- The back-end must keep track of the time of the escape room, starting from the start of play. This is used to get the remaining time.
- The back-end must keep track of the status of all devices that are required by the configuration of the escape room.
- The back-end must send instructions to client computers based on input from client computers and information from the configuration file.
- The back-end must send the instruction messages to client computers with a latency threshold of 100 ms.
- The back-end must keep track of the events, whether they have been executed and how many times.
- The back-end must be able to mark an event as executed based on input.
- The back-end must be configurable to schedule events based on in-game time, as well as based on a pre-condition from user input.
- The back-end must keep track of events that happen and messages send and received, writing these to a log text file to allow for the debugging of the system.

Should Haves

- The back-end should allow for the reset to be done manually (on the front-end) or to be automated with the start or stop button implementation.
- The back-end should be able to send messages specific to devices with a certain label, grouping devices together.
- The configuration should specify the the labels of devices, allowing for the grouping of devices.

Could Haves

- The back-end could have a connection to the booking system.
- The back-end could store the date, time and number of hints of a team who played the escape room, providing logs for the front-end.
- The back-end could allow for Bluetooth connections to be made with client computers.

Won't Haves

- An instance of the system will not be able to run in multiple locations at the same time.
- The back-end will not store the video recordings for a limited amount of time (one run or 24 hours), so the team can see their performance and cooperation afterwards.

4.4 Client computers

Must Haves

- A client computer must be able to receive an instruction message from the back-end.
- A client computer must be able to determine whether to act upon a received message and which device to trigger.
- A client computer must be able to send a confirmation message to the back-end.
- A client computer must be able to send a message with the status of its connected devices to the back-end.
- A client computer must be able to understand instructions to alter state of its connected devices.
- A client computer must be able to handle different instruction messages which it receives from the back-end in parallel, such that all instructions are executed properly and nothing is left out.
- A client computer must be able to handle different instructions messages from the back-end, while still being able to perceive the status of its connected devices.
- A client computer must be able to send status messages when the status of a connected device changes.
- A client computer must keep track of the messages send and received, writing these to a log text file to allow for the debugging of the system.

Should Haves

- A client computer should be able to be labeled, to allow for the grouping of devices.
- A client computer should be able to send status messages in intervals, which can be set in the configuration.

Could Haves

- The client computer could allow for Bluetooth connections to be made to the back-end.

Won't Haves

- The client computer will not be told how to tell a device to alter its state.

5 Roadmap

This roadmap contains the deadlines that the team wants to make each week. The team will develop using SCRUM, using weekly sprints. This way, the team can ensure to have agile development throughout the course of this project. By the end of each sprint, the following should be achieved.

5.1 Roadmap VERSION 1

Week 1 - Start November 11

- Draw up Project Plan.
- Start research.
- Communicate with Raccoon Serious Games about what features they want in the product.

Week 2 - Start November 18

- Draw conclusions about the research.
- Set up environments for the code base.
- Test the back-end environment to be compatible with the hardware, by running a test set-up.
- **Deadline Wednesday 20:** Discuss research conclusions with client.
- **Deadline Thursday 21:** Present findings of research to Raccoon Serious Games' team.
- **Deadline Friday 22:** Research Report.

Week 3 - Start November 25

The prototype should:

- have a minimal communication setup: the front-end should have a device test button which communicates to the back-end to send a message to the client computer to test the device;
- be able to read from configurations and interpret the information necessary for the the minimal communication.

The user should:

- be able to write a configuration file according to a manual;
- see a test button in the front-end;
- see the test functions of the connected devices after clicking (e.g. lamp flickers).

Week 4 - Start December 2

The prototype should:

- allow the client computers to send feedback to the front-end (via back-end) on whether actions have been performed;
- have client computer and back-end writing logs to text files with their messages send and received;
- have the front-end keeping an overview of the status of devices.

The user should:

- be able to see logs from client computers;
- see an overview of devices in the room in the front-end.

Week 5 - Start December 9

The prototype should:

- have the back-end which can perform an event by acting on input and sending a message to client computers based on the event.

The user should:

- be able to activate trigger and see its response (e.g. place the correct object in the right place and see the lamp turn on).

Week 6 - Start December 16

There should be a minimal working product, where the prototype should:

- have a complete front-end where the operator can see time, device connection status and can send hints to the designated client computers;
- have a complete front-end functionality with start, test, reset and stop button.

The user should:

- be able to send a hint from the front-end and see it appear on a connected device;
- be able to start, stop, test and reset the room.

* **Deadline December 19:** There is a midterm meeting with the client, coach and coordinator.

* **Deadline December 20:** SIG upload.

Week 7 - Start January 6

The prototype should:

- have the front-end keeping track of an overview of event;
- have the front-end show a set of pre-defined hints;
- have the front-end should give a test button per device;
- allow for client computers to be grouped by labels;
- allow for client computers to send status messages on intervals.

The user should:

- see an overview of puzzles in the front-end;
- be able to send a pre-defined hint into the room;
- be able to test each device on its own;
- be able to configure devices as groups by giving labels;
- see the feed of the camera in the front-end;
- be able to configure devices to send status messages on intervals.

Week 8 - Start January 13

The prototype should:

- have a pause button in the front-end;
- have a camera feed in the front-end;
- allow for the reset of the room to be done either manually, with the stop or with the start button.

The user should:

- be able to pause and then resume the game;
- be able to configure whether to reset manually, through start or through stop.

Week 9 - Start January 20

The prototype should:

- look very nice in styling on the front-end.

The user should:

- easily navigate through the front-end.

* **Deadline January 24:** SIG upload.

Week 10 - Start January 27

The prototype should:

- be final.

* **Deadline TBD:** Final report.
* **Deadline TBD:** Final presentation.

5.2 Roadmap VERSION 2.1

Week 7 - Start January 6

The prototype should:

- have the front-end keeping track of an overview of events;
- allow for client computers to be grouped by labels;
- have a camera feed in the front-end;
- have the front-end show a set of pre-defined hints;
- have the front-end should give a test button per device;
- have the front-end allow for entering a config file to check and use for the system;
- be reliably guaranteed through non-functional testing (performance testing, load testing, stress testing, volume testing, compatibility testing);
- use the client computer library as a properly imported library for Python using pip;
- have a client computer library for JavaScript.

The user should:

- see an overview of puzzles in the front-end;
- be able to configure devices as groups by giving labels;
- see the feed of the camera in the front-end;
- be able to send a pre-defined hint into the room;
- be able to test each device on its own;
- be able to check and use a configuration file;

- be able to run the system with as many device(input)s as the user wants;
- be able to use the client computer library for devices that run JavaScript;

Week 8 - Start January 13

The prototype should:

- be able to generate operator buttons in front-end from config file, such that it allows the end/reset/start of the room to be done in stages with corresponding reset buttons in the front-end;
- be configurable with delays for actions;
- allow for time to be added to or subtracted from timers as an action;
- have an easier to write configuration set-up and include proper error handling on faulty configuration files.

The user should:

- be able to pause and then resume the game;
- be able to reset properly;
- be able to configure the escape room with proper delays.
- be able to easily write configuration files.

Week 9 - Start January 20

The prototype should:

- look very nice in styling on the front-end;
- be easy to use in production;
- have a well-documented and well-structured code base.

The user should:

- be able to easily navigate through the front-end;
- be able to easily understand the code base and create configuration documents;
- be able to easily program client computers compatible with the system;
- be able to easily install the system on devices.

* **Deadline January 24:** SIG upload.

Week 10 - Start January 27

The prototype should:

- be final.

* **Deadline TBD:** Final report.

* **Deadline TBD:** Final presentation.

6 Collaboration plan

All three team members have agreed on the following:

- Using Scrum with weekly sprints: Monday to Monday, start with meeting to set up the new issues for the week and review previous sprint.
- Milestones per sprint, create issues per milestone.
- Always work on site from 9 AM till 5 PM, communicate clearly if not there, at least a day ahead, preferably more.
- Issues and merge requests should be complete with a description, time estimate, labels, assignee and due date. While working on issues, they should be updated on the boards, with time spend and comments when running into problems. When closing an issue, update the time spend and add a comment if issue was not fully completed.
- Communicate clearly when something is not going according to plan. If an issue turns out to be bigger or smaller then expected, or not possible at all, let everybody know as soon as possible to minimize chances of later complications within either the project or group.
- Merge requests should be evaluated by all other team members. At least one person who is not the main developer of the branch, should look closely at the code.
- The team will do weekly meetings with customer.
- The team will aim for weekly meetings with the coach.

J

Research report

Research Report v.3

Bachelor End Project 2019-2020 Q2
Live communication system for escape rooms
Delft University of Technology

Issa Hanou   Gwennan Smitskamp   Marijn de Schipper

Supervisors:
Jan-Willem Manenschijn (Raccoon Serious Games)
Taico Aerts (Delft University of Technology)

January 23, 2020

Contents

Introduction

As described in the Project Plan, the product to be developed will be used in small physical escape rooms. An escape room is an isolated location, consisting of one or more physical rooms, in which a game is played by a team of players who want to escape from the room or reach another type of goal by finding and solving puzzles.

This product is commissioned by Raccoon Serious Games, who develop both physical escape rooms as well as escape events. They want a new system to be developed to run as the communication system for the smaller escape rooms. This new system will be called Sensory Communication Inside a Live Escape Room, or S.C.I.L.E.R..

To develop this system, some research has to be done to figure out the best course of action. The research questions were formulated in the Project Plan and are discussed here. The main objective was to find the limitations of the old system, referred to as the Popup system, used by Raccoon Serious Games. Furthermore, the best platforms, languages and techniques to use for the communication within S.C.I.L.E.R. have to be found. Finally, this research is focused on deciding what features are most important to put into S.C.I.L.E.R..

1 In what way is the old system of the customer not suitable for future use?

The system already owned by Raccoon Serious Games, which also allows for the configuration and operation of physical escape rooms, will be referred to as the Popup system. It was created to facilitate the management of small physical escape rooms. This system is studied to see its limitations which will be analysed here.

1.1 Front-end

First of all, the Popup system allows for multiple escape rooms to be configured. The fields to create a new room are not very specific and do not enforce any restrictions or types on how to enter the information. When the wrong input is given and the user hits save, the action will be discarded and no fail massage is shown.

The configurations can be entered by creating events in the interface. Events must be created before they can be set as the start event of a room. So, you first create a room, then an event and then set the event as the start event of the room. This does not seem like a very efficient process.

The configurations for each escape room are entered through the interface, so they must be stored. However, the system will run locally each time. Therefore, the customer wonders whether it is really necessary to save the configurations within the system. If they can be read in before starting up the escape room, without a lot of processing time, there is no problem in doing this each time when setting up the escape room.

When running an escape room, the admin can fill in the team name, reservation name and e-mail. This way the run of the escape room can be easily connected to the booking system.

In the Popup system, a Dutch word can sometimes be found on the user interface, even though it is an English website. This seems sloppy and should be prevented in the S.C.I.L.E.R. system, by keeping one consistent language.

Finally, the Popup system is created in a way such that only the creator, and an operator guided by the creator, can use the interface. The properties of events and actions are not clearly listed after creation, input fields are not specified, and there is no clear operation flow.

1.2 Back-end

Lights, doors and sounds are implemented individually. They each have their own app within the Django project. On the front-end, the operator can also set each individually in a specific menu. For light objects, the fading and flickering of a light in both coloured and white light is hard-coded into the code, using a lot of randoms. This does not allow for future lights, which might be added with more abilities, to be used with these special capabilities.

The devices can be pinged to check their status. The connection to the devices is done using JSON, sending authentication codes. The system keeps track of the devices present in the room, as well as a list of connected devices. A room device is linked to a connected device, and when this link in the database exists, the device is seen as connected.

The code has minimal documentation, there is just enough written down to start the system as a developer, but in the whole project, it is hard to find any files with proper documentation, although a few do exist. Therefore, it is very difficult to do any maintenance or improve the Popup system.

The Popup system runs on a Raspberry Pi, however, the customer indicated that the system would crash sometimes. He could not find out why this happened and whether the problem was in the back-end, communication system or hardware.

1.3 Client computer library

Client computers are the machines (e.g. Raspberry Pis) that manage a certain device in the room (e.g. a lamp) and its communication to other components by a library. In the Popup system are handlers defined for specific devices that can be connected. There is also a general abstract file for the handlers, which would be good to have in the new library, so each device can get a handler implementing from this abstract handler. This provides a way of allowing new devices to be added easily, as they just have to implement an abstract handler.

The devices are set to connect to both the room and the full system. They will also keep trying to connect until they are successful. If a ConnectionError is given, the device should reconnect within one second. When the connection failed and an OSError is given, the device should also reconnect within one second.

Each device has a system uri, where the device can connect to the back-end, which is formatted as `ws://host/device/`.

The current client computer library has a handle function, which uses Python asyncio. It will ping the WebSocket on the back-end server and while that is done, it will also get the actual message ready to send and then send it, allowing for a concurrent flow.

Conclusion A new system will be developed as the Popup system does not have enough documentation to further build on it. Furthermore, the Popup system was not reliable as it would crash when running for long periods of time. Therefore, S.C.I.L.E.R. will be developed as the new system for Raccoon Serious Games.

For the front-end, it is very important to restrict user input to what is allowed and to help the user by giving instructions on what to enter in a specific field. The user has to receive feedback on its actions, like success and fail messages. The set-up process has to have a clear flow for the user, so the user will not feel like going back and forth to make a single adjustment in the escape room. The front-end will be written in Dutch, as the operators will be Dutch and they want to read normal terms (e.g. "apparaat" and "puzzel") and not the English technical words (like "client computer" and "event"). These take-aways have been included in the front-end design, which can be found in Appendix C

The back-end should be more flexible than the current Popup system is, it should regard lights, doors and music as devices instead of special events. Furthermore, the connection of devices will not be implemented similarly as the Popup system for S.C.I.L.E.R.. Instead of having both connected devices and room devices, which are linked when the connection for a device in the room exists, it would be preferable to have an object for each device, which keeps track of the connection. The concurrent flow as practised by the Popup system is definitely something to keep in S.C.I.L.E.R., though it should not be specific for WebSockets.

For the client computer library, an abstract handler is a good practice to implement specific handlers in the future.

2 What are the most important features to have in an escape room system?

To gather more insights into features that should be in the S.C.I.L.E.R. system, other systems were studied and analysed to determine which of their features should also be incorporated in S.C.I.L.E.R.. At the end of this chapter, a conclusion is drawn regarding all systems studied.

2.1 Existing systems on the market

Many systems exist which facilitate escape room management and control. Two systems were selected and studied in detail: QUEEN[1] and Houdini[2]. These two were selected after looking at available systems, eliminating those that did not provide systems for configuration and selecting those will extensive documentation. Examples of escape room system that do not provide all the functionality that S.C.I.L.E.R. will are EscapeAssist[3] and Escape Room Adming[4].

2.1.1 Houdini

Houdini is meant for controlling an escape room. It allows for configurations to be set and includes a back-end system that connects to Raspberry Pis and Arduinos in the escape room, using Web requests. A lot of documentation on the system can be found online, both about its features as well as the techniques it uses.

Houdini is a very extensive system, giving an operator many abilities to manage the escape room before and during play. As it includes many features, only the most interesting and seemingly unique features are mentioned here.

- Houdini is able to integrate with High-Voltage devices, such as a smoke machine.
- Houdini allows the operator to trigger devices in the room to perform an action during the game.
- Houdini has the option to show a scoreboard both to the operator as well as on the in-game screen, where the players can see the top player or average together with their own time.
- Houdini includes integration with mobile devices, so players can, for example, receive hints on the mobile phone.
- Houdini has an integrated CCTV viewer, allowing the operator to see the live footage of all cameras in the room, although sound is not included. Two feeds can be shown simultaneously and the operator can switch between the feeds of all cameras. It also allows the operator to take screenshots and recognise QR codes.
- Houdini allows for both timed scheduled events to take place as well as events that are triggered, though scheduled to trigger.
- Houdini offers integration with HUE lighting.

[1]https://escaperoomdoctor.com/queen
[2]https://houdinimc.com/
[3]https://escapeassist.com/
[4]https://www.escaperoomadmin.com/

- Houdini allows the operator to alter the time of the room in play, by adding or subtracting minutes.
- Houdini allows the operator to see the in-game screen on their operator screen, which is updated every two seconds.
- Houdini offers options to the operator to customise their screen: background, timer display, success message, countdown animation, etc.
- Houdini offers a sound mixer to the operator, where he/she can define what audio is played by different speakers.
- Houdini offers multiple game modes: where the number of clues can be predefined or not and the players can ask for clues (perhaps after x minutes) or the operator decides when the players receive a clue.

The Houdini system has extensive documentation online about what techniques they use to facilitate the communication between components of the escape room. Its main technique is using HTTP Web requests to communicate between the master-PC and the Raspberry Pis.

It sends specific requests at predefined URLs. All the Raspberry Pis are configured at specific IP addresses and the master PC should always be configured at the same IP too. Then, when the master PC schedules an event to take place, it will send a request to the Raspberry Pi that the event is directed at. For example, if the Pi is configured at IP 192.xxx.xxx.xxx and the event is called dimLights, then a Web request is sent to 192.xxx.xxx.xxx/dimLights and the Pi configured at this IP will dim the lights.

Furthermore, each of the Pis has a Python library configured to handle the sending and receiving of the HTTP requests.

Analysis After studying this system closely, some interesting features could be very nice for the S.C.I.L.E.R. system as well. Therefore, the following requirements have been drawn up as a result:

Should Have

- The configuration of devices should be flexible to allow multiple types of devices, including High-Voltage ones like a smoke machine.
- The front-end should contain buttons for the operator to trigger several actions during the game. This can be a predefined, select set of items.
- The system should allow for multiple game modes, distinguishing modes where users can ask for predefined hints or the operator decides on hints to be sent.
- The back-end should be configurable to schedule events based on in-game time, not only based on a pre-condition from user input.

Could Have

- The front-end could allow the operator to switch between the feeds of different cameras.
- The in-game screen could show a scoreboard with the current team's actual time and the times of previous teams.
- The front-end could show the in-game screen to the operator.
- The front-end could be optimisable for the operator, with custom success messages, countdown animations, etc.

Won't Have

- The front-end will not include a sound mixer where the operator can define what audio comes from which speaker.

Houdini does not offer the possibility of the operator hearing the recording of a microphone in the room. This is a feature that the customer would want to be able to use, therefore S.C.I.L.E.R. will be developed instead of using Houdini.

2.1.2 QUEEN

This system offers a vast amount of possibilities, including easy configuration and an interface to do so. This was a striking feature as offering easy-to-do escape room configuration, without requiring the user to write many lines of code, allows more people to work with the system.

QUEEN offers a lot of possibilities in their interface and also provides its users with a hardware kit to which their software can connect. It then asks the user to sketch the escape room set-up in their program and indicate where all the sensors and devices are located. They provide a wiki in which all different type of devices are labelled, for example, L.X for all lights, where X is a number.

QUEEN also provides the user with an interface to test the set-up of the escape room and offers a panel to the operator to manage the escape room in-play. This is a very extensive panel, as it allows for manual control of the devices in the room, while the game is played. Furthermore, the clue system offers both possibilities for automated clue sending, for example, upon a certain input, as well as sending custom clues by the operator. An interesting feature was script switching, allowing the escape room to be run in multiple languages or targeted at different age groups, without changing the configuration of the escape room.

Finally, a nice feature was the use of a typewriter sound when displaying a clue on the in-game screen. This notifies the user of a hint coming in, while also keeping the mystery vibe of the escape room alive with a hint slowly popping up 'out of nowhere'.

Analysis From studying their configuration system, a few features stood out as interesting and should also be thought of for the development of S.C.I.L.E.R.. This included the different type of devices that can be hooked up and how those can be used. For example, the tutorial on QUEEN's website illustrated an escape room where the players should turn a wheel to dim the lights and only with the proper illumination does the door open. The configuration system needs to be able to handle this.

Other examples include a light ray that is on for 10 seconds, then off for 20 seconds and only when a puzzle is solved when the ray is off does the next door open. Furthermore, the different types of sensors that can be hooked up are also important to consider. For example, the detection of players screaming.

These type of puzzles are an example of what the customer would want to do with the S.C.I.L.E.R. system, so the configuration should be able to handle this. This is not explicitly translated to a new requirement, as the current requirements for the configuration should cover this flexibility.

The following new requirements were drawn as possibilities from this study:

Should Have

- The front-end should allow the operator to choose options for how players are alerted of a hint, for example, only speaking out the hint or having a typewriting sound.

The QUEEN system only works with their custom hardware set. So, this does not allow the customer to use their own hardware. Therefore, S.C.I.L.E.R. will be developed instead of using the QUEEN system.

2.2 Existing system for large-scale escape events from Raccoon Serious Games

Raccoon Serious Games owns software, called the M.O.R.S.E. system, for managing large-scale escape rooms with a modular system. These escape rooms are called escape events, where large groups of people are split up into smaller groups, and they have to solve puzzles to win. Although this system has a different target than the S.C.I.L.E.R. system, which is focused on smaller physical escape rooms, they both have similar aspects, and therefore, M.O.R.S.E. is an interesting source to look for important features.

For the M.O.R.S.E. system, it is crucial that users can identify their team, do puzzles and see their own progress. The system serves as the tool where the team can enter their answers to puzzles. They might need extra physical objects to complete and solve a puzzle, but these tools are not sensed by nor do they communicate with the system. This is the biggest difference with the S.C.I.L.E.R. system.

However, in the S.C.I.L.E.R. system, it will still be important that players do have a possibility to track themselves. This will depend on the specific escape room, but there must be a place where they can see the time, receive hints and possibly see their progress. This should be in real-time, just like players see their progress in the M.O.R.S.E. system.

Where the M.O.R.S.E. admin side contains tables of groups and puzzles, the S.C.I.L.E.R. system's operator front-end will contain tables of events and devices. As the M.O.R.S.E. system noted, it is important that when these tables contain buttons, the tables are still styled in a way to provide an easy overview for the operator.

The M.O.R.S.E. system put a lot of thought into their hint process. An important take-away is having an overview of hints, that were given at a previous point in time, available to the players on an in-game screen.

Finally, M.O.R.S.E. uses a configuration for rule sets which define the actions to progress in the game as a team. This is very interesting, as it could be similar for the new system. This is analysed more in Section 5.1.

Analysis In terms of the system as is, the non-functional requirements are very similar to the new system. Both systems need to be very reliable, usable, maintainable and extendable. Therefore, these non-functional requirements overlap:

- **Maintainability**

  This concerns the restoring of a failed system to its normal operable state. Important aspects here are: how fast can defects or their causes be corrected and to what extent can unexpected errors be prevented.

- **Modularity**

  This concerns the disconnecting of system elements into interchangeable modules, such that different elements are not dependent on each other.

- **Deployment**

  This concerns the ability to quickly deploy the system after an update has been done. This requires the system's software to be easily available.

- **Performance**

  Escape rooms are all about user experience, so the system needs to work well. If the performance is not top-notch, then the user experience will be affected and this is definitely not in the interest of the customer.

- **Robustness**

  This concerns the ability of the system to deal with errors while the system is running, and how it reacts to erroneous input. If someone enters a different type of answer, the system should not all of a sudden break.

- **Security**

  This concerns the protection against unauthorised access to the system, allowing players to cheat the system, which should be avoided.

2.3 Features suggested by Raccoon Serious Games' team

As the product will be designed for Raccoon Serious Games and they want to use it for their actual escape rooms in February, it is important that the developers have a good understanding of what the customer wants to see in their product. Therefore, the team will be asked several questions in order to get a grip of wanted features. The following questionnaire will be used:

1. What would be the expected behaviour of a start, pause, stop, test and reset button?
2. How important is it to see the status of devices? (e.g. the current value of a scale)
3. What would you want to see in the operator interface?
4. Can players ask for hints in your escape rooms?
5. What type of devices do you typically use in escape rooms?
6. What type of actions must these devices be able to perform?
7. Do you want music/audio files loaded in with the configurations or prepared on the client computers?
8. Would you want devices in the room to connect over Bluetooth as well as over WiFi and LAN?
9. Can you name some more advanced devices an escape room can use (can there be more than one camera, sound systems and lighting)?
10. What data is important to store for a specific escape room? (e.g. previous team's scores/statistics)
11. What is the process of setting up an escape room and operating one which is in play?
12. What is the most important aspect of this system for you?

Three team members were asked these questions, all with different backgrounds and roles within the team. One did not have a lot of experience yet with managing these type of escape rooms, but does have a lot of experience with using the M.O.R.S.E. system and also designing puzzles for smaller escape rooms. Another person has a little bit of experience with working with the Popup system. He has used it once as operator, and built three physical escape rooms himself. Finally, the client himself was interviewed as well, who has a lot of experience and developed the old system. Their answers were analysed and the following conclusions were drawn.

2.3.1 Operator buttons

First of all, several phases in playing the game can be identified: pre-game, in-game, post-game and default. The pre-game and post-game phases should set a vibe in the room, where the post-game vibe might also depend on whether or not the players escaped. The default should be normal lights on and music off. The start button should start the game-changing from pre- to in-game phase.

The stop button should turn on the normal lights, shut down the music and open the doors. Considering the pause button, it can be important to understand when this would be needed. In case of emergency, the stop button might be used, but perhaps the pause button can be used for smaller cases where the players need to stop playing, allowing players to continue again. When the game is paused, users must not be able to solve any puzzles and the time should halt.

The reset button should reset all puzzles, however, on the implementation there are different possibilities when it should fire. If a post-game phase is part of the room, then the room should not immediately reset. However, if there is a pre-game phase, the room should be reset before the start button is pressed. This could perhaps be defined in the configuration: reset upon start, reset upon stop/finish, reset manually.

The test button should test whether all devices are working and should display warnings when a device is not connected or not reset properly. Furthermore, another button was suggested, the lost button to have a team 'lose' the game before time is actually up.

2.3.2 Status of devices

Displaying the status of devices (their values) can be a very nice to have feature, as it allows for operators to give specific hints. Currently, this can be done by seeing the feed from the camera in the room or listening to the recordings of a microphone in the room. However, seeing the actual status of how much weight is on a scale, for example, would be a good addition. For the display of the statuses, it might be easy to have a fold-down menu or an automatic sorting of devices which are 'active'/have been used most recently.

2.3.3 Operator interface

Aspects that the team wants to see in the operator interface: the video feed, the puzzle status (solved or not) and also why a puzzle has not been solved yet (which can be indicated by the device status). Finally, how long it takes players to solve a puzzle and where the operator's attention is needed to help the players.

2.3.4 Asking for hints

This would definitely be a wanted feature. Also, the possibility that hints are not shown straight away, but in a way the players can decide whether they really want to see a hint, e.g. the in-game screen showing "Hint?". However, when players ask for a hint, this might be subtracted from their time.

2.3.5 Type of devices and actions

From experience, the following devices were mentioned as previously used in escape rooms: buttons, sensors, switches, slider, turning wheels, lamps, RFID scanners, electromagnets, pumps, small motors, emergency button (power keeps the door locked, so pressing this button takes power off and opens the door), mi-lights, camera, microphone, televisions (show video when a specific puzzle is solved), led strips, lamps, actuators, system board, puzzle pieces that need to be matched.

2.3.6 Data storage

The time needed per room or puzzle would be nice to save, the number of hints a team used and the date. When an escape room would be hosted in a big school with lots of different groups, you would want to display the winner, but the messages should be customisable as different groups might react differently in seeing "you made it out" versus "you were three minutes slower than the front runner". Finally, the camera recordings could be saved for a limited amount of time (till the next run or for 24 hours). This way, after a group finishes (or runs out of time), the operator can go through the feed with them to show them what they missed or to discuss their thinking/cooperation process.

However, it is important to distinguish between different types of data storage. The data mentioned above is for statistics of an escape room, which the customer would think of as a nice to have feature, but not an important one. On the other hand, both the back-end as well as all client computers should keep track of their messages sent and received, and what actions they have performed. This should form a log, which can be saved separate from user data, and can provide a lot of information when debugging.

2.3.7 Other remarks

An additional wanted feature would be to link the system to the booking system used by Raccoon Serious Games, where players can receive an e-mail with their results and a photo.

From the perspective of a non-programmer, the system should be user-friendly for configuring an escape room.

Analysis Based on the answers given by the team, the following new requirements were established:

Must Have

- The back-end must allow for the reset to be done manually (on the front-end) or to be automated with the start or stop button implementation.
- The front-end must have a pause button which allows for continuity of the game but temporarily makes solving puzzles impossible and stopping the time.
- The front-end must give a warning of devices that are not connected; either constant or repeated at the start of a run.
- The front-end must have a test button, testing the connection of each device.
- The back-end must keep track of events that happen and messages send and received, writing these to a log text file to allow for the debugging of the system.
- A client computer must keep track of the messages send and received, writing these to a log text file to allow for the debugging of the system.

Should Have

- The back-end should allow for configurations to set pre- and post-game phases, which include light settings and music.
- The front-end should give a warning of devices that are not reset.
- The front-end should have a lost button, initiating the event that the group does not finish the escape room; regardless of the remaining time.
- The front-end should have a test button, testing the sequence of the devices.

- The front-end should have a foldable overview of the room's devices with their status, ordered by active.
- The system should allow for players to ask for hints.
- The back-end should store the date, time and number of hints of a team who played the escape room.

  Could Have

- The front-end could give an overview of the room's events that have been executed or not, with the time it took players to solve them.
- The system could have the option for the operator to ask the players if they want to receive a hint, before giving a hint.
- The front-end could play the recordings of a microphone in the room to the operator.
- The back-end could store the video recordings for a limited amount of time (one run or 24 hours), so the team can see their performance and cooperation afterwards.
- The back-end could have a connection to the booking system.
- The back-end could allow for Bluetooth connections to be made with client computers.

2.4 Home automation systems

Besides escape room systems, home automation system can also provide an interesting angle to provide additional features useful for S.C.I.L.E.R.. Home automation systems allow for users to develop their own Smart Home system, enabling users without programming experience to still program their home.

An example of such a home automation system is Hass.io[5]. It runs its back-end on a Raspberry Pi and uses the MQTT protocol to communicate with clients in the house. Furthermore, it already provides users with an interface to manage their homes, which can be seen in Figure 1. Finally, Hass.io allows for add-ons to be added, so the system can easily be connected to tools like Google Assistant or provide encryption.

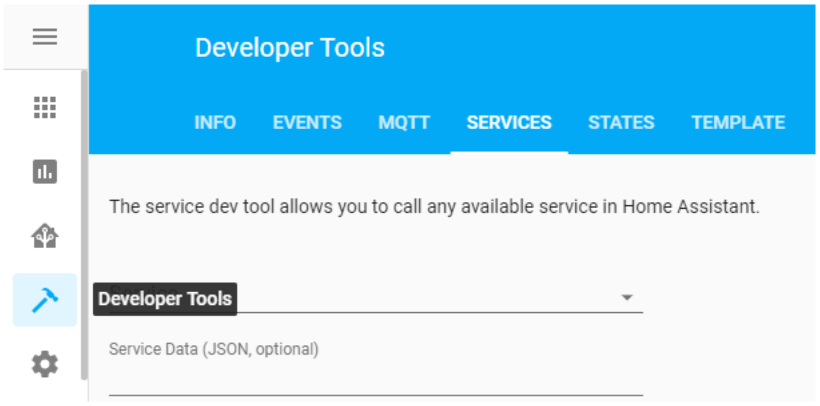

The service dev tool allows you to call any available service in Home Assistant.

Figure 1: Hass.io user interface

Examples of functionalities that can be implemented with the Hass.io system include: turning lights on when the sun rises, dimming the lights when a movie is watched or turning lights when for 10 minutes when a motion is detected in the room. The website of Hass.io also provides users with examples of how to program their automation.

[5]https://www.home-assistant.io/hassio/

Analysis This system is similar to an escape room system, in the sense that configurations determine how different devices react to each other. The features provided by Hass.io would definitely be possibilities that should be able to be implemented in S.C.I.L.E.R., although the current requirements should already cover this.

2.5 Conclusion

Although systems already exist for managing escape rooms, these are limited as using them means depending on their updates and software. Furthermore, one system required using their own hardware, while the other did not allow for the possibilities of hearing microphone recording from the room. Therefore, S.C.I.L.E.R. will be developed as a new system.

Several different requirements were derived in this chapter. These were each discussed with the customer and some were changed afterwards to a different priority or better defined. The final versions can be found in the Project Plan.

The most important take-aways were to keep the configuration as flexible as possible to allow for any type of device as well as allow all possible actions and combinations of actions. The features and requirements mentioned in this chapter can be very nice additions to the S.C.I.L.E.R. system, although most were not seen as must haves. Furthermore, previously derived must have requirements were defined in more detail. The must have requirements of logs was added for both back-end and client computers. Finally, many should and could have requirements for the front-end were derived, which might be added to S.C.I.L.E.R. if there is enough time within the project.

3 What is the best set-up to use to communicate between different components of an escape room system?

This research question was split up into multiple sub-questions, which are answered below. Each of these sub-questions is answered in the form of comparison. Finally, at the end of this chapter, in Section 3.5, a conclusion will be drawn on the different techniques, languages and frameworks chosen as they all work together and should be chosen coherently.

3.1 What is the best technique to use for communication between the back-end and client computers?

The best technique to communicate between the back-end and client computers should facilitate that the back-end can send instructions to client computers in such a way that they can be handled synchronously. Furthermore, the communication should be reliable as a disruption in the game would greatly impact the player's experience in a negative way. Finally, the delay of the instruction should not be noticeable for players.

3.1.1 Smart Home practises

The S.C.I.L.E.R. system contains multiple components that both require data from and send data to other components, and there are multiple models of network communication which such a system can use. To find common practice software patterns, one can look at Smart Homes. Home automation is very comparable to the escape room system to be developed, since they both practice the automation of physical objects like lighting, entertainment systems, and appliances, called an Internet of Things (IoT) application. Common software patterns in Smart Home application are request-response and publish-subscribe.

Request-response Request-response is a powerful messaging pattern for two machines to have a secure two-way conversation over a channel. One computer sends a request for data to the second and the second responds to the request. The request-response protocol is used in the client-server model, where the client sends a request and the server returns a response.

Because both sides only answer when asked, and the back-end always wants to know the current status of the client, the server requests data at regular intervals. This way the back-end is always aware of the most recent state of the client. For communication over a secure internal network, request-response is a reliable communication method, as argued by Rodriguez et al. [18].

Publish-subscribe With the publish-subscribe protocol, all communication between machines is processed by a broker, which receives a published message with a topic, and pushes this message to all subscribed machines interested in this topic [18].

This one-to-many system is used to notify changes of the state of a client to all interested other clients, like the back-end (and front-end), or directly to other interested clients. The other way around, the back-end sends event messages to the broker, which in turn sends it to the correct clients.

Quality of Service (QoS) is the measurement of the performance of these communication protocols, in which several related aspects of the network service are considered.

Both models can be used for synchronous communication between the back-end and clients. Commonly used Messaging Protocols in Smart Homes are the following:

MQTT Message Queue Telemetry Transport Protocol
The messages which usually consist of device data are sent asynchronously to the servers through a publish-subscribe architecture. Because of its simplicity, and a very small message header comparing with other messaging protocols, it is often recommended as the communication solution of choice in IoT. MQTT has low power requirements, making it one of the most prominent protocol solutions in constrained environments. It has a flexibility regarding Quality of Services with the given functionality.

XMPP Extensible Messaging and Presence Protocol
XMPP is based on Extensible Markup Language (XML), it implements the client-server protocol, but with extensions also allowing publish-subscribe interaction. The client starts an XML stream by sending an opening <stream> tag. The server then replies back to the client with an XML stream. Since XMPP is an open protocol, anyone can have their own XMPP server in their network without necessarily connecting to the internet. it does not have Quality of Service provision

DDS Data Distribution Service Publish-subscribe
DDS is also based on a publish-subscribe model, except it connects devices directly instead of via a server that has a broker, like MQTT. DDS is not very big in the community of Internet of Things yet, so implementing it will be hard as there is little documentation.

AMQP Advanced Message Queuing Protocol
AMQP is a publish-subscribe protocol, using a broker with a queue to ensure every message is delivered as intended. While providing a high QoS, a big downside is the high requirements in power, memory and processing, which a big disadvantage in IoT applications.

REST HTTP Representational State Transfer Hyper Text Transport Protocol
HTTP is the fundamental client-server model protocol using request-response messaging. REST-based interactions happen using constructs that are familiar to anyone who is accustomed to using the Internet's HTTP.

CoAP Constrained Application Protocol
CoAP implements the request-response model, using two layers. One layer uses HTTP-like methods to send requests, and it relies on its second layer for reliability, designed for re-transmitting lost packets.

WebSocket WebSocket

WebSockets use the same request-response handshake using HTTP, but after that, it stays open for full-duplex communication and has asynchronous communication.

Comparison All these protocols can be used for communication between the back-end and client computers. Reports that have compared some of these protocols properties, like speed and reliability, mostly conclude that one or the other is recommended depending on the specific application one wants to use it for, examples are Thangavel et al. [20], Atmoko, Riantini & Hasin [1] and Liu [17].

One report [7], comparing almost all named protocols in this chapter, concluded from a performance comparison of latency, bandwidth consumption and throughput, energy consumption, security, and developer's choice, that both MQTT and RESTful HTTP were the most sophisticated choices to consider for usage in IoT solutions. Since WebSockets were not compared in this survey, and no survey comparing RESTful HTTP and WebSockets could be found (except in energy consumption [11]), one can conclude from non-peer-reviewed articles[6][7][8][9] that MQTT is preferred over REST and WebSockets, and WebSockets over REST in usefulness in IoT. Languages that have a library for handling MQTT include, but are not limited to: C[10], Python[11], JavaScript[12], TypeScript[13], Go[14], C++[15], C#[16], and Java[17].

Analysis To conclude which of these protocols will fit the development of S.C.I.L.E.R., a design choice between software patterns for network communication must be made, while looking at the compatibility of the protocols with the rest of the system.

3.2 What language and framework is most suitable for maintaining communication with different client computers, while managing the escape room?

The back-end should be able to interpret a configurations file. It should use the configuration to manages all client computers in the network. The configuration describes the puzzles, devices, triggers and effects; all necessary information to run an escape room. So, when players finish a puzzle, the back-end should be notified by one or more of the client computers and react by, for example, instructing a client computer to unlock a door or play some music. This should be done fast and reliably, meaning fast enough that the players will not experience a notable delay or stuttering or another failure of one of the devices, more on this in Section 6. The language of the back-end should support the protocol for communication with the client computers as well as be easy to use for interpreting the configuration and managing the escape room's logic.

Another important consideration is whether to have a separate back- and front-end or have the front-end and back-end combined. An example of a combined set-up would be a Node.js server which manages a site while managing an escape room; and an example of a separated set-up is a Node.js server running the front-end which communicates with a different server, the back-end, which controls the escape room. The advantage of separating them is having more freedom in choosing different languages for the front- and back-end. Furthermore, when separating them, the front-end can be

[6] https://www.linkedin.com/pulse/internet-things-http-vs-websockets-mqtt-ronak-singh-cspo/
[7] https://www.educba.com/websocket-vs-rest/
[8] https://systembash.com/mqtt-vs-websockets-vs-http2-the-best-iot-messaging-protocol/
[9] https://coconauts.net/blog/2017/11/20/websocket-vs-rest/
[10] https://www.eclipse.org/paho/files/mqttdoc/MQTTClient/html/index.html
[11] https://pypi.org/project/paho-mqtt/
[12] https://www.eclipse.org/paho/clients/js/
[13] https://www.npmjs.com/package/mqtt
[14] https://github.com/eclipse/paho.mqtt.golang
[15] https://www.eclipse.org/paho/clients/cpp/
[16] https://www.eclipse.org/paho/clients/dotnet/
[17] https://www.eclipse.org/paho/clients/java/

seen as another client to the system instead of being a part of the back-end. An advantage to the combined set-up is having all information in one location, minimising the retrieval of information. A disadvantage to this approach is the vulnerability that when the front-end breaks down, the back-end can still run.

3.2.1 JavaScript

It is possible to write the back-end in JavaScript[18] with Node.js[19], which allows for a high performance, real-time application. This can be combined with other frameworks like Meteor, Express or Sails. JavaScript will work well on almost all systems since almost all browsers and machines support it.

3.2.2 TypeScript

TypeScript[20] is a superset of JavaScript. It converts to plain JavaScript which also allows it to run on almost all browsers and machines. TypeScript is in comparison to JavaScript strongly typed and allows for a better code structure and more object-oriented techniques.

3.2.3 Go

Go is also a possibility for the back-end. Go[21] is an efficient language with good support for JSON[22]. Furthermore, it has strong support for concurrency and parallelism. Furthermore, it is a statically, strongly typed language.

3.2.4 Java

Java[23] is a statically, strongly typed language that runs off the Java Virtual Machine (JVM). This allows Java to be cross-platform and thus run on almost all systems. It also has support for multi-threading.

3.2.5 Python

Python[24] is a dynamically, strongly typed language. It is an interpreted language that is often described as easy to read and learn. Python does support concurrency, however, does not implement true parallelism.

Comparison JavaScript and Typescript are very similar in syntax and abilities. However, there are some important differences [4]. JavaScript is an excellent language for Web development, but was not intended as server-side language, Node.js was developed as a solution to this. TypeScript was based off of JavaScript, and is strongly typed, allowing for better use of objects. TypeScript can do everything JavaScript can, and compiles in any JavaScript compiler. The main difference is JavaScript giving errors in RunTime, while TypeScript gives them during compilation. Furthermore, the asynchronous functionality provided by Node.js can work with both TypeScript and JavaScript.

Java and TypeScript offer similar possibilities as both are strongly typed language [6]. While Java is the better option for fully object-orient programming, TypeScript is still a bit more flexible and offers more possibilities for functional programming.

[18]https://www.javascript.com/
[19]https://nodejs.org/
[20]https://www.typescriptlang.org/
[21]https://golang.org/
[22]http://www.json.org/
[23]https://www.java.com/
[24]https://www.python.org/

Python is another language suitable for Web development, as it has a wide variety of libraries available to facilitate just about any use of the language. Go, on the other hand [5], has more built-in functionalities, although it allows for a lot of the same possibilities as Python. They are both used for Web development, although Go has proven additionally useful for micro-services and mobile applications. Go is considered more compliant with concurrency than Python. Finally, as Go is compiled automatically, it throws errors during compilation, while Python has more runtime errors.

When comparing Go and TypeScript [10], they both come out as compelling languages. Performance-wise, Go is a very fast language, while Node has asynchronous functionality and can perform small tasks in the back without affecting the main thread. Furthermore, Node.js is reusable and widely used for applications requiring real-time data updates, as an IoT system does too. Go is stronger in handling concurrency, and also supports parallelism, where Node.js runs on a single thread. The M.O.R.S.E. system is fully written in TypeScript, and this system also handles concurrency with several connected clients.

Finally, all languages have libraries to support different communication protocols, so this is not a condition for choosing any language in particular. The full team has a lot of experience in Java, two team members with Python and one with Go.

3.3 What language can be used to code the library for the client computers to ensure proper instruction handling and easy to control the connected device?

The client computers must be able to receive instructions from the back-end, decide whether the message applies to them and act upon it. This means they should be able to instruct the client computer to change the state of its connected device. A library must be developed to handle this.

Raccoon Serious Games developed a Python library to handle these requests in the Popup system. Shete and Agrawal [19] also propose the use of Python for a library on a Raspberry Pi to communicate with an MQTT Protocol instructing the Pi. The same goes for Grgić and Heđi [8]. As Weychan, Marciniak and Dabrowski [21] argue, Python is a very efficient language to use for Raspberry Pi because there are a lot of packages for algebra and signal processing available, making the embedded systems faster.

For the use of Arduinos in IoT systems, similar libraries can be found. Barbon et al. [2] provide a service for Arduinos to easily add new capabilities to the micro-controller. They provide libraries for Java, Python, Racket and Erlang. Another application uses a Thinger.io library for its Arduino, which is an Internet of Things platform.

Analysis So, although several different languages are used, Python seems to be a recurring and efficient option. Furthermore, the Pis and micro-controllers the customer already has, all run on Python too.

3.4 What language can generate a user-friendly interface on the mobile device of the operator, while communicating with the back-end?

The front-end should be accessible by mobile devices. The escape room operator will use this interface for managing an escape room by watching a live feed as well as giving hints and configuring the room. To accomplish this, there are two possibilities: a (native) mobile application or a mobile-friendly Website. Since the front-end has to work on both Android and iOS devices, one native mobile application is not an option. Instead, a mobile-friendly website or a progressive Web app (PWA) would be a good option, as well as a hybrid application.

Below three of the most-used JavaScript front-end frameworks are compared to each other, as well as to Python Django.

3.4.1 Angular

Angular[25] is an open-source Web application framework. It is developed by a team at Google. This framework is used to make modular (progressive) Web applications and supports TypeScript which is its main language. When compared to React and Vue, Angular has a steeper learning curve [22].

3.4.2 React

React[26] is a by Facebook developed, JavaScript library for building user interfaces. Since React is only a user interface library, it is not as self-sufficient as Angular.

3.4.3 Vue

Vue[27] is an open-source framework for building user interfaces and single-page applications. It uses JavaScript like React, but can also be used with TypeScript.

3.4.4 Django

Django[28] is a Python framework, which allows for fast Web development and supports the user in designing their Web pages. It facilitates both front- and back-end development.

Comparison All of these frameworks would work well for a mobile-friendly Web application, which also allows for making a PWA. Django, Angular and React are backed by big companies ensuring long-term stability and support, where Vue is backed by an open-source community which has the advantage of not having decisions based on commercial interests.

All of the JavaScript frameworks can be combined with a framework like Ionic[29], an open-source SDK, to make a hybrid mobile application. A hybrid application would be desired if, for example, the application should be able to work when not connected or if more native features are needed. Since this is not the case, the choice was made to create a mobile-friendly website and possibly make it a PWA.

Although both JavaScript and Python can be used for the front-end, JavaScript is much more efficient for front-end as Python is very focused on back-end development. A framework like Django solves part of this, but a well-working front-end language would be preferred over a framework.

React is more a library than a framework, in contrast to Vue and Angular. All three support TypeScript as well as JavaScript, although for Angular it is the standard language.

3.5 Conclusion of the system set-up

The first decision made is to have a separate front- and back-end. Mostly because this gives a lot of flexibility to the system. Furthermore, this ensures a working back-end when the front-end is down. While running an escape room, the back-end is most important. If the front-end were to break down during a run, the back-end can still function and although the operator loses some functionality, the players can still continue.

The front-end will be a mobile-friendly website, possibly a progressive Web app, because it will always be accessed from the local network and should work on an Android or iOS system for the operator. JavaScript turned up as an efficient language for this, with multiple well-working frameworks to support it.

[25] https://angular.io/
[26] https://reactjs.org/
[27] https://vuejs.org/
[28] https://www.djangoproject.com/
[29] https://ionicframework.com/

As the M.O.R.S.E. system from Raccoon Serious Games is developed using the Angular framework, this is an advantage over the other frameworks mentioned in Section 3.4 to ensure more consistency over the companies different systems. All the frameworks would work equally well, and this advantage gave the tipping point to select Angular. Although Angular does have a steeper learning curve than the other options, this was not a strong enough reason to opt out of choosing as it is a widely supported framework. TypeScript is the standard language for Angular, so the front-end will be developed in Angular using TypeScript. Additionally, it is a superset of JavaScript, which allows for object-oriented programming.

The communication protocol chosen is MQTT. With the choice between publish-subscribe and request-response, it turned out that publish-subscribe suits the needs of S.C.I.L.E.R.. As it is a one-to-many system, the back-end can easily communicate with all necessary client computers, as well as the front-end. It is also seen a lot in IoT systems and provides a good Quality of Service. Considering publish-subscribe, the most supported protocol is MQTT, which is also a recommended communication solution for IoT systems and has a range of libraries supporting languages of the other parts in the system.

The client computers will have a library written in Python, as this works well with the Raspberry Pis and other micro-controllers used in escape rooms and existing Pis of the customers already run in Python.

Finally, the decision for the back-end language was most important. The two front-runners are Go and Node with TypeScript. The system will already be developed using two different languages, so having one consistent language throughout the whole system is no longer an option. The customer values concurrency and parallelism a lot as the client computers must be able to communicate concurrently to handle all instructions properly. One team member has experience with Go, while the customer's other system, M.O.R.S.E., is developed in TypeScript. In the end, although both are strong options, as concurrency was seen as most important, and having some experience with the language weighed up against adding another language to the project, it is decided to use Go.

A summary of the design choices is visualised in Figure 6 in Appendix D.

4 What type of hardware is small enough to hide in the room, while it is also suitable as a reliable back-end, communicating with multiple client computers?

The hardware of the back-end should be compact in order to hide it in the escape room. Furthermore, it should be powerful enough to handle the communication with all client computers. There exist many different types of hardware, so it is important to select one that fits the requirements for this project.

The QUEEN[30] escape room system uses a circuit board, allowing for many connections to be made to connect different devices to the board. It has an Arduino Mega 2560 processor with 12V power voltage.

Smart Homes have a similar set-up as an escape room system, as they connect multiple devices which must be controlled remotely, and are thus IoT systems, like S.C.I.L.E.R. will be. The Smart Home system with LabView [9] uses both RF modules (connecting with radio waves) and IR systems. The remote control unit they use is a PIC16F877A. The CASAS Smart Home [3] uses a ZigBee wireless bridge to communicate with client computers. As Liu [14] stated, the ZigBee platform is based on the ARM920T core S3C2410X microprocessor, where the CC2430 chip functions as a wireless receiver. Zhai and Cheng [23] argue for the use of embedded systems for Smart Home applications. They use a Liod platform as the main controller, with an IntelXScale PXA270 as processor.

When using an Arduino, the ESP8266 chip[31] can be used to run Arduino functions and libraries

[30]https://escaperoomdoctor.com/queen/hardware
[31]https://github.com/esp8266/Arduino

directly on the Arduino without requiring external micro-controllers. It includes libraries to communicate over WiFi, TCP, UDP, HTTP, mDNS, SSDP and more.

The Popup system is run on a Raspberry Pi. This machine usually works as intended, but according to the customer, the Pi would crash after a long period of running the system. The customer was not able to find out the reason for this crash, it could be the back-end system or the Pi. So, a device similar to a Raspberry Pi but a little bit more stable would run the system fine. A couple of options in the same price range that are also compatible with Python are: Asus Tinker Board, ODroid XU4, Banana Pi-M64, Rock64.

Conclusion As a Raspberry Pi is easily available to the developers and the customer could not pinpoint any problem with using it, the system will initially be run on a Raspberry Pi. During development, tests will be done on the system to determine whether the Raspberry Pi is stable enough to handle it. These will also include tests with the system running for a longer period of time to see if it does not crash. If these tests turn out unsuccessful, then a new device will be bought to see if it holds up to the tests better than the Pi. A device mentioned above can be used as an option for this.

5 What configuration format or template contains all necessary information, can be written by a non-programmer and can be interpreted by the system?

5.1 M.O.R.S.E. system configuration

The configuration of the escape room is very important. This includes the indication of puzzles and sensors and how players can progress in their game. It specifies what puzzles need to be solved, and in which order, for the players to actually win the game.

A good example of a system which uses a similar configuration is M.O.R.S.E, a product of Raccoon Serious Games for large-scale escape events. This system was studied focusing specifically on their configuration of the escape event, and analysed to determine which parts of their configuration can be useful for the new system.

The M.O.R.S.E. system allows admin users to export different components of the system to a CSV file, for which they use a JSON to CSV converter. Configurations can be entered through an interface on the admin side or by entering a JSON file. The JSON is converted to their MongoDB objects using Astronomy, which uses a JSON-like structure.

5.1.1 RuleSets

The most important configuration setting that applies to the S.C.I.L.E.R. system is the RuleSet. The RuleSet has the following properties: id, name, eventId, limit, executed, triggers, conditions and actions. There is an id to identify each rule, a name to make it easy to read for humans and an eventId to specify to which event the RuleSet belongs. This is because the M.O.R.S.E. system allows for several events to be scheduled at the same time. Then, there is a limit, which is the maximum number of times this rule can be executed and an executed parameter to specify how often it has been executed.

The three most important attributes are the triggers, conditions and actions. A trigger is defined by a type of trigger, which specifies the necessary attributes. For example, a PuzzleStatusTrigger includes an eventId and puzzleId, where giving an answer to the puzzle of a certain event triggers this RuleSet.

A condition is defined by a type of condition. It includes a list of constraints, which in turn have a type, defining the other fields. Two examples are the PuzzelAnswerConstraint with eventId, puzzleId, comp (comparison type: '==' or 'contains'), answer and exact; and the PuzzelStatusConstraint with eventId, puzzleId, check (boolean) and field (solved, opened or picked up).

Finally, the actions are defined by a type (who does it apply to: team, project, etc). For a TeamAction, there is an eventId and a list of acts. These acts have a type, for example PuzzleStatusAct, which requires an eventId, puzzleId, result and field, which enables the team to pick up the next puzzle.

For the M.O.R.S.E. system an example of a RuleSet from a real escape room is:

1. name: Team chooses game Security
2. trigger: Puzzle with question: "Which hubs can support all minerals", which has multiple correct answers, of which Security is one
3. conditions: the answer to the puzzle equals 1 and the status of the puzzle is solved
4. actions: the next puzzle with a question related to Security is opened to the team

5.1.2 Scheduling

The M.O.R.S.E. system includes a schedule where the admins can see an overview of the full escape event. It shows the different phases of the event and the puzzles which belong to each event. As the escape events include multiple groups and clusters, some phases may have different puzzles which are not all done by all groups.

5.1.3 Other configurations

The M.O.R.S.E. system also includes more general settings for an escape event, including items specific to the M.O.R.S.E. system, like projector settings, but also audio settings and general settings for each event (start time, remaining time, suspended message etc).

Analysis It is very nice to see a similar system with a configuration which has the same goal as the S.C.I.L.E.R. system. The M.O.R.S.E. developers started out with reading in JSON files to set the configuration and added a manual interface for configuration later.

Although there are some differences in implementation of the systems, the core of how the configurations are set are definitely applicable to the S.C.I.L.E.R. system. Events will not be needed in the S.C.I.L.E.R. system and the different options specified per type are also different. For example, M.O.R.S.E. has specific actions for allowing new puzzles to be picked up which would not be necessary for S.C.I.L.E.R..

M.O.R.S.E. seemed satisfied with a JSON format, which would be a good option for the new system. Furthermore, the use of trigger, condition and action to specify what needs to happen can work very well. Especially because a trigger can specify what puzzle, in the case of S.C.I.L.E.R. device, can trigger a rule and then the condition specifying if something really needs to happen. Then, if those two match, an action can be performed.

5.2 Home automation configuration

The Hass.io[32] system was already studied in Section 2.4. In this section, its configurations will be analysed.

Users should provide an automation file, which has the following format:

[32] https://www.home-assistant.io/hassio/

```
automation:
    - alias: Turn on kitchen light when there is movement
      trigger:
        platform: state
        entity_id: sensor.motion_sensor
        to: 'on'
      action:
        service: light.turn_on
        data:
            entity_id: light.kitchen_light
```

Each 'rule' is identified by an alias. It then includes a trigger and an action. A trigger is always defined by a platform. This can be one of the following: state, event, homeassistant, mqtt, numeric_state, sun, template, time, time pattern, webhook, zone or geolocation. Furthermore, different triggers can be defined per automation rule.

To specify some of the platforms: zone is used for identifying zones within the house and the trigger requires an entity_id, zone and event (enter or leave); time just takes the paramter at, to define an action at a specific time; time pattern can specify, for example, *minutes* : 30 means every 30 minutes past the hour, but also *minutes* : /5 meaning every 5 minutes; numeric_state defines an entity_id, above and/or below .

Actions are specified by service and data, they can also activates so-called scenes, providing certain features. The service can specify an action on a certain device, providing an entity_id, and by adding data can allow a certain state to be set on the device. Finally, a delay can be added to an action.

Finally, besides triggers and action, also conditions can be specified. It start with condition (and/or) and can then define conditions in a similar fashion to triggers.

Analysis The Hass.io configuration does not use JSON, but a format similar to JSON, though with less syntax. The trigger, condition and action format is a little similar to M.O.R.S.E., though it uses different pre-defined types (platforms) and the use of conditions is slightly different.

Conclusion Just like the M.O.R.S.E. system, S.C.I.L.E.R. will also be implemented JSON configuration files. Although this requries the user to provide more syntax, it does allow for easy handling of the configuration file on the back-end, since most languages have standard JSON libraries. Configuraiton will use rules with triggers, conditions and actions. In the M.O.R.S.E. system this works very well for a similar purpose and the Raccoon Serious Games team is also used to these. Both studied systems offer interesting examples of how a configuration can be set up efficiently.

6 What is the latency threshold that the system can allow to avoid hindrance to the players?

The delay of communication in an escape room system should be very minimal. Players should not notice a delay in the system from processing their input. Using the definition of an event with a trigger, condition and action from the Glossary in Appendix A, delay is defined by the moment a trigger is prompted to the moment the back-end registers the action as performed. This is also referred to as the latency.

In Figure 2, the results from an experiment by Lee, Kim, Hong & Ju [13] measuring MQTT loss and delay can be seen. They analysed three different QoS layers and measured the end-to-end delay in a wired with respect to the size of the payload. In Figure 3, the results of their experiment in a wireless network can be seen.

The most important reason to keep the delay below a threshold is that users can suspect a malfunction if an action following an event takes too long. However, defining this threshold is difficult.

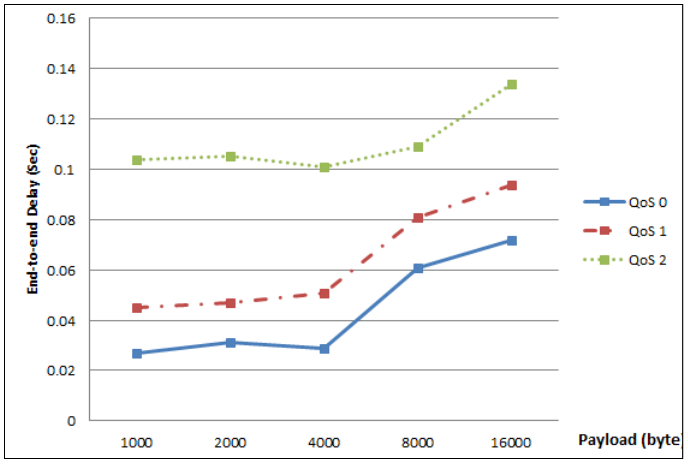


Figure 2: Wired Network Mean End-to-End Delay [13]

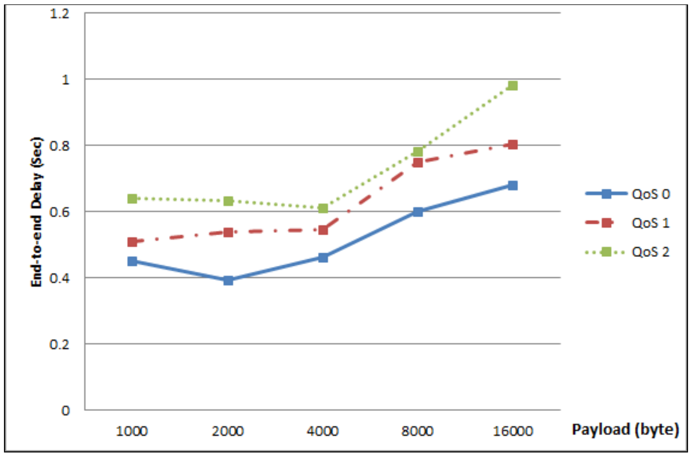


Figure 3: Wireless Network Mean End-to-End Delay [13]

Not all players will have the same experience of delay or the same expectations of when the delay is 'too long'.

Latency requirements specify that a predefined latency value should be kept with a certain probability [12]. As argued by Jiang et al. [12], the predefined value depends on the application. For example, delivering an e-mail can take a few seconds, but when calling someone, the user does not want to wait seconds before hearing an answer. Jiang et al. have listed the latency requirements for several different types of applications, which can be seen in Table 1 [12].

As seen from the table, the differences can be quite large. An escape room system will not have a specifically high data rate, as the number of client computers in a room should typically be around 20. Each of these client computers will communicate over MQTT, which uses a heart beat updating of 60 seconds. Additionally, each client computer will message when its connected device changes value and confirmation messages are sent once.

The control traffic in smart grid is aimed at monitoring, communicating and computing abilities of a smart grid [12]. This is probably the application that most closely resembles an escape room system. Therefore, the goal of the S.C.I.L.E.R. system is to keep the latency below the threshold of 100 ms. As mentioned in a blog on Noction [16], the rule of thumb for latency is that users start noticing it, when it exceeds 100 ms. Katherine Lynch from Data Center [15] argued a rule of thumb

	Latency	Reliability	Other
Virtual reality	1 ms	-	high data rate
Automated guided vehicle	few ms	99.99999%	high data rate
Financial market	few ms	high	-
Exoskeletons and Prosthetic hands	few ms	high	-
Tele-surgery	1-10 ms	98%	high data rate
Protection traffic in smart grid	1-10 ms	high	-
Factory automation	1-10 ms	99.9999999%	-
Control traffic in smart grid	100 ms	high	-
Process automation	100 ms-1s	99.9999999%	-

Table 1: The requirement for low-latency applications (reliability marked with high is due to no availability of precise numbers)

of 300 ms for round-trip communication to the maximum before users start noticing a delay.

Conclusion So, keeping the latency below the threshold of 100 ms should suffice in keeping up the user experience by no notice of delay. This was taken from a comparison between different type of systems and also turned up as a general rule of thumb for users not noticing a delay.

7 What are the best tools to use for keeping a consistent code base?

High code quality is required to keep the code base clean, understandable, maintainable and extendable. To assure this code quality, the team will be implementing tools for enforcing code style and testing. Furthermore, each piece of code will also be reviewed by another team member.

7.1 CI/CD

There will be three different components of the system, which will all be developed in the same GitHub repository, where different modules will be created for the different components. This also allows the repository to have just one CI/CD tool running, which should be configured to check all modules. As the customer has his own private GitHub, the S.C.I.L.E.R. system will be developed in a repository from this account, so he always has access to it.

Travis[33] and Jenkins[34] are two popular CI/CD tools. There are some differences[35], but both are widely supported. However, as Jenkins requires an external server, the tool to be used is Travis. As the repository on GitHub can be made public, Travis will be free to use.

7.2 Front-end

Since the front-end will be written in TypeScript, TSLint[36] will be used since it is a static analysis tool for TypeScript that checks the readability, maintainability, and functionality of the code base, also spotting errors early on. It has configurable style rules, so the developers can enforce custom additional style rules or exceptions. Prettier[37] is a code formatter, which will not only check, but also

[33]https://travis-ci.org/
[34]https://jenkins.io/
[35]https://www.keycdn.com/blog/jenkins-vs-travis
[36]https://palantir.github.io/tslint/
[37]https://prettier.io/

rewrite wrongly formatted code. This can work in tandem with TSLint. These tools will help to keep the code style consistent among the different team members.

For testing, Jasmine[38] will be used, since it is a testing framework for JavaScript, which includes a mocking library. As TypeScript is based off of JavaScript, this should work fine too. To run all tests automatically, Karma[39] will load an environment and also find and run all tests. Karma also has support for various continuous integration servers, including Travis[40].

7.3 Back-end

Gofmt[41] and Golint[42] will both be used to ensure code style is consistent. Golint prints out mistakes in the code style and Gofmt will reformat the code, fixing spacing for example.

For the testing of communication between the different components of the system, the connections will have to be mocked. Testify[43] is a testing package for Go which has easy assertions, mocking and a testing suite for interfaces and functions.

7.4 Client computer library

Flake8[44] will be used to check the code style and make sure all code conforms to the Python style guide (PEP8). Furthermore, Black[45] is the code formatter that will be used. Other tools are available for this purpose, however, one team member already has experience with these tools.

Unittest[46] is a testing framework for Phython, inspired by JUnit. This framework will be used since it supports mocking, which will be needed to mock the connection with the back-end.

Conclusion The repository will use Travis for continuous integration, as this is a well supported and documented tool. The coding tools chosen are TSLint, Prettier, Jasmine and Karma for the front-end, which are popular tools that satisfy the needs of the team for these tools. For the back-end, Gofmt, Golint and Testify are chosen as they work well for Go and provide the functions desired by the team. Finally, Flake8, Black and Unittest will be used for the client computer library, as a team member already has experience with these, and the provide the required functionality.

Conclusion

From reviewing the Popup system from Raccoon Serious Games, a few conclusions were drawn about how to implement S.C.I.L.E.R. compared to the Popup system. S.C.I.L.E.R. will be developed as a fully new system, because Popup was not documented enough to build further on. Considering the front-end, the most important conclusions were to restrict user input in fields, have clear descriptions and provide feedback whether or not an action has succeeded. For the back-end, it was concluded that the implementation of tracking connected client computers should be done by keeping a list of devices that should be in the room, while updating their connections status. The client computers are already pretty solid, and the S.C.I.L.E.R. system should keep the format of a library with an abstract handler and concurrent instruction handling.

After discussing with the Raccoon Serious Games team and reviewing existing systems, several requirements were drawn up that could be either nice to have in the S.C.I.L.E.R. system, but also

[38]https://jasmine.github.io/
[39]https://karma-runner.github.io/latest/index.html
[40]https://travis-ci.org/
[41]https://golang.org/cmd/gofmt/
[42]https://godoc.org/golang.org/x/lint/golint
[43]https://godoc.org/github.com/stretchr/testify
[44]http://flake8.pycqa.org/
[45]http://flake8.pycqa.org/
[46]https://docs.python.org/3/library/unittest.html

a few features were discovered which would be very good contributions and should really be in the system. The product backlog in the Project Plan was updated after considering all possible new requirements mentioned in Section 2 and discussing these with the customer to decide which ones to actually include.

To answer the question of the best set-up for the communication system, the following choices were made. There will be a mobile-friendly website as a front-end, separated from the back-end, written in Angular with TypeScript. The communication protocol will be MQTT. This facilitates the communication between the front-end, the Python library running on the client computers, and the back-end implemented in Go.

For the hardware to run the back-end on, the choice was made to see how well the new system will run on a Raspberry Pi. The biggest problem with the Popup system is that the Pi would crash after running for a long time, but it was never determined whether this was a problem with the Pi or the actual system. So, tests will be done with the Pi, also over longer periods of time, and if these turn out to be unsuccessful, a new hardware will be chosen, from the ones mentioned in Section 4.

The configurations from the M.O.R.S.E. system turned out to be pretty applicable to the S.C.I.L.E.R. system, especially RuleSets to define the game flow. JSON will be used as the language to define the configurations in, and a guide will be provided to specify which tags need to be present in a configuration.

The latency threshold that can be allowed in the system was found to be 100 ms, after comparing an escape room system to several applications' latencies. This was also found to be the rule of thumb for the threshold of users noticing the latency .

Finally, the repository will use Travis for continuous integration. The coding tools chosen are TSLint, Prettier, Jasmine and Karma for the front-end; Gofmt, Golint and Testify for the back-end; and Flake8, Black and Unittest for the client computer library.

A Glossary

- A user is an escape room operator.
- A room is a full escape room which might consist of several physical rooms.
- A client computer, like a Raspberry Pi, controls a device.
- A device is a physical object in the room, like a lamp.
- The status of a device is the value which it currently holds, for a lamp this can be on/off, for a scale this can be a certain weight.
- An event specifies what happens in the escape room, by defining triggers, conditions and actions.

  * The trigger of an event is another event, a device or the time of the escape room.

  * The condition of an event is the checking of a certain value from a device or result of an event.

  * The action of an event is the changing of a device's status or the trigger of a next event.
- Input to the back-end is a message of a client computer or the front-end.
- The label of a device specifies it belonging to a certain group, like 'hint' to specify all devices that can give a hint.
- The pre-game phase is the mood set in the room when players enter, but the game has not started yet. This includes lighting and music.
- The post-game phase is the mood set in the room when players have finished the game. This includes lighting and music. Two different ones can be set, one when players have escape and one when time runs out.
- The reset phase is the default lighting of the room, in which the room can be reset.

B Repository structure

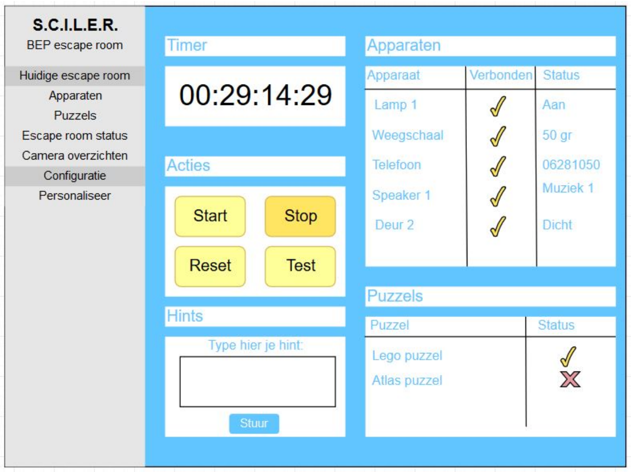
This is a concept set-up of the repository structure.

```
root/
├── front-end/
│   ├── e2e/
│   ├── node_modules/
│   └── src/
│       ├── app/
│       ├── assets/
│       └── environments/
├── back-end/
│   ├── src/
│   └── test/
├── cc-library/
│   ├── src/
│   │   ├── scripts/
│   │   │   ├── device.ini
│   │   │   └── start_device.py
│   │   └── device/
│   │       ├── app.py
│   │       ├── base_handler.py
│   │       ├── config.py
│   │       └── connection.py
│   └── test/
└── broker_resources/
```

Figure 4: Repository structure

C Front-end design

Having studied the Popup system, conclusions have been drawn on what to include in the front-end and not. Having studied the M.O.R.S.E. system, conclusions have been drawn on how the front-end should look, to ensure consistency across the different products of Raccoon Serious Games.



Apparaat	Verbonden	Status
Lamp 1	✓	Aan
Weegschaal	✓	50 gr
Telefoon	✓	06281050
Speaker 1	✓	Muziek 1
Deur 2	✓	Dicht

Puzzel	Status
Lego puzzel	✓
Atlas puzzel	✗

Figure 5: Front-end design

D Structure

This diagram is a visualisation of the design choices regarding systems, languages, frameworks and components as decided in Section 3.5.

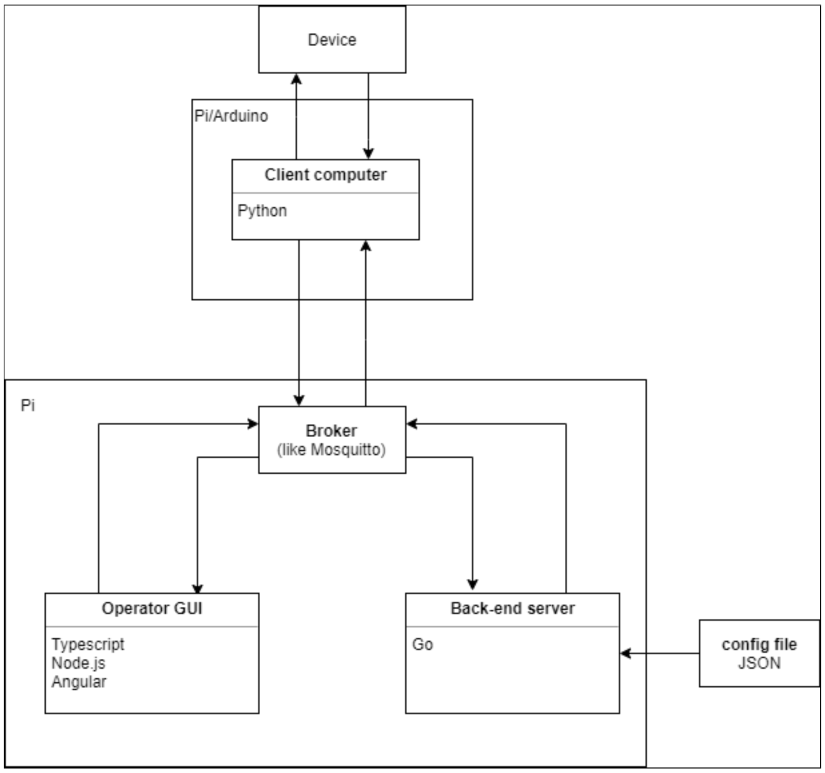


Figure 6: Design choices of the project structure

References

[1] Rachmad Atmoko, R Riantini, and M Hasin. Iot real time data acquisition using mqtt protocol. *Journal of Physics: Conference Series*, 853:012003, 05 2017.

[2] Gianluca Barbon, Michael Margolis, Filippo Palumbo, Franco Raimondi, and Nick Weldin. Taking arduino to the internet of things: the asip programming model. *Computer Communications*, 89:128–140, 2016.

[3] Diane J Cook, Aaron S Crandall, Brian L Thomas, and Narayanan C Krishnan. Casas: A smart home in a box. *Computer*, 46(7):62–69, 2012.

[4] Difference between javascript and typescript. *Javapoint*. Retrieved from: `https://www.javatpoint.com/javascript-vs-typescript`.

[5] Differences between to python vs go. *Educba*. Retrieved from: `https://www.educba.com/python-vs-go/`.

[6] Comparing typescript to java. *Beyond Java*. Retrieved from: `https://www.beyondjava.net/comparing-typescript-java`.

[7] Jasenka Dizdarević, Francisco Carpio, Admela Jukan, and Xavi Masip-Bruin. A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. *ACM Computing Surveys (CSUR)*, 51(6):116, 2019.

[8] Krešimir Grgić, Ivan Špeh, and Ivan Heđi. A web-based iot solution for monitoring data using mqtt protocol. In *2016 International Conference on Smart Systems and Technologies (SST)*, pages 249–253. IEEE, 2016.

[9] Basil Hamed. Design & implementation of smart house control using labview. *Design & implementation of smart house control using LabVIEW*, 1(6), 2012.

[10] Liliia Harkushko. Node.js vs go: Which is better for backend web development? Retrieved from: `https://yalantis.com/blog/golang-vs-nodejs-comparison/`.

[11] Volker Herwig, René Fischer, and Peter Braun. Assessment of rest and websocket in regards to their energy consumption for mobile applications. In *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, volume 1, pages 342–347. IEEE, 2015.

[12] Xiaolin Jiang, Hossein Shokri-Ghadikolaei, Gabor Fodor, Eytan Modiano, Zhibo Pang, Michele Zorzi, and Carlo Fischione. Low-latency networking: Where latency lurks and how to tame it. *Proceedings of the IEEE*, 107(2):280–306, 2018.

[13] Shinho Lee, Hyeonwoo Kim, Dong-kweon Hong, and Hongtaek Ju. Correlation analysis of mqtt loss and delay according to qos level. In *The International Conference on Information Networking 2013 (ICOIN)*, pages 714–717. IEEE, 2013.

[14] Zhen-ya Liu. Hardware design of smart home system based on zigbee wireless sensor network. *Aasri Procedia*, 8:75–81, 2014.

[15] Kathryn Lynch. Is your network ready to handle videoconferencing? *Data Center*. Retrieved from: `https://www.techrepublic.com/blog/data-center/is-your-network-ready-to-handle-videoconferencing/`.

[16] Network Intelligence Noction. Network latency and its effect on application performance. Retrieved from: `https://www.noction.com/blog/network-latency-effect-on-application-performance`.

[17] Guilherme M. B. Oliveira, Danielly C. M. Costa, Ricardo J. B. V. M. Cavalcanti, Josiel P. P. Oliveira, Diego R. C. Silva, Marcelo B. Nogueira, and Marconi C. Rodrigues. Comparison between mqtt and websocket protocols for iot applications using esp8266. *2018 Workshop on Metrology for Industry 4.0 and IoT*, pages 236–241, 2018.

[18] Carlos Rodríguez-Domínguez, Kawtar Benghazi, Manuel Noguera, José Luis Garrido, María Luisa Rodríguez, and Tomás Ruiz-López. A communication model to integrate the request-response and the publish-subscribe paradigms into ubiquitous systems. *Sensors*, 12(6):7648–7668, 2012.

[19] Rohini Shete and Sushma Agrawal. Iot based urban climate monitoring using raspberry pi. In *2016 International Conference on Communication and Signal Processing (ICCSP)*, pages 2008–2012. IEEE, 2016.

[20] Dinesh Thangavel, Xiaoping Ma, Alvin Valera, Hwee-Xian Tan, and Colin Keng-Yan Tan. Performance evaluation of mqtt and coap via a common middleware. In *2014 IEEE ninth international conference on intelligent sensors, sensor networks and information processing (ISSNIP)*, pages 1–6. IEEE, 2014.

[21] Radoslaw Weychan, Tomasz Marciniak, and Adam Dabrowski. Implementation aspects of speaker recognition using python language and raspberry pi platform. In *2015 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, pages 162–167. IEEE, 2015.

[22] Eric Wohlgethan. *SupportingWeb Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue. js.* PhD thesis, Hochschule für Angewandte Wissenschaften Hamburg, 2018.

[23] Yanni Zhai and Xiaodong Cheng. Design of smart home remote monitoring system based on embedded system. In *2011 IEEE 2nd International Conference on Computing, Control and Industrial Engineering*, volume 2, pages 41–44. IEEE, 2011.