LifeDrop Media Manager

Department of Computer Science
OST – University of Applied Sciences
Campus Rapperswil-Jona

Autumn Term 2020

Author(s):	Robin Elvedi, Lukas Schiltknecht
Advisor:	Thomas Corbat
Project Partner:	LifeDrop, Davos

Lifedrop Media Manager

Table of Contents

1 Abstract

We extend the features of the lifedrop pod, a deck-chair like device with a closable hood, large screen and speakers, by adding a web based media management user interface, called the *media manager*. As part of that process we also remodel the *media service*, the component responsible for playing media inside the pod.

The goal is to enable easy management and playback of videos and music inside the pod, and to a smaller extent, the goal is to redesign of the pod controller user interface used to control the pod.

We incorporate a continuous integration workflow with automatic test execution and coverage reporting for code stability. Integration tests are used to guarantee the interoperability of various components.

The end result is an intuitive media management user interface with an appealing design as well as a drop in replacement for the existing media service.

We finish our work by integrating the media manager and the media service into the existing lifedrop pod on site.

2 Management Summary

This chapter will sum up the project activity in a less technical fashion. The first chapter deals with the existing system called the LifeDrop pod. Then the existing software and its features will be described, motivation and goals of the project are explained and the results are presented in a commonly comprehensible manner.

2.1 The Pod

The LifeDrop pod is a device that aims at giving refuge from stress to people in crowded areas. It is a chair contained in a hooded compartment that shields the user from visible and partly audible outside influences to create a relaxing environment detached from the hectic outside world. To help a person letting go of the current thought process, there is a media library with movies and songs that can be played. This creates a small bubble of peace in an otherwise stressed environment like for example in an airport, on a harbor, in a sky scraper or other places with few retreat options.

This device is comparable to other media pods, with the unique selling proposition of comfort and luxury. The prototype was already up and running at the start of the project and the requirements analysis started with a test run. Then management and the development team set up a list with priorities to work on for the term project. On the top of this list was a software to manage the media on the pod. To achieve this feature the *media manager ui* and the *media server* were created over the course of 14 weeks.

2.2 Existing Solution

As mentioned previously, the *pod control ui* had already been implemented. In the original version, the application developer had to update the list of media links manually and deploy the application again every time the media material was changing. This was taking a lot of time, since the deployment had to be done on site. Additionally, the design of the application was kept functional and organized just as the visual appearance was.

In the implementation of the software there were minor details that were implemented in a quick and functional way, which could have lead to bigger problems in the future. The whole security concept is based on a WPA2 password. The system consists of two raspberry pi devices linked together via a router. One of these devices handles the engine of the pod and one handles the audio and video experience.

2.3 Motivation

The main motivation of improving the existing solution was to enable the pod admin to manage media without knowledge of programming languages, JSON syntax or the deployment process. Additionally, the design was reworked, so that the two applications appeared more luxurious, less functional and still recognizable as one unit. This meant that some of the features had to be displayed in a different way.

Cleaning up the code base was part of the process and applying the principles taught at OST was also an implied advantage of developing the solution as a term project. This included the structure of software as well as the communication among software components.

2.4 Goals

The project had two main goals, the first was to develop an application to manage the media on the pod via web browser called *media manager ui* and the second one was to redesign the *pod control ui* to communicate the unique selling proposition more accurately. This implied setting up a second backend to handle the media files, media entries and collections called *media server*, developing a frontend application offering the features requested and redesigning both frontend appearances to represent a uniform design.

Additionally, the team agreed to clean up the existing application so that maintenance would be less time consuming and writing tests for the newly implemented parts of the application.

2.5 Results

This resulted in the *media manager ui*, the *media server*, the redesign of the *pod control ui* and a structured and easy to maintain code base of the *controller.* These results are briefly outlined below and described in depth in the following chapters.

For the *media manager ui* we created a browser application using the JavaScript library React with the datastore library redux. It enables the pod administrator to create, update and delete media entries and collections.

The *media server* is the backend component for the *media manager ui*, offering a way to retrieve the media files, entries and collections to the *media manager ui* and the *pod control ui.*

With the *pod control ui* the media inside the pod can be played and the seat position can be adjusted. This requires access to the *media server* over the integrated private network. This application already existed and was merely redesigned and enhanced with features like skipping or pausing media.

The *controller* is responsible for adjusting the seat position and in combination with the prototype also for opening the hood. Later versions may not offer that feature anymore. This part of the system already existed and was refactored to be less time consuming when new features are requested.

3 Introduction

3.1 Supervisor and Advisor

The LifeDrop term project is conducted in collaboration with LifeDrop GmbH and supervised by Thomas Corbat (thomas.corbat@ost.ch) from the OST *Institute für Software*.

3.2 Client

The client for this term project is *LifeDrop GmbH* with its CEO Noe Tüfer (noe.tuefer@life-drop.ch) and CTO Enzo Scossa-Romano (enzo.scossa.romano@gmail.ch).

3.3 Students

The LifeDrop term project is conducted as part of the course *Studienarbeit Informatik* in the fall semester of 2020 by:

- Lukas Schiltknecht (lukas.schiltknecht@ost.ch)
- Robin Elvedi (robin.elvedi@ost.ch)

3.4 Introduction

LifeDrop offers a one of a kind multisensorial pod experience - LifeDrop GmbH

Built by LifeDrop GmbH in Switzerland, the pod resembles a comfortable deck chair with a closable hood. Inside, one can find a beamer as well as a set of powerful speakers. At the start of our term project, the product is in its prototype stage. It is designed to offer a unique visual and auditive experience. It is intended to be used in office spaces, providing a relaxing retreat for employees, or alternatively, as a show case device, enabling exciting product displays.

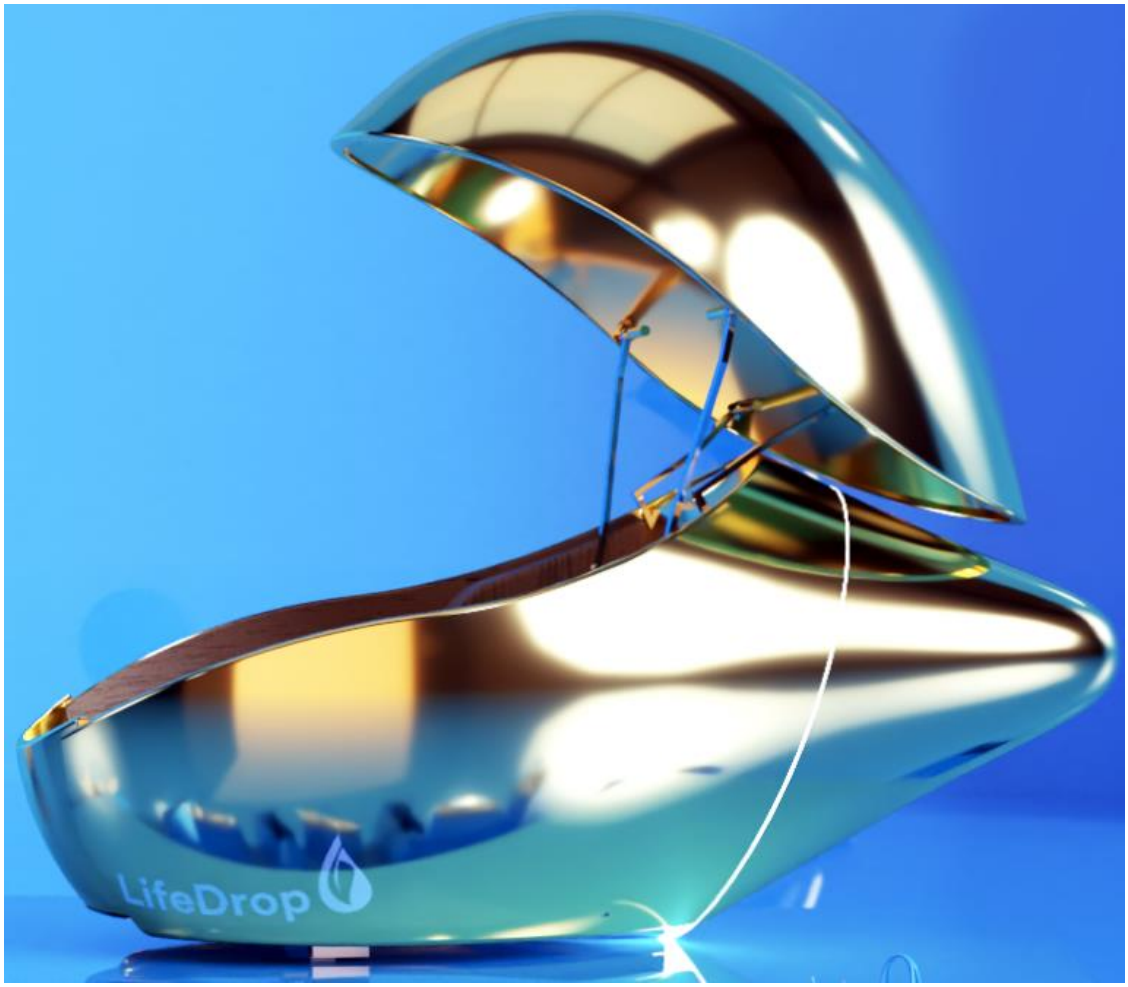


Figure 1: lifedrop chair (source: life-drop.ch)

3.5 Goals

The LifeDrop term project consists of two main goals. The redesign of the react app that steers the pod, and the implementation of a new react app, used to manage the media available to be played inside the pod. Two additional minor goals consist of enabling remote management of the pods media and the steering of a hood used to cover the beamer's lens.

3.5.1 Redesign Pod Control UI

At the start of this term project, the pod is controlled by a react web app running on a tablet. The tablet is used by the person inside the pod to adjust the seating position, close and open the hood as well as play media. Its visual design and user experience is to be enhanced by this term project.

3.5.2 Media Management UI

When the term project began, the media available to be played inside the pod is managed manually by copying files to the raspberry pi installed within the pod. This process is to be replaced by a new react app, offering functionality to upload and manage the media available to the pod.

3.5.3 Remote Media Management

An optional goal of this term project consists of enabling the remote management of media for the pod. This includes making the media manager accessible from the internet.

3.5.4 Hood lens

Another optional goal of this term project consists of enabling the steering of a lens hood for the beamer inside the pod.

3.6 Administrative

3.6.1 Format

The LifeDrop term project follows the documentation guidelines *published by the OST*[1] on Microsoft Teams. As agreed upon between the students and the supervisor, the documentation is written in English.

3.6.2 Dates

Date	*Event*
14.09.2020	Start of the term project.
30.11.2020	On site integration.
07.12.2020	On site demo.
14.12.2020	Abstract hand-in on http://eprints.rj.ost.ch and to supervisor.
18.12.2020	Term project hand-in to supervisor and upload to https://archiv-i.hsr.ch/Overview/All. Deadline is at 5pm.

4 Analysis

In this chapter we describe the existing solution for the LifeDrop Pod.

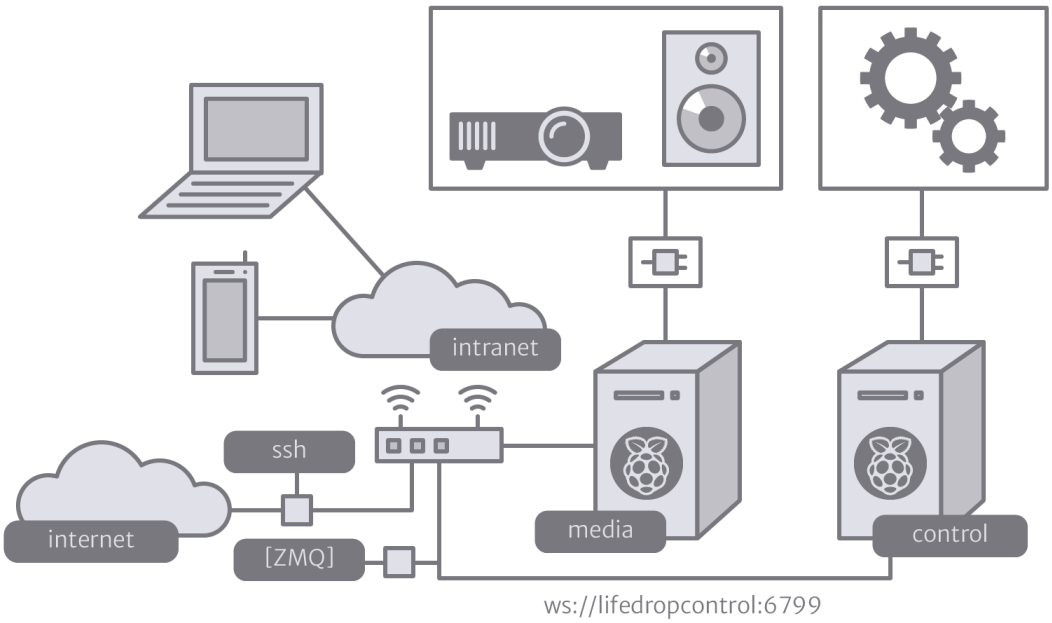


Figure 2: Initial system diagram

4.1 Hardware Infrastructure

Enclosed within the walls of the wooden pod shell are two raspberry pi computers. One of them is responsible for sending and receiving the signals that control the pods functions. This includes sending commands to the mechanical motors powering the pod hood, turning on and off the seat heating as well as turning on and off the beamer. This raspberry pi is called *control pi* hereinafter. The *control pi* also hosts the *media manager*, which is the interface displayed on the tablet for the user inside the pod.

The other one is responsible for storing the media available to the pod. This raspberry pi is called *media pi* hereinafter. The *media pi* hosts the *media service*, which is the service responsible for accepting, handling and returning media events such as a *play video* or *stop playing video.*

The two raspberry pi computers are connected to a portable router which acts as the gateway to the internet. It is itself connected to another local router via LAN connector or WLAN.

4.2 Pod Control UI

The frontend running on the *control pi* and operated from the tablet is referred to as *pod control ui* hereinafter. The *pod control ui* is a react web app. At the time of writing the analysis, it is able to control all mechanical movements of the pod as well as play all media present on the *media pi*. The *pod control ui* has a bare bones design with missing contrast and limited considerations towards the user experience.

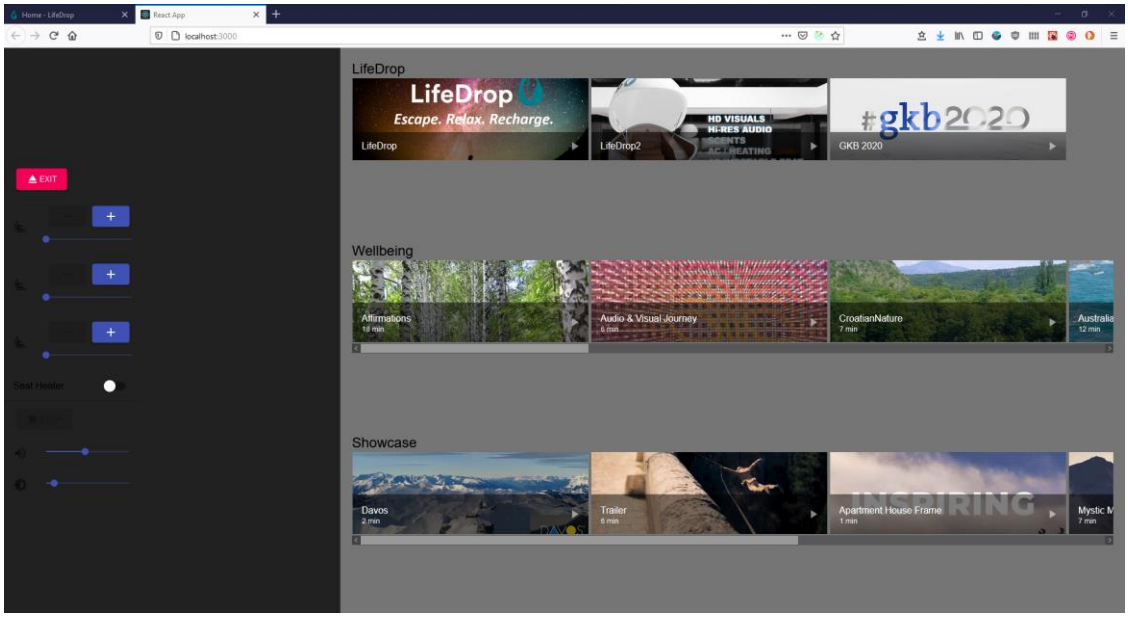


Figure 3: Initial user interface design

4.2.1 Use Cases of Pod Control UI

The use cases of the existing pod control UI are the following:

4.2.1.1 *Play Media*

Clicking the play button below a media topic element, plays the media on the beamer.

4.2.1.2 *Open and Close*

Opening and closing the hood of the pod is done remotely by the pod front-end. (This will not be the case in future pods.)

4.2.1.3 *Control Seat and Seat Heater*

Seat position can be adjusted and seat heater can be turned on and off by the pod front-end.

4.2.1.4 *Control Volume*

The volume of the audio system can be adjusted by the pod front-end.

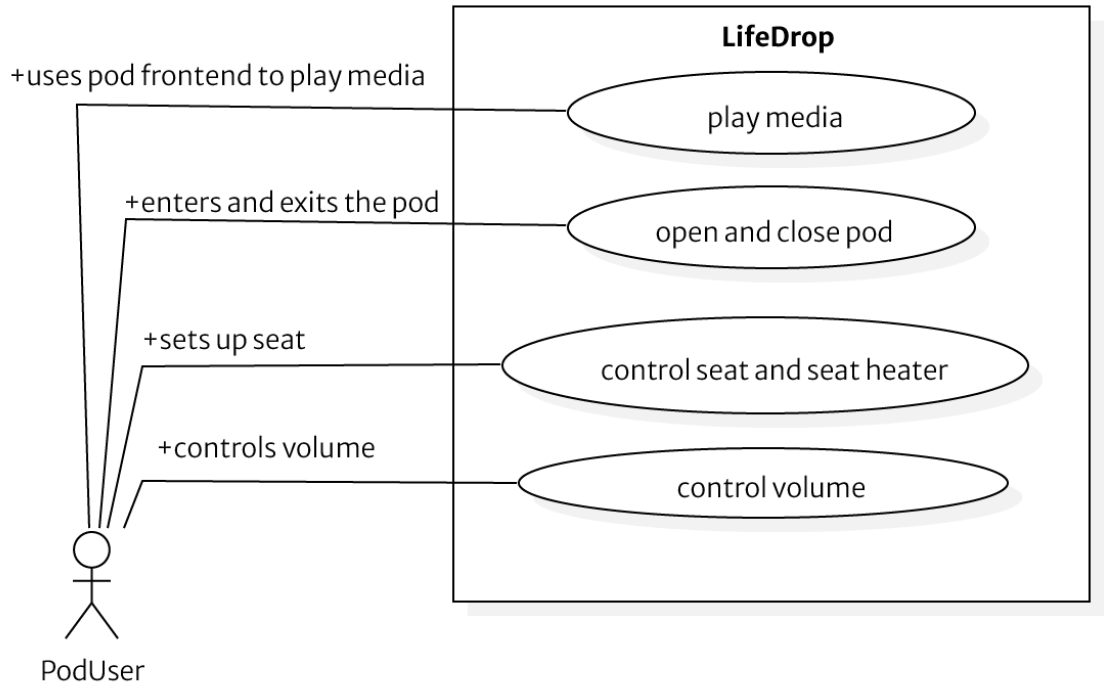


Figure 4: Initial use case diagram

4.2.2 Tests

When the term project started, no tests of any kind exist for the *pod control ui*, therefore choosing appropriate testing environment is an implicit requirement of the project.

4.2.3 Connectivity

The *pod control ui* connects to the *control pi* by using a WebSocket connection. This is initiated in the index.js by instantiating a new WebSocket, connecting it to the host of the *control pi* and passing the instance into the redux data store as a reduxWebsocket.

```
const ws = new WebSocket('ws://lifedropcontrol:6799/');
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || co
mpose;
const store = createStore(rootReducer,
    initialState,
    composeEnhancers(applyMiddleware(reduxWebsocket(ws))),
)
```

At the beginning of the project the host configuration, protocol and host name, was done directly in the file, where the WebSocket was instantiated and forwarded to the redux data store.

4.2.4 Redux Data Store

The redux data store of the *pod control ui* initializes the state of the state machine described in the corresponding chapter as well as the states for seat, door, media and LED. The new state returned by the reducer is instantiated via the static assign method of the object class. All state transitions are caught by the rootReducer.

4.2.5 Material UI

The CSS is applied via the material design library. All the export statements wrap the exported component into a withStyles(styles)(App) function. Also every class unwraps classes prop in the parameters to access the CSS styles. These styles are usually directly defined in the components that use them and rarely cascade through to other components.

4.3 Media Manager UI

The *media pi* hosts a selection of media available to be played inside the pod. At the start of term project, there is no *media manager ui* that would allow media management. Therefore, this workflow was done manually at that point.

4.3.1 Manual Workflow

To add and remove media for the pod, the *media pi* is accessed manually via SSH. The necessary video, audio and image files are then added or removed from the *media pi*'s file system.

Next, a configuration file that acts as the *media catalog* is updated manually to reflect the new state of files available on the *media pi*. This file contains entries for each media file, including its path on the file system as well as some metadata.

Lastly, the react application needs to be rebuild and redeployed onto the *control pi*, where it is hosted. This step is necessary because the *media catalog* configuration file is a part of the application.

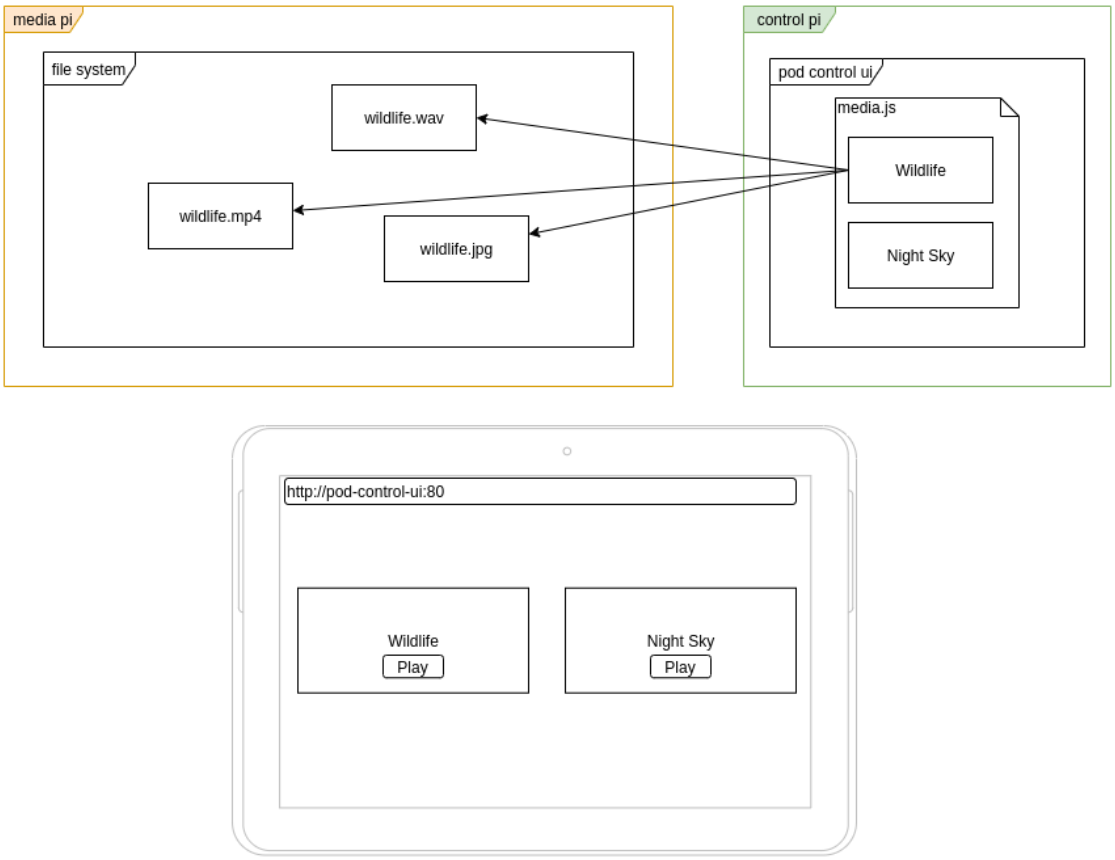


Figure 5: Manual media management workflow diagram

4.3.2 Media Catalog

As mentioned, the *media catalog* consists of a javascript configuration file containing the necessary metadata for the media. Catalog entries are referred to as *cards*. An example of a *card* is shown below. The section titled *LifeDrop* contains three cards. Each card has a fixed set of fields. The image and video file paths point to the location of the media files on the *control pi*, which is the same device that the *pod control ui* is running on, hence the relative file paths.

```
[{
    title: 'LifeDrop',
    cards: [
        {
            name: 'LifeDrop',
            title: 'LifeDrop',
            text: '',
            image: require('./images/lifedrop.png'),
            video: 'lifedrop.mp4',
        },
        {
            name: 'LifeDrop2',
            title: 'LifeDrop2',
            text: '',
            image: require('./images/LifeDrop V3.1 NoVoice.JPG'),
            video: 'LifeDrop V3.1 NoVoice.mov',
        },
        {
            name: 'GKB 2020',
            title: 'GKB 2020',
            text: '',
            image: require('./images/GKB 2020 1min.JPG'),
            video: 'GKB 2020 1min.mp4',
        },
    ]
}]
```

4.4 Remote Media Management

In the beginning of this term project, there is no option to manage a pods media remotely over the internet.

4.5 Lens Hood

At the start of the term project, there is no hardware in place that would allow for the manipulation of a lens hood covering the beamer via the *pod control ui.*

4.6 Media Service

The existing *media service* is implemented in Python and interacts very closely with the underlying operating system on the *media pi* and its installed system software. The systems software deals with outputting sound and video via HDMI and an ALSA sound interface, respectively.

The following is a list of system software that is called by the *media service*:

Nr.	Software	Description	Used for
1.	fbi	Linux frame-buffer image-viewer	Displaying a static background image when no video is played on the pod.
2.	omxplayer	An accelerated command line media player	Used to play video and music on the pod.
3.	mpd	Music Player Daemon (MPD). Server-side application for playing music	Used to play background music on the pod when idle.
4.	mpc	Program for controlling Music Player Daemon (MPD)	Used to control mpd.
5.	amixer	command-line mixer for ALSA sound-card driver	Used to control the ALSA sound-card driver.

4.6.1 OS Interaction

As seen in the table above, the *media service* calls a variety of systems software. This is implemented by spawning shell processes with the corresponding system command and required parameters. The system commands with parameters are coded as strings.

This takes the following form in the code (simplified):

```
async def _call_cmd(self, cmd):
    p = await asyncio.create_subprocess_shell(
        cmd, stdout=asyncio.subprocess.PIPE, stderr=asyncio.subprocess.PIP
E)
```

Various commands are sent to the operating system using this method, `_call_cmd`. This is done as shown below in the case of stopping a video:

```
async def video_stop(self):
    await self._call_cmd('killall omxplayer.bin')
```

Playing videos is handled slightly differently, as can be seen below in a simplified version of the code:

```
async def video_play(self, video):
    args = ['-b', '--no-key', video_p]
    args.extend(['-o','alsa'])
    self.video_proc = await asyncio.create_subprocess_exec(
        'omxplayer', *args, stdout=asyncio.subprocess.PIPE, stderr=asyncio
.subprocess.PIPE )
```

In the code above, the `create_subprocess_exec` method is used directly instead of `_call_cmd`, which in turn would have called `create_subprocess_shell`.

4.6.2 Tasks and Communication

The *media service* receives, processes and returns messages to the *controller*. Those messages are transmitted by the ZeroMQ messaging library, both in synchronous and asynchronous fashion, depending on the scenario.

This is implemented by opening two sockets. One of them is a synchronous ZeroMQ socket, also referred to as a *response* socket. The other one is an asynchronous ZeroMQ socket, also referred to as a *publish* socket.

To facilitate the co-existence of both sockets and the spawned shell processes, a setup using Pythons asyncio package is used. To aid in this cause, a custom wrapper, called *asyncio_tool* exists. It is used to spawn all shells as well as the server listening for synchronous messages in the same asynchronous io-loop, which provided by Pythons asyncio package.

The mentioned *asyncio_tool* helper contains various functions, the main ones used to control the various threads and tasks are *run_tasks* and *run_tasks2*:

```
def run_tasks(loop,tasks,logger):
    for s in (signal.SIGHUP, signal.SIGTERM, signal.SIGINT):
        loop.add_signal_handler(
            s, lambda s=s: asyncio.create_task(shutdown(s, loop,logger)))
    try:
        exception = loop.run_until_complete(exception_handler(tasks, loop,
logger))
    except asyncio.CancelledError as e:
        pass
    else:
        raise exception

def run_tasks2(loop,queue,logger):
    for s in (signal.SIGHUP, signal.SIGTERM, signal.SIGINT):
        loop.add_signal_handler(
            s, lambda s=s: asyncio.create_task(shutdown(s, loop,logger)))
    try:
        exception = loop.run_until_complete(task_handler(queue, loop,logge
r))
    except asyncio.CancelledError as e:
        pass
    else:
        raise exception
```

In the case of the *media service*, a small helper, *add_cb* is used to add tasks, which are asynchronous functions in Python:

```
loop = asyncio.get_event_loop()
def add_cb(task):
    def cb(arg):
        TASK_DONE.put_nowait(arg)
    task.add_done_callback(cb)

media = MediaService(loop, logger, add_cb)

add_cb(loop.create_task(media.setup()))
add_cb(loop.create_task(media.listen()))

if args['raise']:
    add_cb(loop.create_task(raise_err(logger)))

run_tasks2(loop, TASK_DONE, logger)
```

The *controller* itself spawns its asynchronous tasks in a slightly different manner, using an array of tasks and a helper method, `run_controller`:

```
loop= asyncio.get_event_loop()

async def run_controller(loop,logger,tasks):
    for s in (signal.SIGHUP, signal.SIGTERM, signal.SIGINT):
        loop.add_signal_handler(
            s, lambda s=s: asyncio.create_task(shutdown(s, loop,logger)))
    try:
        async with websockets.serve(SM.ui_handler,'0.0.0.0', WS_PORT) as s
erver:
            exception = await exception_handler(tasks, loop,logger)
    except asyncio.CancelledError as e:
        pass
    else:
        raise exception

tasks.extend([
    loop.create_task(MEDIA.listen_for_state_change(SM.media_event)),
    loop.create_task(SM.consume_events()),
    loop.create_task(startup(args['enable'])),
])
loop.run_until_complete(run_controller(loop, logger, tasks))
```

4.7 Controller

The lifedrop pod has a fair amount of motors, sensors and actuators, which all send and receive signals. The central component processing those signals is referred to as the *controller*. The controller runs on a raspberry pi, called the *control pi*. The existing implementation deals with a lot of lower-level signal processing and as such the domain of the controller lies mostly outside the scope of this term project.

Additionally, almost the whole term project will be implemented without physical access to the lifedrop pod due to the impracticality of shipping such a large object around. As a consequence, large parts of the pod are treated as a black box, namely the motors, seat and as mentioned, the controller.

Therefore, our aim is to keep modifications to the controller to a minimum.

The following is a complete list of the services provided by the controller:

Nr.	Service	Description	Out of scope
1	door_service	Controls motors responsible for opening and closing the pod hood.	Yes
2	led	Controls the led lights inside the pod.	Yes
3	media_client	Client for the media_service. Receives messages from the pod control ui and routes them to the media service.	No
4	seat_service	Controls the position of the seat.	Yes
5	state_machine	Controls the transitions between different states of the pod.	No

Services labeled "Out of scope" are not relevant to the term project and are not explained in more detail.

4.8 Media Client

The *media client* acts as an intermediary between the *pod control ui* and the *media service*. User interactions on the *pod control ui* trigger messages that are sent via the WebSockets protocol to the *controller*. There they are eventually delegated to the *media client*, where they are relayed by ZeroMQ to the *media service*.

4.9 State Machine

The *state machine* handles transitions between different states of the pod. Most of these states are not of significant importance to the term project and therefore not explained in detail. The one relevant state is the *READY* state, which signals that the *pod control ui* can now be operated by a person. Unless this is the active state, the *pod control ui* is not active.

A strongly simplified state machine diagram with all states looks as follows:

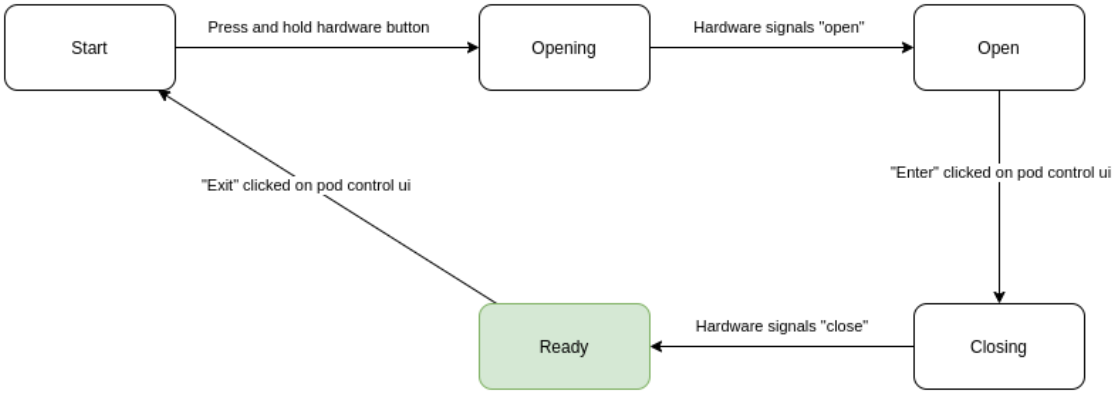


Figure 6: Initial state machine of the pod

The main difference between the states lie in what functions the pod offers. The *READY* state defines the following messages that activate those functions:

Nr.	*Message*	*Description*	*Out of scope*
1.	S_NEW_SET_STEP	Control seat position	Yes
2.	S_NEW_SET	Control seat position	Yes
3.	S_SET_HEATER	Control seat heating	Yes
4.	EXIT	Open the hood	Yes
5.	M_PLAY_VIDEO	Play video	No
6.	M_STOP_VIDEO	Stop video	No
7.	M_VOLUME	Change volume	No
8.	L_BRIGHTNESS	Control leds	Yes

Again, messages marked "Out of scope" are not of significant relevance to the term project.

5 Design

First the changes to the system of the existing life drop will be shown.

5.1 System Architecture

The designed changes do not change much about the physical architecture except for that a database will be installed on the media Raspberry Pi device, so that the media information can be queried dynamically, as can be seen in the system diagram and the system architecture overview.

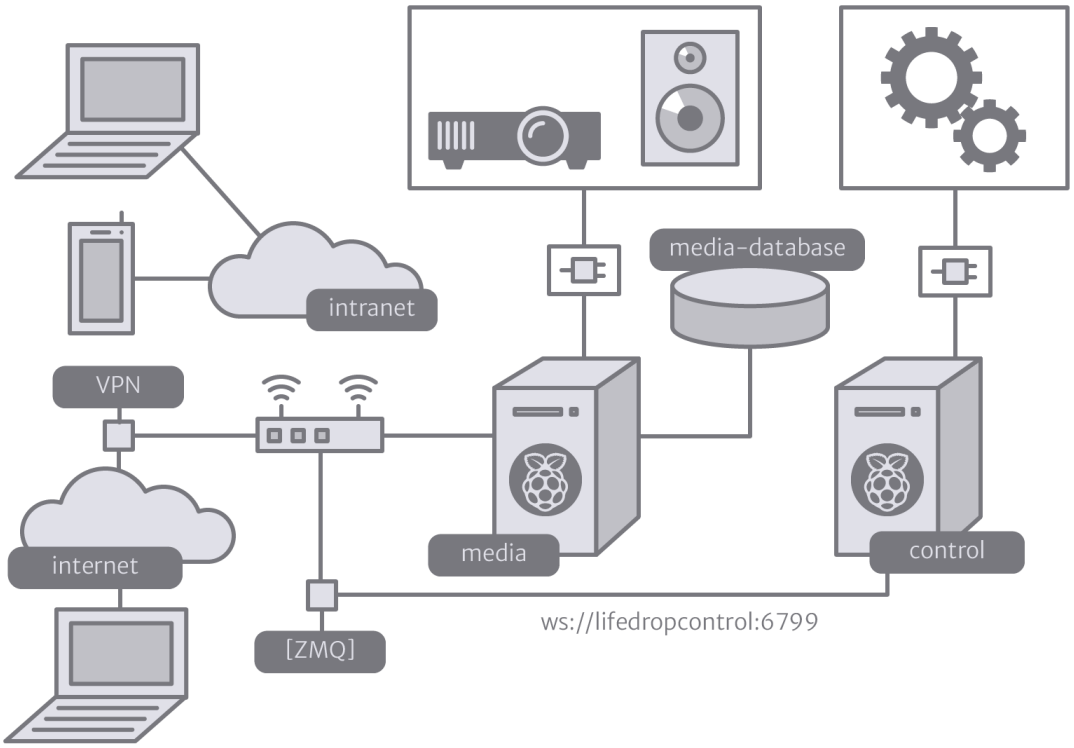


Figure 7: System diagram

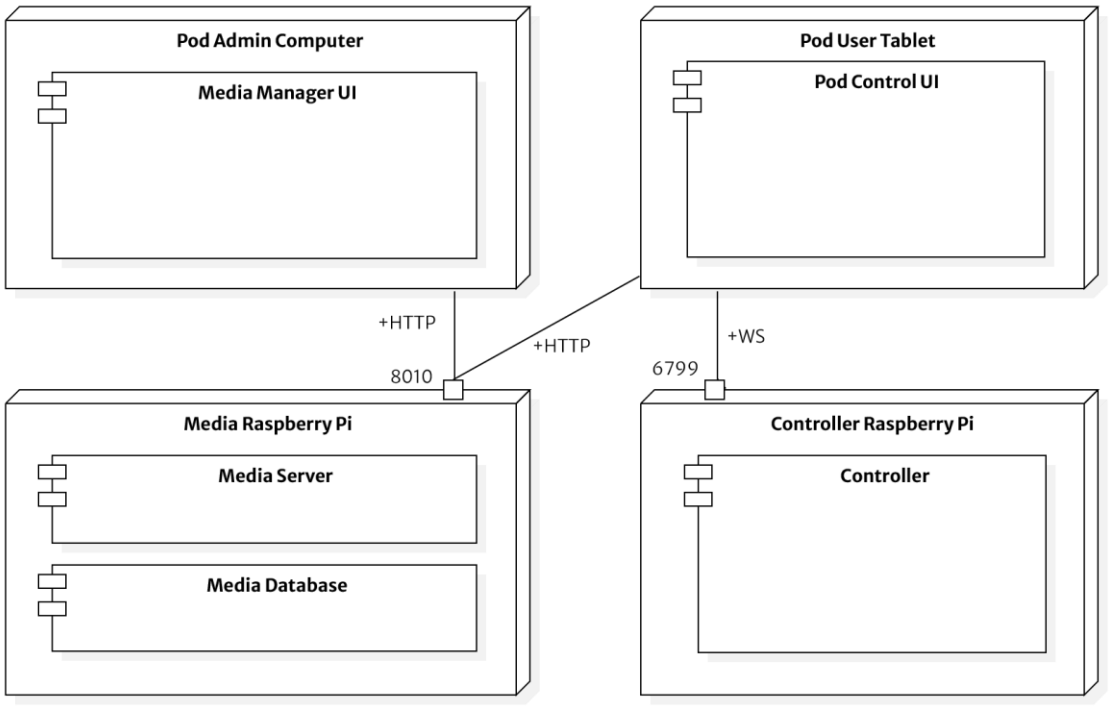


Figure 8: System architecture overview

5.2 Use Cases

5.2.1 CRUD Media Entries

To manage the media shown on the pod frontend is one of the core use cases that the media manager achieves.

5.2.2 CRUD Media Collections

To manage the collections shown on the pod frontend is one of the core use cases as well.

5.2.3 Show Statistics

To show how many times the user interacted with the play button is an additional use case achieved by the system.

The use case diagram displays use cases of the media manager as well as the existing use cases for the pod frontend.

It is planned in the future to enable the possibility to upload applications to the pod and start them on the frontend. However, this was outside the scope of this project and therefore it is displayed in grey to indicate the potential of expansion.

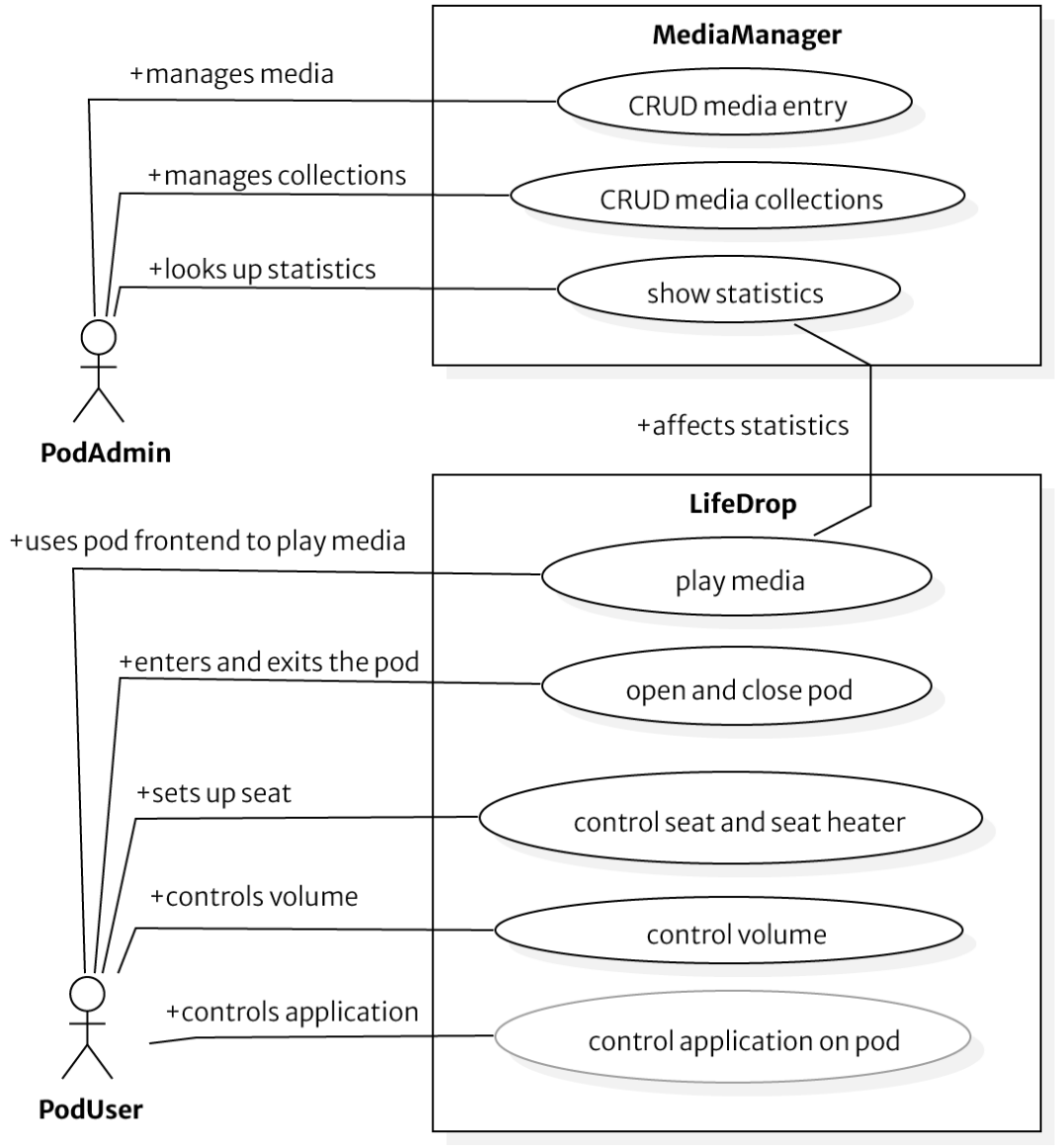


Figure 9: Use case diagram

5.2.4 Media Manager

The use cases for the media manager are very straightforward as can be seen in the figure "use case diagram". It enables the pod administrator to do CRUD on the media files and the media collections, as well as displaying the play counter for each media entry on the pod. The use case "show statistics" depends on the interaction of the pod user with the play button of the media entry displayed on the frontend.

Due to technical limitations of the ZMQ architecture, it was not possible to reliably count the time that a media file was played, so the mere clicking of the button is sufficient to trigger a notification to the media manager backend and increase the play count as such, this is explained in more detail in the chapter about the implementation of the media service.

5.3 Domain Model

The domain model consists of the objects that carry the data of the media entries and the notifications sent by the media service. It is important to note that on the initial approach of saving the media, there was a title, description, thumbnail and a media file all in one entry. This was a practical approach and if the data is manually uploaded, this is a great way to deal with media files. However, the conscious decision was made to split the MediaFile from the MediaEntry, so that the MediaEntry is more loosely coupled to the title, description, thumbnail file entry and actual file entry.

Another decision to change the initial concept was made concerning the MediaCollection. It now only contains an array of MediaEntryIDs, so it is easily possible for a MediaEntry to be contained by several MediaCollections. This makes creating, storing, updating and loading of the media collections really efficient.

The notification class was not yet established in the existing software, so it was logical to connect the play count of a file with the hash of the file itself. This comes with the advantage that the pod frontend can send the file entry and the time stamp and the media manager can match the hash of the file of a MediaEntry, with the hash of the file in the notification. Like this it is easy to identify how many times a file has been played. However, one thing to notice is that if the file is uploaded twice for two different MediaEntries, both media entries will share the same file source and the same play count as well, so even if the thumbnails of the MediaEntries differ, this will not be taken into account for the play count.

5.3.1 MediaCollection

This domain object represents a collection that can be seen in the pod frontend. If a MediaEntry is not contained by a MediaCollection it will not be shown in the frontend.

5.3.2 MediaEntry

This domain object represents a combination of a thumbnail and a file, both being represented by a MediaFile object.

5.3.3 MediaFile

This domain object represents the actual file meta information behind a thumbnail and a file.

5.3.4 Notification

This domain object represents the information that is sent from the pod controller to the media manager, so that the play count can be calculated by the number of notifications matching the file.

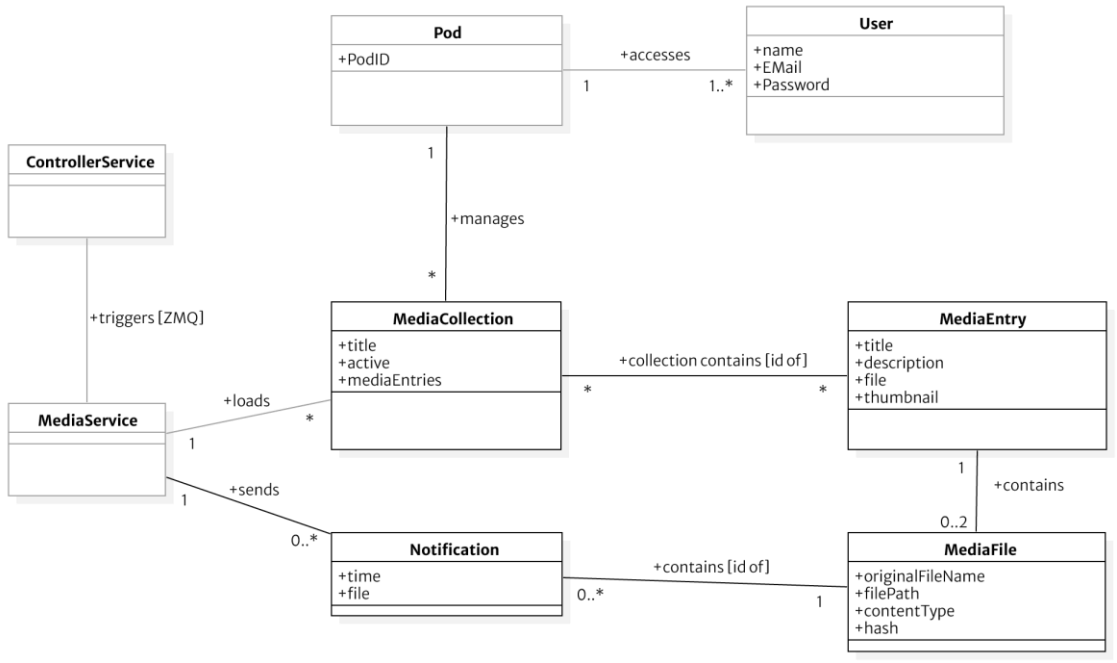


Figure 10: Domain model

5.4 Pod Control UI

This component is redesigned to match the look of the *media manager ui.* There were no design decisions made specifically for this component worth mentioning.

5.5 Design Media Manager

This chapter deals with the design decisions for the media manager frontend.

5.5.1 Design Media Manager Frontend

There are 3 different types of data that the media manager frontend has to store. There is file data, collection data and the state of the navigation, so the components can be loaded accordingly. Therefore the media manager frontend includes three data stores for redux. The stores include a *fileloader* datastore, a *collectionloader* datastore and a *navigation* datastore with following objects:

```
const initialFileLoaderState = {
  loadedFilesArray: undefined,
  loading: undefined,
  statistics: undefined,
  fileID: null,
}

const initialCollectionsState = {
  loadedCollections: undefined,
  selectedCollection: undefined,
}

const initialNavigationState = {
  showNavigation: undefined,
  showError: undefined,
}
```

As can be seen in the initial states, the state objects are generally initialized as undefined values and the actual values are loaded when the component is mounted via the UseEffect event handler. This is a conscious design decision, so the loading process will always be triggered on loading a new component. This comes at the cost of having to reload the statistics component to see the updated view of the play count.

5.5.2 Backend Media Manager

5.5.3 NFRs

The non-functional requirements of the pod are of nonfatal nature mostly. The taxonomy used for this project is FURPS. Not all of the non functional requirements are by definition SMART, but they are designed to match as many of the SMART criteria as possible.

5.5.3.1 *Functionality*

1. As a user of the media manager, I can manage (CRUD) video files, audio files and image files as thumbnails for the media pi, so they can be played from within the pod.
2. As a user of the media manager, I can manage (CRUD) collections of Media, so that the media files can be organized.
3. As a user of the media manager, I can view the play count of each media entry.

5.5.3.2 *Usability*

1. As a user of the media manager, I can perform the media CRUD operations for all 2 types of media from the same interface.
2. As a user of the media manager, I can see a status indicator above each media element, confirming whether or not it is actually on the media pi.

5.5.3.3 *Reliability*

1. As a user of the media manager, I get a warning if the media pi is about to run out of disk space in advance, and am denied upload of new media.
2. As a user of the media manager, I get a warning if media is deleted from the disk of the pod.
3. As a user of the media manager, I can only upload images to the thumbnail and audio or video files to the file entry of a media entry.

5.5.3.4 *Performance*

1. As a user of the media manager, I can upload one file at a time via local ethernet connection at the speed of 100MB/s and get a response within 100ms when the upload is done.
2. As a user of the media manager, I can upload one file at a time via VPN connection at the speed of approximately 5MB/s and get a response within 300ms when the upload is done.

5.5.3.5 *Supportability*

1. As an admin of the media manager, I want to have the option to easily migrate the media manager to a different device.
2. As an admin of the media manager, I want a straightforward update process.

5.6 Controller

No design changes are made to the controller. One goal from the analysis was to modify the controller as little as possible, due to reasons described in the analysis chapter.

5.7 Media Service

Following the analysis, the decision was made to modify various aspects of the *media service* design.

The most important one being the interaction with `omxplayer`. After some research, a suitable library emerged, `omxplayer-wrapper`, which controls the `omxplayer` over the operating system's *dbus* mechanism. This allows for a cleaner implementation without the need to manually send system commands and spawn shells from within the python runtime.

As further research did not yield suitable library replacements for interacting with the other system tools, namely `mpc`, `fbi` and `amixer` (see analysis), the decision was made to use `sh`. This is a library which abstracts away the sending of system commands to the operating system in a uniform way.

Those decisions are intended to eliminate manual spawning of shells and processes and the sending of system commands to the operating system. In the case of the `sh` library, this is ultimately still the case under the hood, but abstracted away from the developers working with the code.

5.7.1 Messages

The current types of messages (see analysis) are not sufficient to handle additional requirements for the term project. Specifically, skipping back and forth in a video as well as pausing a video can not be implemented without extending the messages. Therefore, two new messages are introduced (9. and 10.):

Nr.	Message	Description	New
1.	S_NEW_SET_STEP	Control seat position	No
2.	S_NEW_SET	Control seat position	No
3.	S_SET_HEATER	Control seat heating	No
4.	EXIT	Open the hood	No
5.	M_PLAY_VIDEO	Play video	No
6.	M_STOP_VIDEO	Stop video	No
7.	M_VOLUME	Change volume	No
8.	L_BRIGHTNESS	Control leds	No
9.	M_PAUSE_VIDEO	Pause video	Yes
10.	M_SKIP	Skip to timestamp	Yes

The addition of those new messages allows for the implementation of the according features. Skipping forth and back while a video is playing as well as pausing and resuming it at the current position.

5.8 Remote Media Management

During the integration phase of the term project a solution for remote media management was put forward by Enzo Scossa Romano.

The approach is based on a specific router by the company *gl-inet* and uses its proprietary software to facilitate a remote VPN connection between the router installed inside the pod and a permanent router acting as a server, which will be installed inside the office space of the lifedrop company.

Using the bundled VPN software, the gl-inet router inside the office can communicate with the second 4G LTE capable gl-inet router inside the pod. Once this communication channel has been established, access to the media manager can be facilitated.

Designing the remote access in this way allows for further addition of new pods using the same gl-inet routers in a simple manner.

6 Implementation

6.1 Overview

In this chapter both the architecture of the media manager as well as the improvement of the existing software for operating the pod will be discussed. This will be split up into two sections. Each of them will be structured into a description of the technology stack, architectural overview, list of features and a description of how the components were tested. Then the integration workflow and the deployment are described in a brief and concise way, leading to the last topic, the metrics of the project realization.

6.2 Media Manager

This part of software was inexistent before and had to be developed over the course of this module from scratch. According to the first meeting with the client, this was the main goal of the project.

6.2.1 Technology Stack

Management of the project was achieved with GitLab features. Ticketing, time tracking and sprint management was largely achieved with issues and annotations (/spend etc.).

For developing the frontend and backend of the media manager the IDEs of choice were Visual Studio Code and WebStorm.

To keep the number of different technologies used in the system at bay, it was logical to implement frontend as well as backend purely with ECMAScript 6 in a react frontend and an express backend running on node.js. For testing purposes cypress was used in the frontend and mochajs in the backend. As a common agreement we did not use any additional software to "eslinter" to ensure code quality but to trust in thorough code reviews.

For Continuous Integration and Continuous Delivery a docker container was set up and a pipeline was configured. This was one of the major issues during the development process, since the testing procedures required quite a number of external components to work together seamlessly.

6.2.2 Architectural Overview

The media manager consists of a react-application as the frontend and a node application as the backend.

The frontend app is divided into two main sections. One is the "sideControls", which contains the NavigationSidebar and the CollectionSidebar components. On the right side there is the "mediaManager" main part, where the media tiles, the update form and the media statistics are displayed. As an overlay, while uploading big files to the server, a loader is displayed.

The backend app consists of an express server (node.js), a web socket for the play notifications from the *pod control ui*, a router, a collectionController, a mediaController and a mediaStore service, that connects to mongodb using mongoose.

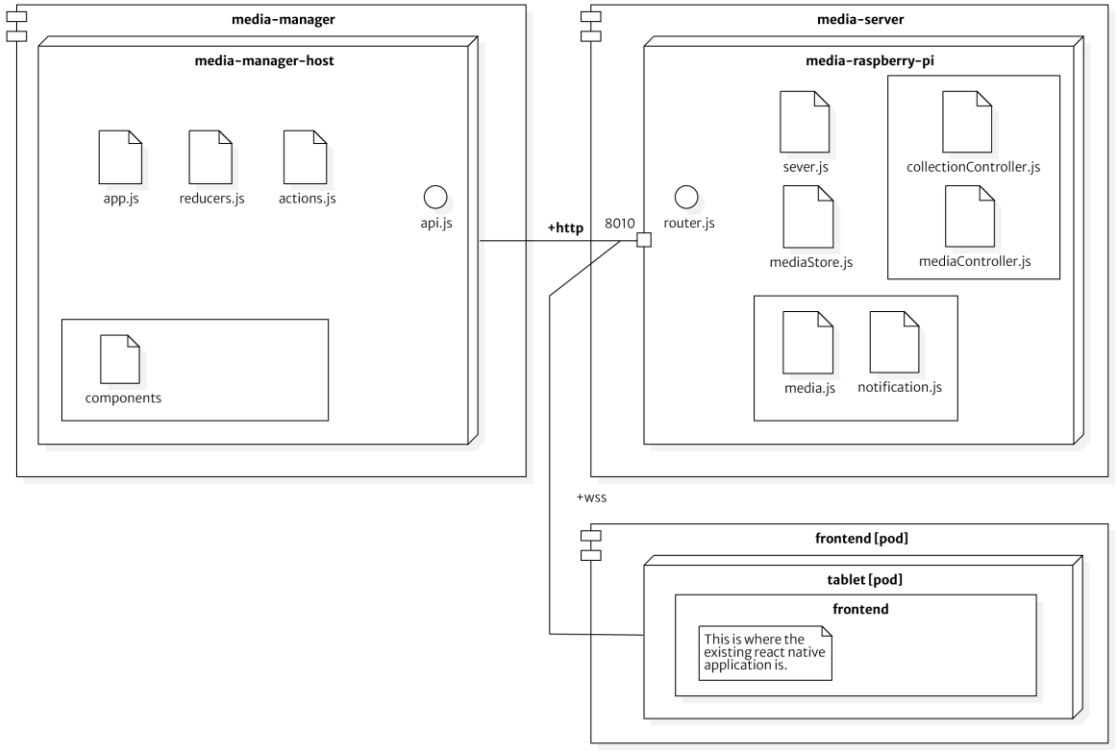


Figure 11: Component diagram

6.2.3 List of Features

The features of the media manager are as follows:

- CRUD for media assets like audio and video files (figure: create media interaction)
- CRUD for media collections shown in the pod (same sequence as for media assets)
- Displaying player stats (figure: notification interaction)

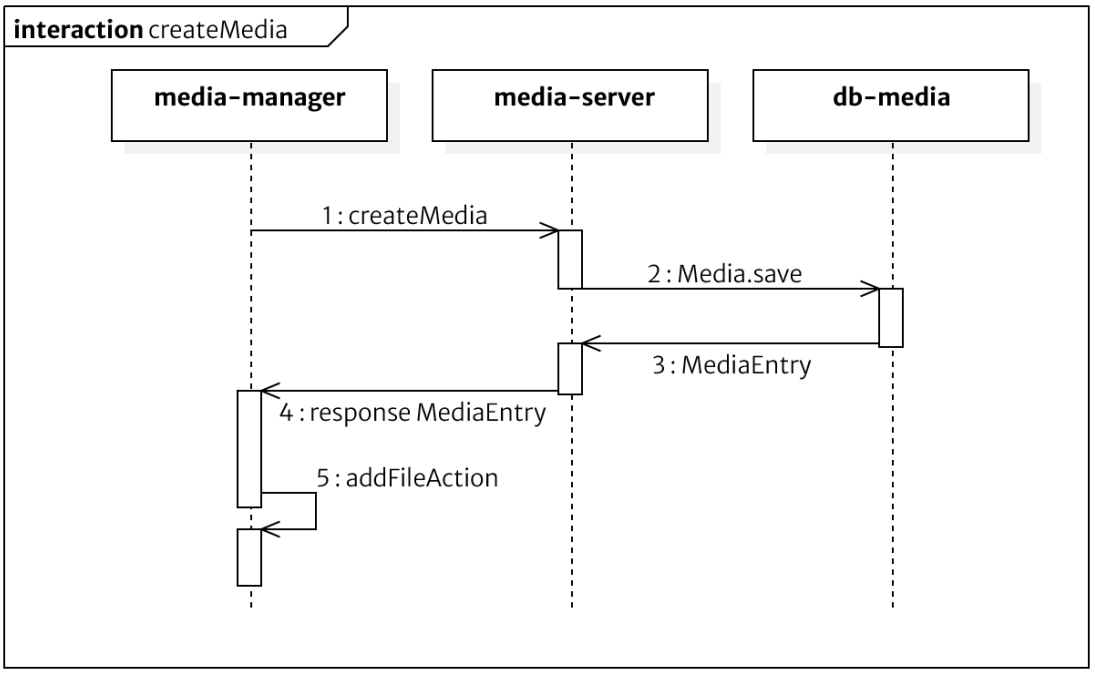


Figure 12: Create media interaction

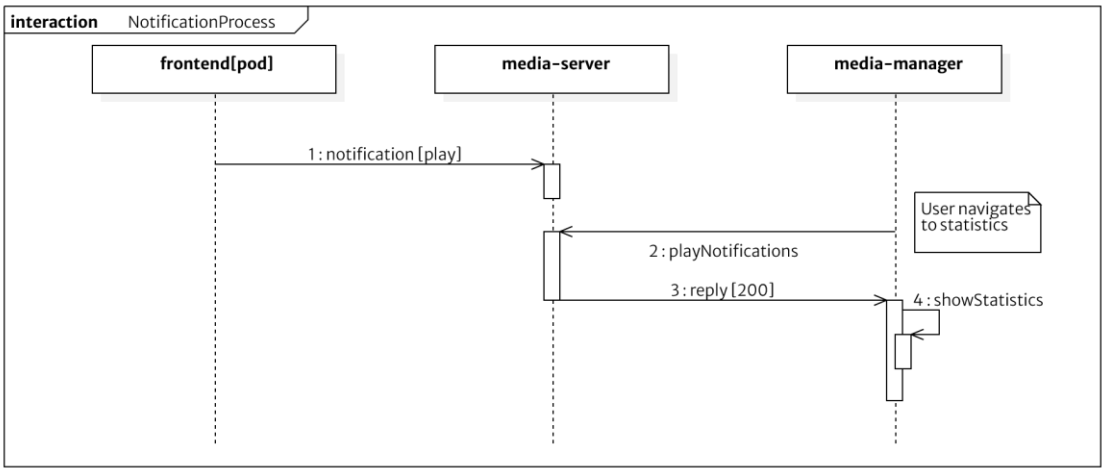


Figure 13: Notification interaction

6.3 Pod Control UI

6.3.1 Overview

The *pod control ui* described in this section refers to the existing react ui for controlling the pod via tablet. It sends the messages to the *

6.3.2 Goals

The *pod control ui* was touched as little as possible to achieve maximum recognition by the original author. However, there were some changes to be made to the original software due to the implementation of the new messages desired by the pod operator namely the M_PAUSE_VIDEO, the M_SKIP and the S_NOTIFY. To automatically load the collections from the *media pi*, it was also necessary to update the MediaTopic component and therefor implement basic testing for the components rendering capability. Last but not least, the *pod control ui* had to be redesigned so the media manager frontend and the *pod control ui* look alike.

6.3.3 Technology

All components of the *pod control ui* were up and running as a react js application. All the code is written in JSX and the data is stored using redux. Therefore it was logical to continue using the existing code and the existing way to communicate between the python backend and the *pod control ui.*

6.3.3.1 *Actions*

To support the new messages, new actions had to be defined to pause the media, to skip to a specific point on the timeline, to request and to receive collections. The new actions are named in camel case so that it is clear that they include the dispatch call and are to be treated as functions not as action return values that have to be dispatched in the component.

To achieve connectivity between the *pod control ui* and the *media server* we had to fetch data using HTTP and the hostname.

```
function requestCollections() {
    return {
        type: REQUEST_COLLECTIONS
    }
}
function receiveCollections(json) {
    return {
        type: RECEIVE_COLLECTIONS,
        collections: json
    }
}
export function fetchCollections() {
    return dispatch => {
        dispatch(requestCollections())
        return fetch(`${hypertextProtocol}${mediaserver}/collections`)
          .then(response => response.json())
          .then(json => dispatch(receiveCollections(json)))
    }
}
```

As seen above the endpoint '/collections' is requested and the response is dispatched on into a function called receiveCollections(json), that passes the collections JSON into the redux data store. This code implements one of the key features of the project, namely requesting the collection data from the new *media server* and making the data accessible in the *pod control ui.*

6.3.3.1.1 *Challenges*

The NewMediaTopic component displays the content of the media entries contained in the collections above and has a slightly tweaked play button. This would lead to a problem at the *integration day*, because the media did not seem to change when a different play button was clicked. Only after pressing the stop button the other buttons would react as intended. This was fixed using a state variable "isPlaying", that is always saving the media, that has been played last. Like this every time the play button of another media entry is used, the stop message can be triggered automatically.

6.3.4 Frontend redesign

The frontend of the pod was merely redesigned to look a little more like the media manager. The media-manager uses only one CSS file to style the whole application, to keep the DRY principle, the *pod control ui* redesign was done with the same tools it was originally designed with, namely *material ui*. The react *material ui* specifications were originally placed at several locations in the js-code and the inheritance of classes and tags were done accordingly.

To keep the structure intact and to alter as little as possible of the existing code was important so that the original author would recognize the spots where the adjustments were made to preserve maintainability.

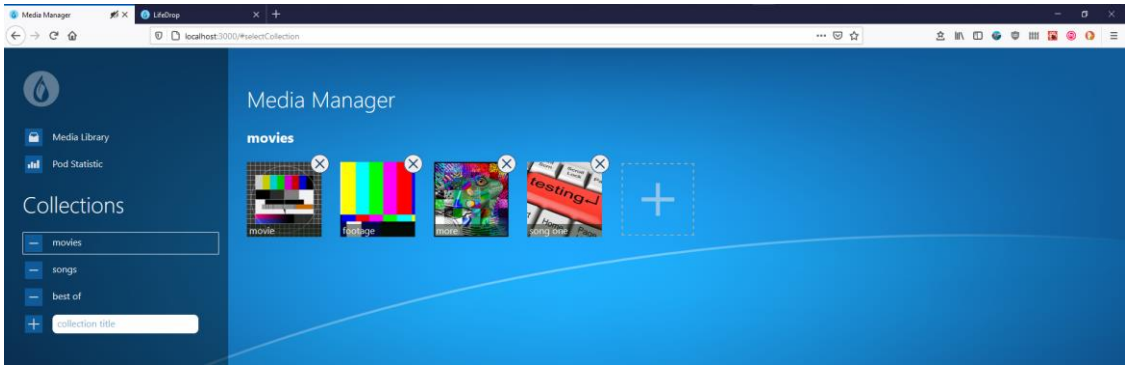


Figure 14: Screenshot of the media manager frontend

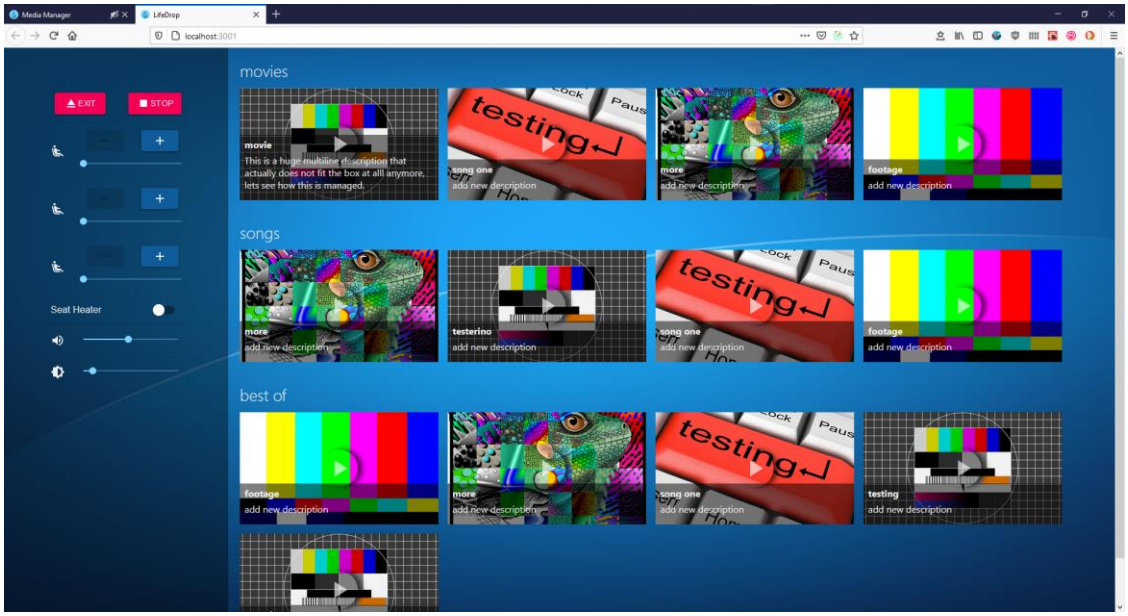


Figure 15: Screenshot of the pod control ui

6.4 Media Service

6.4.1 Overview

The *media service* described in this section refers to the component running on the *media pi*. It is responsible for playing media, which it can do because it is connected to a beamer and a set of speakers.

6.4.2 Goals

The main goal of the *media service* is to play media (mainly videos) on the raspberry pi its running on. The video is outputted via HDMI to a beamer, which projects it onto the screen inside the lifedrop pod.

The goal of the re-implementation of the *media service* is to turn it into a better maintainable component. By doing so, the *media service* can act as a reference for future refactorings of other components.

6.4.3 Technology

Multiple components for handling interactions with the pod's hardware were already in place. They are all written in Python. Based on that, we decided to re-implement the *media-service* in a clean and maintainable way using Python as well.

The existing solution features ZMQ based communication between the different components. As such, we decided to continue using ZMQ for this purpose.

6.4.3.1 *Player*

Playing media on a raspberry pi comes with its challenges. One of which is the choice of an adequate video player.

People familiar with the raspberry pi ecosystem might object that there are in fact a plethora of available media players out there ready to choose from. This is only true however, when it comes to players that are exposed by a web interface, such as plex, and therefore usually use the browser's default player or a custom javascript player.

The lifedrop pod necessitates a player that outputs its content *directly* over HDMI to be picked up by the beamer, which severely limits the choice.

The existing solution uses the *omxplayer*, a fairly straightforward player with a CLI interface. As such, controlling this player from software is done by spawning shells and emitting CLI commands, which has multiple drawbacks. See *Analysis* for more.

After some consideration, we opted to keep the existing *omxplayer*, but only because we found a wrapper-library for it after some searching. The aptly named *omxplayer-wrapper* library for python exposes an API that talks to the *omxplayer* via the system's *dbus*.

6.4.4 Context

The *media service* is a single component in a collection of many components, referred to as the *controller*, whose purpose as a whole is to handle the interaction between the *pod control ui* and the pod hardware.

This includes playing media, controlling the seating position, opening and closing the hood and adjusting the seat heating.

The *media service* is best understood by looking at it in the context of all the other components.

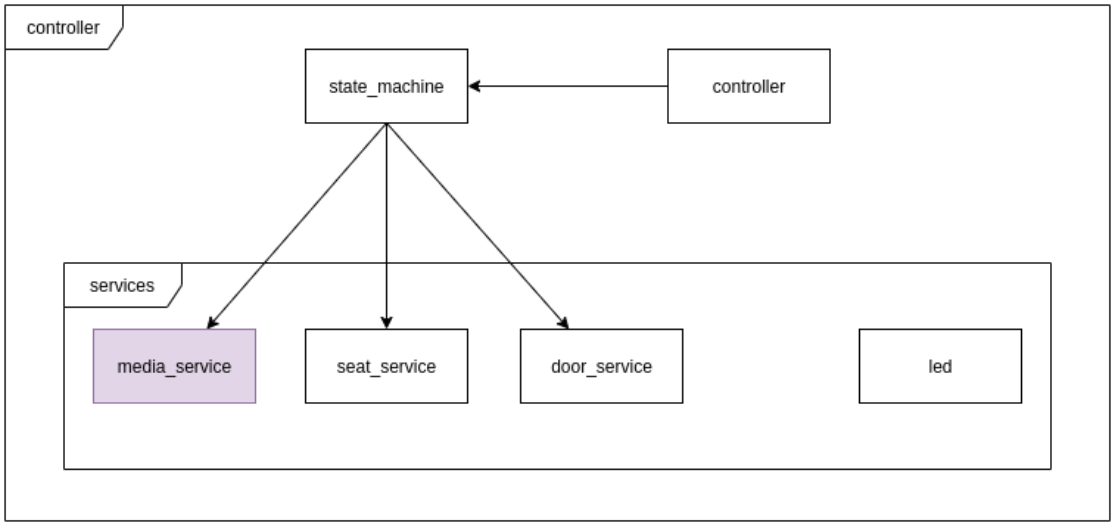


Figure 16: Controller

The above image shows the context of the *media service*. A more detailed explanation is given in the *Analysis* chapter.

6.4.5 Components

The *media service* itself is divided into two main components. The *Player* and the actual *service*, referred to as *media-service.*

6.4.5.1 *Player*

The *Player* exposes the necessary functions for interacting with the *omxplayer* to the *media service.*

```
def play(self, file_name):
def pause(self):
def seek(self, time):
def quit(self):
def set_volume(self, volume):
def current_volume(self):
def current_timestamp(self):
def current_video_name(self):
def video_length(self):
```

This interface capsules the player and presents a limited set of operations available to the *media service*. As such, it is not too difficult to add new capabilities to the *Player* if necessary, since the only place requiring change in such a case is the *Player*.

The *Player* itself uses the *omxplayer-wrapper* library to talk to the *omxplayer* installed on the host OS.

As an example, this is a simplified but representative version of the *seek* method, used to jump to a particular timestamp in the video:

```
    def seek(self, time):
        if self.player is None or not self.player.can_seek():
            print("Video is not in a seekable state")
            return

        if time < 0:
            print("Seek time cannot be negative")
            return

        if time >= (self.video_length()):
            print("Cant seek past end of video")
            return

        position = int(self.player.position())
        seek = time - position

        self.player.seek(seek)
```

The instance variable *self.player* represents the *omxplayer-wrapper* player. This is a significant improvement over the existing implementation both in regards to code readability and maintainability.

6.4.5.1.1 Challenges

Although *omxplayer-wrapper* exposes a reasonable API, one particular aspect of it makes for a challenge.

Because the player is spawned in a separate thread, managed by *omxplayer-wrapper* and *not* exposed to the client, one is limited to interacting with it *solely* via API.

This came in as a stumbling stone when implementing the video progress feature. This feature includes displaying a video progress bar in the *pod controller ui*. This required keeping track of the video position (as in time position) and updating the *pod controller ui* periodically (every second) about the new video position.

This requires periodically querying the player for its current position from *a separate* asynchronous python routine in order to not block other requests from the *pod controller ui*.

So far, nothing unusual or challenging. The issue arises when trying to execute a callback from within the player thread, such as publishing a message when the video is done playing. In order to do so, one needs access to the ZMQ, which also runs as part of the main thread in its own asynchronous routine. A callback given to the player can therefore not access anything that is part of the main thread async loop, because it is not part of that loop.

The video progress feature plays insofar a role in this as that it is the reason an asynchronous loop is needed in the first place.

One solution to this would be to fork the *omxplayer-wrapper* library and expose thread information. We opted against this.

Another solution is to let the asynchronous progress routine handle the publishing of "video finished" messages, as it is part of the main thread async loop.

```
async def __progress(self):
    try:
        while True:
            self.timestamp = int(self.player.position())
            reply = # gather video progress and some state
            self.pub_socket.send_json(reply)
            await asyncio.sleep(1)

    except asyncio.CancelledError:
        pass

    finally:
        self.pub_socket.send_json(PUBLISH_EVENT_VIDEO_FINISHED)
```

The above code shows approximately how this is implemented. While the video is running, an update is sent every second about the video's position.

As soon as the video stops playing, a callback is executed, which as part of its cleanup also cancels the `__progress(self)` routine. This is done by means of calling `self.progress_task.cancel()`.

The `finally` clause inside `__progress(self)` can now, as a final stage of the cleanup process, publish the message that the video has stopped.

This solution is by no means perfect. It is fairly pragmatic though, and compared to forking the whole *omxplayer-wrapper* library significantly less expensive.

6.4.5.2 *Service*

The *media service* does not so much export an API as it is rather a side effect defined service. The rules it must follow are dictated by the messages it receives from the *controller* and by extension the *state_machine*.

The messages are listed below:

```
self.api = {
    "video_play": self.__play_video,
    "video_stop": self.__stop_video,
    "skip_video": self.__skip_video,
    "set_volume": self.__set_volume,
    "music_play": self.__play_music,
    "music_stop": self.__stop_music,
    "music_pause": self.__pause_music,
}
```

The code above is a complete implementation of the "interface" that the *media-service* has to implement. Each message carries with it the appropriate payload to allow for its execution.

The *media service* can the be designed as a simple server with both synchronous and asynchronous communication patterns at the same time:

```
async def listen(self):
    while True:
        print("Listening...")
        message = await self.reply_socket.recv_json()
        self.dispatch(message)
```

Handling of a `video_play` message is then handled as follows:

```
def __play_video(self, msg):
    playing = self.player.play(msg['video_play'])
    if playing:
        reply = # prepare the reply
        self.reply_socket.send_json(reply)
    else:
        self.reply_socket.send_json(REPLY_PLAY_NONE)
    return
```

It should be noted that this will lead an immediate response, which is synchronous, to the *controller*, indicating that the `video_play` request has been processed.

The actual video is then played by the *Player*. Also, the `exitEvent` is the last piece of the puzzle concerning the asynchronous progress tracking and opaque player thread. If called, it executes `self.progress_task.cancel()`.

```
def play(self, file_name):
    self.player = OMXPlayer(video_file)
    self.player.exitEvent = self.__video_stop_cb
    self.progress_task = self.loop.create_task(self.__progress())
    return True
```

6.4.5.2.1 *Startup*

The *media service* can be started as a normal python script, which is in line with the existing implementation of the other services.

```
if __name__ == "__main__":
    media_service = MediaService(REP_ADDR, PUB_ADDR)
    loop = asyncio.get_event_loop()
    loop.run_until_complete(media_service.listen())
```

This is a significantly easier to read and understand procedure than the existing solution, see *Analysis* for more.

6.4.6 Hot Plugging

As is the case with many ARM based systems, device handling is implemented in a non uniform way, meaning that every system is free to do how it pleases. In the case of our raspberry pi, the os, raspbian, implements device handling in a way that, by default, does not support hot plugging. This means, connecting the HDMI cable to the raspberry after it has booted won't work.

To overcome this, one can set up a "forced" HDMI output. This can be achieved by editing the `/boot/config.txt`:

```
#### Custom settings BEGIN
# See https://www.raspberrypi.org/documentation/configuration/config-txt/v
ideo.md

# uncomment if hdmi display is not detected and composite is being output
hdmi_force_hotplug=1

# uncomment to force a specific HDMI mode (this will force VGA)
hdmi_group=2
hdmi_mode=82

#### Custom settings END
```

The drawback is that the ability to dynamically detect the resolution and aspect ration of the targeted monitor or beamer is lost.

This requires manual configuration of the output format as shown above.

6.5 Media Manager UI

6.5.1 Overview

The *media manager ui* described in this section refers to the react application that enables media management of the pod.

6.5.2 Technology

Due to the fact that the *pod control ui* was already implemented with react and the knowledge of this technology already exists in the company, the *media manager ui* was also implemented as a react app. In contrast to the *pod control ui*, the *media manager ui* uses HTTP requests to handle the functionality. The data of the application is stored in a redux store, just as it is in the *pod control ui.*

The application is written in JavaScript, so there is no type safety. This makes it necessary to use techniques like defensive programming, for example when implementing the callback for the onChange event of the update file field.

```
const onUpdateFileHandler = async (event) => {
  event.preventDefault();
  if (!selectedUpdateMediaEntry) {
    return;
  }
  /* ... */
}
```

6.5.3 Components

The components are separated according to their function, their names are chosen accordingly to improve the maintainability. In general the creation of components of the *media manager ui* follows the principle “as few as possible and as many as necessary”. This ensures that the amount of datasets in the redux data store is minimized and the data managed via the use of useState hooks is maximized.

6.5.3.1 *CollectionSidebar*

This component enables collection management. It handles collection creation and deletion. Adding and removing media is by design not part of its responsibility.

6.5.3.2 *MediaCollection*

This component displays the media entries of the currently selected media collection. When no collection is selected, all media is displayed. It also contains an UploadForm component, which uploads media to the pod. If a collection is selected, the media will be added directly to the collection as well.

6.5.3.3 *MediaContainer*

This component displays title and thumbnail of the media entry. By default the thumbnail is shown as an add image symbol, so that it is clear by clicking it, the user will be able to update the entry.

6.5.3.4 *NavigationSidebar*

This component got added later in the process due to the request of adding statistics to the *media manager ui.* It enables the pod manager to switch between statistics window and the media library.

6.5.3.5 *StatisticsWindow*

This component displays the count of interactions with the play button of a media entry on the *pod control ui.* These counts are derived from the amount of notifications sent to the *media server* containing time and media entry. The numbers are connected with the file name, containing the md5 hash of the file. The upside of this procedure is, that if the same file is contained by different media entries, the count remains the same. The downside is that it may occur that two entries share the same number.

6.5.3.6 *UploadForm*

This component enables the initial file upload. It is displayed as a big plus, when clicking on the plus, the local file system opens and a file can be selected. No matter what valid content type is selected, audio, video or image, the media entry is created successfully. If a video or audio file is selected, then the file is sent as file entry and otherwise as image entry.

To check if the remaining disc space is sufficient to store the file, the api checks the '/discspace' endpoint to compare beforehand, whether the file can be uploaded. This way it is virtually highly unlikely to ever reach the error section of the apis upload function.

6.5.3.7 *UpdateForm*

This component enables the pod administrator to complete the information for the media entry. It implements file size checks and error messages according to the specific requirements of the fields.

6.5.3.8 *api*

This is the component that communicates to the *media server.* Every call to an HTTP endpoint gets sent from this location. The error sections serve mainly to debug the code. These sections should not ever be executed, therefore testing these sections was intentionally omitted. The data exchange between the *media manager ui* and the backend is achieved via asynchronously initiated and handled HTTP requests. If no appropriate data is returned by the response, the default value for collections is an empty array and for a single value it is "undefined". This way it is easier to use defensive programming on the single values and collections will always have static methods like "map".

6.5.3.9 *reducers*

This component contains the three reducers of the redux data store. The *fileloader*, the *collectionloader* and the navigation. The *fileloader* is responsible for everything that considers media entries, files and thumbnails. The *collectionloader* manages the state of the collections and the navigation is managing the global error state and the navigation state.

6.5.3.10 *actions*

This component is responsible for dispatching the state changes to trigger a redux state update procedure. It uses functions of the api to connect to the *media server.*

6.6 Media Server

6.6.1 Overview

The *media server* described in this section refers to the component responsible for handling requests from the *media manager ui*.

6.6.2 Goals

The main goal of the *media server* is to manage the persistence of media files on the *media pi*. This includes handling the CRUD operations for media files and maintaining a catalog of the respective files with their metadata.

6.6.3 Technology

The decision was made to use Node.js for the implementation. The reasoning behind it is that the *media manager ui* is a react web application written in JavaScript. Given the circumstance that the *media server* primarily serves the *media manager ui*, using the same language and ecosystem (Node.js, npm) is a reasonable choice.

6.6.4 Components

The *media server* follows a well known design taught to the students of OST in the courses WED1 and WED2.

6.6.4.1 *Router*

All capabilities of the *media server* are exposed by a straightforward RESTful HTTP API. The router is a component responsible for declaratively specifying the HTTP routes and linking them to the corresponding controllers, which handle the request.

An example for such a declarative route definition:

```
router.get('/collections', collectionController.getCollections.bind(collec
tionController));
```

The following table shows all the API endpoints exposed by the *media server*.

Method	*Path*	*Parameters*	*Functionality*	*Controller*
`GET`	`/collections`	–	Returns a list of all collections	`CollectionController`
`GET`	`/media`	–	Returns a list of all media entries	`MediaController`
`GET`	`/discspace`	–	Returns the remaining available disk space	`MediaController`
`GET`	`/collection/:id`	`id::collection_id`	Returns a collection with id :id	`CollectionController`
`GET`	`/media/:id`	`id::media_id`	Returns a media entry with id :id	`MediaController`
`POST`	`/collection`	`collection payload`	Creates a new collection	`CollectionController`
`POST`	`/media`	`media entry payload`	Creates a new media entry	`MediaController`
`PUT`	`/collection/:id`	`id::collection_id collection payload`	Updates an existing collection with id :id	`CollectionController`
`PUT`	`/media/:id`	`id::media_id media entry payload`	Updates an existing media entry with id :id	`MediaController`
`DELETE`	`/collection/:id`	`id::collection_id`	Removes an existing collection with id :id	`CollectionController`
`DELETE`	`/media/:id`	`id::media_id`	Removes an existing media entry with id :id	`MediaController`
`GET`	`/thumbnails/:thumbnail`	`thumbnail::file_name`	Returns a thumbnail with file name :thumbnail	`MediaController`
`GET`	`/playNotifications`	–	Returns a list of play events	`MediaController`
`SUBSCRIBE`	`/notifications`	–	WebSocket. Publishes events when data changes occur	–

6.6.4.2 Controllers

As seen in the table in the previous sections, all API endpoints are handled by one of two controllers. The design of the Controllers is such that they face one single responsibility, that is, to extract all necessary data from the bare bones HTTP Request, sanitizing the data if necessary, and passing the data on to the *MediaStore*.

An example of a Controller method:

```
  async createCollection(req, res) {
    console.log('createCollection()');

    let { title } = req.body;
    title = title.replace(/[\u00A0-\u9999<>\&]/gim, '');
    const { active } = req.body;
    const { mediaEntries } = req.body;

    const collection = await this.mediaStore.createCollection(title, activ
e, mediaEntries);
    res.send(JSON.stringify(collection));
  }
```

The above example shows the single responsibility of a controller method to extract data from a Request and pass it on to the *MediaStore*.

6.6.4.3 MediaStore

After the router has passed the HTTP request to the controller method, the controller calls the corresponding *MediaStore* method with the formatted parameters. In the previous example, this would be the `mediaStore.createCollection` method.

The *MediaStore* is responsible for doing the actual work of maintaining the catalog and storing media files on disk. It does this by interacting with the storage back-end, which is *Mongoose*, and directly with the file system.

An example of a *MediaStore* method:

```
  async createCollection(title, active, mediaEntries) {
    const collection = new MediaCollection({
      title,
      active,
      mediaEntries,
    });
    const res = await collection.save();

    this.notify(new Notification('collection', res._id, 'create'));
    return MediaCollection.findOne({ _id: res._id })
      .populate('mediaEntries')
      .exec();
  }
```

6.6.4.4 *Catalog*

Keeping track of all the collections and media entries, as well as files on the file system, is done by a *MongoDB* database. Because *MongoDB* is schemaless by default, *Mongoose* is used as a front-end to *MongoDB*. By doing this, it is possible to define a schema in JavaScript.

The following is the complete description of the *media server* database schema:

```
const mediaCollectionSchema = new mongoose.Schema({
  title: String,
  active: Boolean,
  mediaEntries: [{ type: Schema.Types.ObjectId, ref: 'MediaEntry' }],
});

const mediaEntrySchema = new mongoose.Schema({
  title: String,
  description: String,
  file: { type: Schema.Types.ObjectId, ref: 'MediaFile' },
  thumbnail: { type: Schema.Types.ObjectId, ref: 'MediaFile' },
});

const mediaFileSchema = new mongoose.Schema({
  originalFileName: String,
  filePath: String,
  contentType: String,
  hash: String,
});

const playNotificationSchema = new mongoose.Schema({
  time: {
    type: Date,
    default: Date.now
  },
  file: { type: Schema.Types.ObjectId, ref: 'MediaFile' },
});
```

The catalog consists of *Collections* which can have zero or more *MediaEntries*. A *MediaEntry* is a type representing a single media file and its metadata. A *MediaFile* holds the path and metadata of a media file persisted on disk. A *PlayNotification* holds information about a *play event*.

6.6.4.5 *Notifications*

The mechanism by which the *pod controller ui* is informed about new media available to be played is handled by a *WebSocket* that publishes such updates on the `/notifications` endpoint. The *pod controller ui* is subscribed to this endpoint.

By doing this, we achieve the goal of immediately updating the *pod controller ui* when new media files or collections ar added or updated.

The notification format is specified as follows:

```
export default class Notification {
  constructor(entity, id, method) {
    this.entity = entity;
    this.id = id;
    this.method = method;
  }
}
```

This results in messages that look like the following:

```
{"entity":"mediaEntry","id":"5fae332238c716285ab9f18b","method":"update"}
{"entity":"mediaEntry","id":"5fb8d5e9e3f5967b9ed8b556","method":"create"}
{"entity":"collection","id":"5fb8d601e3f5967b9ed8b557","method":"create"}
```

This allows the *pod controller ui* to react accordingly, which can include reloading or discarding the necessary data.

The notification format was intentionally designed to *not* include the names of the fields inside a *MediaEntry* or *Collection* that were changed.

The argument could be made that doing so would allow for more granular re-fetching of data by the *pod controller ui*. Although this would only be possible if there were such a capability provided by the *media server* in form of an exposed RESTful HTTP route, which is not the case.

Therefore, even if only the *title* of a *MediaEntry* or *Collection* changes, the whole entity is re-fetched.

6.6.4.5.1 Source of Notifications

The *MediaStore* is the component responsible for handling data changes, both to the database and the file system. Because of this, it is reasonable to outfit the *MediaStore* with the capability of publishing notifications about those changes.

This is done by the following method inside the *MediaStore*

```
notify(notification) {
  if (this.wss == null) return;
  this.wss.clients.forEach((client) => {
    if (client.readyState === WebSocket.OPEN) {
      client.send(JSON.stringify(notification));
    }
  });
}
```

We intentionally chose to give the *MediaStore* this capability, despite it being a violation of the *separation of concerns* principle. We justify this by showing the whole implementation of this feature in the code snippet above. Because it is so small and isolated, we deemed it unnecessary to extract this into its own service, say a *NotificationService*.

However, it should be noted that, should the need for more sophisticated notifications ever arise, extracting it into a service would be simple. That is because of the already implemented *dependency injection* pattern, which could provide the *MediaStore* with such a service.

6.6.4.5.2 Web Socket

The *media server* effectively runs a second, independent *WebSocket* server under the *same* host-name, but reachable via a dedicated API endpoint (`/notifications`). This has one implication that necessitates a sub optimal solution.

The code below is simplified but representative of what happens inside the *media server* on startup:

```
/*1*/ const server = http.createServer(app);
/*2*/ server.listen(port, () => {});
/*3*/ const wss = new WebSocket.Server({ server, path: '/notifications' })
;
/*4*/ mediaStore.enableNotifications(wss);
```

It should be noted that line 3 depends on `server`, created on line 1. This is how one can integrate a *WebSocket* server into an existing, non-web-socket server. Consequently, the *MediaStore*, which somehow needs to obtain the notification *WebSocket*, has to get it *after* `server` has been created.

The obvious solution would then be to initialize it *after* `server` and `wss` have been initialized. This cannot be done however, because `server` itself transitively depends on the *MediaStore* (via the router).

We resolve this awkward situation by manually enabling notifications in the *MediaStore* after it has been initialized (`enableNotifications(wss)`).

It sould be mentioned that the aforementioned possibility of extracting the notification handling into its own service and injecting it into the *MediaStore* would *not* solve this issue. The dependency would simply go through one more step of indirection.

6.6.5 Trying it together

The components explained above are integrated into the *media server* by means of *dependency injection*. This can be seen in various places.

The router obtaining the controllers it needs:

```
app.use('/', routes(collectionController, mediaController));
```

The controllers obtaining the *MediaStore*:

```
const mediaController = new MediaController(mediaStore);
const collectionController = new CollectionController(mediaStore);
```

6.7 Testing

6.7.1 Media Manager

This chapter will document the testing approach for the media manager. To have less testing overhead, only the server side was tested using unit tests and the react application was tested using cypress. To be able to run the cypress tests it is required to have an instance of both software components running on “localhost:3000” or on the specified port, that the react app got started on.

6.7.1.1 *Unit Tests Express*

The unit tests for the express server mock away the user interface and therefore the user input using chaiHttp. The two components mediaController and collectionController are tested seperately.

6.7.1.2 *React UI-Tests with Cypress*

The integration tests are fully automated using the cypress framework. This was probably one of the most challenging tasks during this project, since it meant that all the other components were able to build and run successfully. Integration tests were the most vulnerable part of the continuous integration, since whenever one component was added or changed there was a big likelihood for the integration test suite to fail.

6.7.2 Media Server

By using the *chai-http* package, testing can be done in an uncomplicated way. A *media server* instance is spun up during every CI build by *chai-http*, which allows for realistic tests that begin as HTTP requests and go all the way through the application until they end up in the database.

This requires a database to be present in the CI environment as well.

An example for a test is given below:

```
it('it should GET all the media entries', (done) => {
    chai.request(server)
    .get('/media')
    .end((err, res) => {
        res.should.have.status(200);
        const mediaEntries = JSON.parse(res.text);

        mediaEntries.should.be.an('array');
        mediaEntries.should.have.lengthOf(2);
        mediaEntries.forEach((me) => me.should.have.property('_id'));

        done();
    });
});
});
```

This way of testing allows for the traversal of the whole application, beginning at the router, through the controller and into the media store. By doing this, we can ensure that all pieces fit together nicely. It also acts as a reasonable regression testing setup in the sense that if a change is made somewhere in the application that has a dependency downstream, at least one test covering that execution path should fail.

Those kinds of tests lies somewhere between unit tests and integration tests. It is obviously not unit tests, because that would be a lot more isolated. It is also not integration tests, because that would mean that the requests have to originate from the *media manager ui*.

6.7.3 Media Service

Testing the media service requires a lot of mocking. The reason for that lies in the nature of the service, essentially being a thin wrapper around various operating system utilities and libraries. All of these operating system dependencies need to be mocked, as they are specific to the raspberry pi hardware and its ARM architecture, which would even prohibit them to work correctly in a containerized environment under an x86 architecture. The solution is a heavily mocked test setup, abstracting away the player and all system utilities.

Once the necessary components are mocked, writing tests for the media service looks as follows:

```
def test_skip_video(self):
    # Arrange
    reply_socket = SocketMock()
    player = PlayerMock()
    oc = None
    ms = MediaService(reply_socket, player, oc)

    # Act
    ms.dispatch({"skip_video": 25})

    # Assert
    expected = {'volume': 50, 'video': 'test.mp4', 'length': 30, 'timestam
p': 25}
    self.assertEqual(expected, reply_socket.output)
```

All dependencies of the `MediaService` which have an operating specific, and therefore ARM specific, implementation are mocked. Only the basic requests and replies are tested. While this test suite does not constitute a fully tested implementation, it nevertheless acts as a useful regression testing setup in the case of future feature additions or refactorings.

6.8 Continuous Integration

The CI is divided into two stages, a build and a test stage. We also divide the project into the server and ui components so that they can be built and tested separately.

During a CI pipeline execution a database container is spun up in order to allow for the cypress integration tests to run.

Coverage results as well as the video of browser interactions performed by cypress are made available for download on each pipeline execution for the developers to inspect.

The following image shows the two stages, build and test:

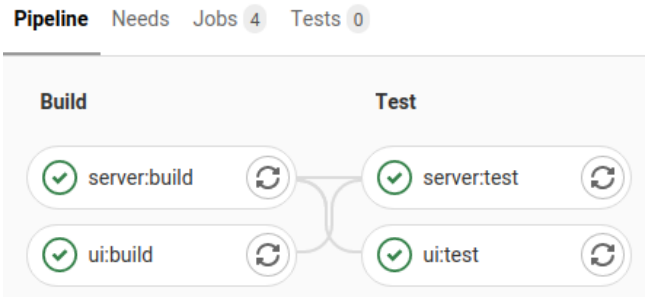


Figure 17: Continuous Integration

We lack a deployment stage. This is due to the dynamic location of the pod and the fact that it is neither available under a static IP address or host-name. It is not even guaranteed to be turned on and connected to a network.

Because of this, the last step in the deployment is a manual pull, performed using git on the pod itself, followed by a restart of the newly updated service.

This is not a perfect solution, but already an improvement compared to the previous method of copying the files by hand onto the raspberry pi inside the pod.

7 Results

In this chapter the results of the project are presented. Therefore the chapter is split up into the different sections that represent the parts of the resulting system.

7.1 Media Manager UI

The media manager as the main result of this project, was the most crucial part of software contributed to the existing system. As described in the chapter "Implementation", the *media manager ui* is composed of following components, that all have their own data responsibilities. The appendix contains the user manual, describing the main use cases as they are depicted in the chapter analysis.

7.1.1 File upload

The file upload process is started by clicking the plus, then selecting a file, either image, audio or movie and then clicking "okay" and the onChange handler checks size and file type to decide whether this is a media entry, if it is audio or video or if it is a thumbnail. Then the entry is created either with the thumbnail and no current file or with a media entry and a default thumbnail.

7.1.2 Update Media Entry

The update process of the media entry is achieved by clicking the media tile, changing title and description, adding thumbnail or a new file and then pressing the update button to confirm the changes. The file size of each file will be crosschecked again by looking up the available disk space on the device and if the file is too big, an error message will be shown.

7.1.3 Delete Media Entry

A media entry can be deleted by clicking the "X" on the top right of each media tile. It is important to notice that this button deletes the file from the pod and every collection, which is why the warning message appears in case a user tries to delete the file from a collection.

7.1.4 Create Collections

A collection can be created by adding a collection title in the title field and then pressing the plus button next to the title field. The collection that is created will be automatically selected and every file uploaded, while the collection is selected will automatically be added to the collection as well as to the media library.

7.1.5 Manage Media of a Collection

As mentioned before, media can be added to a collection by having the collection selected when uploading the media or when the updating form is active, by checking the checkbox next to the collection. To remove the media from the collection the update process can be activated and the checkbox can be unticked.

7.1.6 Statistics

To view the play counter of the current system, the navigation link "statistics" can be selected and the statistics view will be displayed. The width of the counter bar from the media entry with the maximum count is always maximum size and all other counter bars are

adjusted accordingly, so that the relative difference is easy to see and the widths never expand over 100% of the view width.

This screenshot will give an example of how the count numbers are displayed.

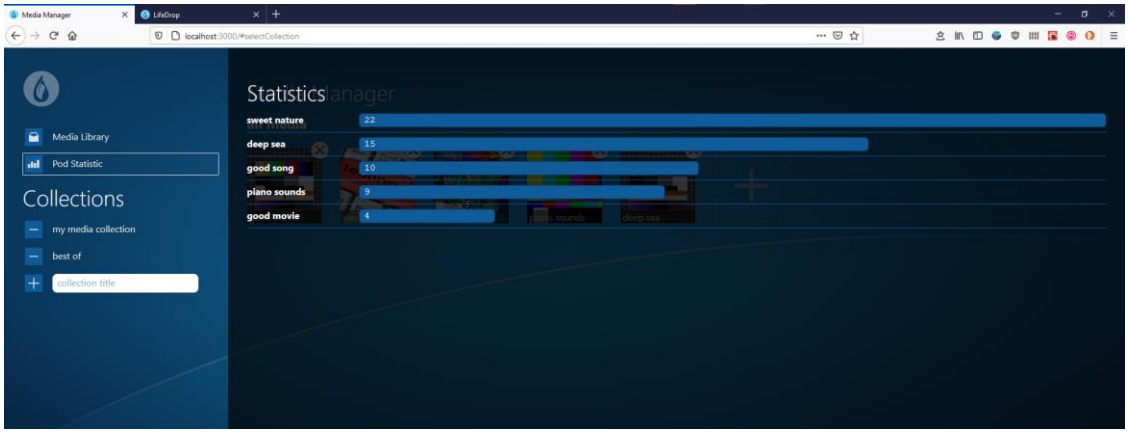


Figure 18: Statistics view

7.2 Media Server

The media server, which is the back-end of the media manager, was implemented from scratch and accomplishes the tasks listed in the following sections.

7.2.1 Media catalog

Persisting media entries, collections and file metadata is the main job of the media server. This results in the so called media catalog, or media database. It supports the operations listed by the media manager in the previous section.

7.2.2 File storage

The actual media files, which can be large in size, are stored by the media server on the operating system's file system. Files are stored under a filename resulting from their computed hash, which prevents storing the same file under a different name twice, and therefore taking up less disk space.

7.2.3 Automatic Synchronization

The media server pushes updates asynchronously to the pod control ui, where they are updated instantly without additional user interaction. This enables an instant feedback loop for the user of the media manager to design a compelling pod control look and feel with all the different media collections and entries.

7.3 Pod Control UI

The *pod control ui* was redesigned during the project. There are very few changes to the actual functionality of the existing software. The changes are listed as follows:

7.3.1 Pause Function

The user base was requesting that the media playing could be paused on a specific point in time. The play button that was static in the existing software now displays a pause icon whenever the media is playing and if pressed again the media is paused at this point of the timeline.

7.3.2 Skip Function

The user base also requested to be able to skip the media to a specific point on the timeline. This is achieved in the new user interface by showing a progress bar in the bottom of the media tile. When the progress bar is touched, the media skips to the specified point of the timeline.

7.3.3 General Redesign

The background image, colors and fonts of the *pod control ui* now match the design of the *media manager ui.* This makes the applications appear as one unit instead of appearing like interfaces from different origins.

7.3.4 Automatic Synchronization

The *pod control ui* automatically synchronizes the media collections and entries with the *media manager ui* so that the person managing the pod is already seeing all the entries that the pod user will see on his screen. This way there is a maximum of consistency between management software and end user interface.

7.4 Media Service

The media service ended up being a drop in replacement for the existing service, with extended capabilities, which are listed in the next sections.

7.4.1 Player control

The media service implements the familiar playback pattern used by most media playback devices, using the play, pause, resume and stop semantics. This results in a more intuitive way of controlling the playback for the user of the pod.

7.4.2 Skipping

Skipping back and forth in a video is a common thing to do when watching videos and is implemented by the new media service. A progress bar is updated every second to reflect the new position of the video to the user and can be dragged left and right to skip the video back and forth respectively.

7.5 Usability Test

The *media manager ui* software was tested for its usability by the management. There was no black box testing, due to management decision. The following use cases have been tested in a white box test scenario.

7.5.1 Upload Media

- Task: The user has to upload a file to the system.
- Intended process: User clicks on the plus icon, the file system upload dialog shows, the user selects the file and clicks okay.
- Actual outcome: User clicks on the plus icon, clicks cancel. User clicks plus again and selects file, then clicks the upload button.
- Result: Cancel returns undefined file entry, this has to be caught by a defensive clause.

7.5.2 Update Media

- Task: The user has to update the meta information of a file on the system.
- Intended process: User clicks on the media tile, updates media information, uploads file or thumbnail and presses update.
- Actual outcome: User clicks on the media tile, asks why thumbnail field is empty since there already was a thumbnail in uploaded, updates media information, uploads file and clicks update.
- Result: There must be some sort of feedback to the user, whether there is a thumbnail or not.

7.5.3 Create and Update Collection

- Task: The user has to add a new collection and add media entries to the collection.
- Intended process: The user enters a title, the user presses the plus button next to the title, the user adds media entries to the collection via the media update form.
- Actual outcome: The user presses the plus button, a nameless collection is created, the user deletes the collection, the user enters a title and presses the plus button, the user adds the media to the collection according to the user manual.
- Result: Empty strings should not be allowed in the collection creation form.

7.6 Controller

As a result of the goal to modify the controller as little as possible, it ended up staying virtually the same, except for the capability to process two more message types. On a higher level though, no capabilities were added or removed from the controller.

7.6.1 Hood lens

The hood lens for the beamer, whose control via the pod control ui is specified as an optional goal for the term project, has not been installed into the pod as of yet. Because of this, we did not pursue this goal any further.

7.7 Unfinished tasks

After the demo day, where the final state of the term project was presented, a number of outstanding tasks were collected and are listed below.

These open tasks are present as *issues* tagged with the label `sa-lifedrop` on the GitLab repository of LifeDrop GmbH.

7.7.1 Bugs

- Play count statistics does not sync with the pod
- Services on the media pi don't start after booting the minicomputer

7.7.2 UI Changes

- Ability to sort videos inside a collection
- Ability to rename a collection without recreating it
- Displaying a progress bar instead of a spinner during the upload of a file
- Improving the visual feedback of the thumbnail and file upload fields of the upload form to clarify whether a file is present or not

7.7.3 Various

- Defining the properties of video files supported for playback by the raspberry pi

8 Conclusion

This chapter deals with a constructive critical outlook on the results described in the chapter results. It therefore has the same structure as the chapter results including additional chapters for the individual experience during the project.

8.1 Media Manager UI

For implementing the media manager, many decisions had to be made and decisions always have upsides and downsides. The main features can be separated into three different categories, file handling, collection handling and the statistics view. Therefore the following chapters will discuss these three categories in detail.

8.1.1 File Handling

How the file handling is implemented right now, has the advantage that a file of a media entry can be exchanged. This has the upside that if the pod admin wants to change the file, that is linked in the media entry he can do so easily. The files are renamed, using their hash, to prevent remote code execution (RCE), if the same file would be in two different media entries, the file could be deleted on one media entry and for the other media entry, the *pod control ui* and the *media manager ui* would be trying to load an already removed file. This error should not result in an exception, but it is not very convenient and the user may not understand why all other entries are now missing the file as well. This problem was never encountered and therefore of very low priority to fix, since there were much more urgent matters to attend to.

A really important feature would have been a multi file upload. This would have made it way easier to fill the media library and could have saved a lot of time for the pod admin.

8.1.2 Collection Handling

The collection handling at the moment is lacking a lot of features. The most important feature that is missing is that it should be possible to order collections. This feature was requested at the day of demonstration and since it would mean adjusting the media model of the media server and adjusting the CollectionSidebar component as well as the MediaCollection component, it was considered too big of a risk to implement before handing the project in.

Another improvement would be if the media could be ordered in the collection as well. Right now the media seems to appear in upload order. This can be useful, but for adjusting the experience of the user and testing the effect the order of the media has it would be crucial to define a different order and let the items appear in a specified order instead of in a chronological order.

Also it would be incredibly time saving if the whole media library and the entire collection set could be downloaded and imported on another pod. This would have cost way too much time to implement but the potential time saved with this feature may have been worth it.

8.1.3 Statistics View

The statistics view has the most potential. Unfortunately, this component was requested very late into the project, so all it does now is count interactions with the play button. However it would be great if it would show play time, date of plays and different diagrams to display the user interaction with certain media entries.

8.1.4 Code Review

For the code review we would like to thank Mr. Stocker, who was showing support by having a look at the implementation. React is his field of expertise and many implementation details were inspired by his react class for the WE3 module. Unfortunately, the review took place in a comparably late stage of the project. This led to a situation, where only few of the points could be implemented. The rest of them are documented here for future improvements.

The following text is an automatic translation of Mr. Stockers code review by deepl.com

I find the structure clear, however you might start to consider extracting code that is very similar into own components. If you do it too early it can be tedious, but maybe now would be the time. NavigationSidebar, for example, has the two divs that could be pulled out nicely.

The extraction of the components mentioned was noted and indeed, there are many parts in the code, that were similar to each other. The extraction of the specified parts would have had implied changes to other components. Unfortunately, the fixing of crucial bugs like the pause bug was of higher priority. However, in a future refactoring, a more consequent way to apply the DRY principle would be imperative.

Also the Redux code is starting to have a size that you could do multiple files with Actions and Reducers in there as well.

The reducers.js file is approaching the 150 LOC mark, therefore it is of growing importance to start separating the already separately implemented reducers. This would imply a change in the folder structure. In the current situation, the risk of breaking links outweighs the reward of a more compact and neat structure.

Then you have partly a lot of logic in the components, which could also be packed into an action. E.G. UpdateForm. There I find it also difficult, because you have setXY methods which are State and such which are Redux Actions. Maybe you could think about a naming scheme?

This crucial input was really important for the future understanding of the code, therefore all set methods, that are also redux actions were renamed to clearly communicate, their context (i.e. setXYAction).

Attention, "setLoading" seems to be a bug, you use the import and not the prop!

This bug was a real head scratcher. Fortunately, it was interpreted correctly in the end and fixed during the writing of this chapter.

MediaCollection is also a component that is connected to Redux, so you shouldn't have to pass the selectedCollection in App.

This redundancy was removed during the writing of the this chapter.

In the package.json there seem to be some dependencies which should be devDependencies.

Obvious devDependencies, like nyc and cypress coverage were moved to the devDependencies.

The original text is contained in the appendix.

8.2 Media Server

The main advantage of the current media server implementation is how simple and non-spectacular it is. Adding new features is as simple as adding a new controller and modifying existing ones is straightforward too. Only basic JavaScript knowledge is needed to accomplish this.

The media server has few dependencies, which results in very fast build and test times in the CI. This makes for a smooth developer experience.

Because no advanced features were used, such as a static type system, a more expansive framework, or a complex build tool, some static guarantees or configuration options are lacking. This might become a problem once the project becomes significantly larger. The future integration of at least some of those tools should however not prove a significant challenge.

8.2.1 Future updates

The current statistics collection implementation only tracks the play count per video. It could be expanded to collect more information, such as where in the video did the user pause, or when were the lights turned back on manually.

Using the file system to store media files has many advantages, one drawback though is that the disk space is finite. A future update could allow for an "overflow" mechanic that stores files in the cloud, such as in an S3 bucket, as soon as the local disk is close to full.

8.3 Pod Control UI

The *pod contol ui* was not initially designed by the project team. Therefore all the improvement potential is thought of as a constructive input to whoever will work on the software in the future.

8.3.1 Playing Media

The current implementation allows to play the media pressing the play button in the middle of the media tile. This could be improved by opening an entire play interface, where the timeline would be an integral part of the UI. The upside of the current implementation is, that the play button is bigger, more obvious, better visible and can also pause the media. The downside is that the play button is slightly off center due to the Material-UI implementation of the icon button and the pause button appears on every media tile, that shares the same media file.

Skipping media can be very challenging, since the skipping progress bar is on a y-scrollable divider element. This was implemented due to management requirements. The upside is that the scrolling of the collections now operates in a similar way, that the collections are scrolled in netflix. The downside is, that when scrolling the collection you may also skip the media or more likely, when trying to skip the media, you may also scroll the collection containing the media.

8.3.2 Pod Control UI Side Navigation

The side control panel is now as slim as possible. This has the upside that the media collections get more visual space and the downside that the range sliders are a challenge to handle. A possible solution to the problem would be to make the side bar a toggle element. This would require a deep restructuring of the existing code. Compared to the solution before, that was using 30% of the width of the view window, this was an improvement nonetheless. For future application it would be recommendable to add a navigation to the *pod control ui*, so the extension of features can be easily achieved. Due to instructions from management and for keeping as much of the existing code intact, this was not implemented as such.

8.3.3 Seat Heater

Due to management decision, the seat heater control was removed from the *pod control ui.* This part of code is not deleted however, in case the seat heater will be implemented in future generations of the LifeDrop.

8.4 Media Service

Because much of the behavior of the media service is dictated by its surroundings, not much room for flexibility exists for implementing it in a fundamentally new way. The current implementations is a step-up in readability, maintainability and testability, which are mostly technical and not so much functional aspects.

The close interplay between the media service and system utilities such as omxplayer and the underlying hardware, in this case the HDMI controller, places additional constraints and challenges onto the development of a new service. This goes even further, the 32bit Raspberry Pi OS and the ARM architecture play a significant role in what hardware acceleration for video playback is available and which tools work together and which ones do not.

8.4.1 Future updates

Future updates to the media service depend upon whether the raspberry pi remains the device of choice or whether a new minicomputer is used. The media pi runs the media manager, the pod control ui and the media service, all of which need hardware resources to work. To support higher than 4K video resolution a more powerful minicomputer would be necessary, which changes the playing field and opens up new possibilities, all of which are out-of-scope for our term project.

8.5 Controller

As mentioned in previous chapters, not much was changed about the controller. In hindsight this turned out to be the correct choice, as we would not have had enough time to re-implement it. It also did not help that we had no access to the actual pod during most of our development, which would have been necessary to get the different signals from the hardware to test and debug the controller.

8.5.1 Future updates

Functionally, the controller implements all the necessary operations for the current hardware configuration of the pod. Some code-improvements could be achieved by refactoring parts of the code.

- The different ways in which asynchronous threads and tasks are launched and managed could be unified into a single one. A move away from pythons relatively low-level asyncio library to a more abstract library could help. We are not in a position to recommend any specific library as of now.
- The controller has various system-wide dependencies such as the system's python installation and packages, this could be improved by taking advantage of a tool such as poetry, which was used to re-implement the media service.
- The parallel usage of synchronous and asynchronous sockets for communication between components could be replaced by a single, asynchronous socket to simplify the code.

9 Project Management

This chapter will deal with planning the work load and organizing the work force. The planning process will include milestones like in a traditional approach but the time management and the work packets will be somewhat flexible.

9.1 Meetings

The weekly report meetings are on Monday afternoon at 2 o'clock. Due to the situation of the corona virus, the team will meet as often as possible on the teams channel to discuss the weekly progress. The physical meeting of the development team, to check on the progress and solve possible issues takes place on Tuesdays at 3 pm at campus of the former HSR institution now named OST.

9.2 Milestones

The following is an overview of the time line that is setting the deadlines for the project. This may be updated during the course of the project. The project has a time range of 14 weeks. Two students spend around 16.5 hours each per week on the project.

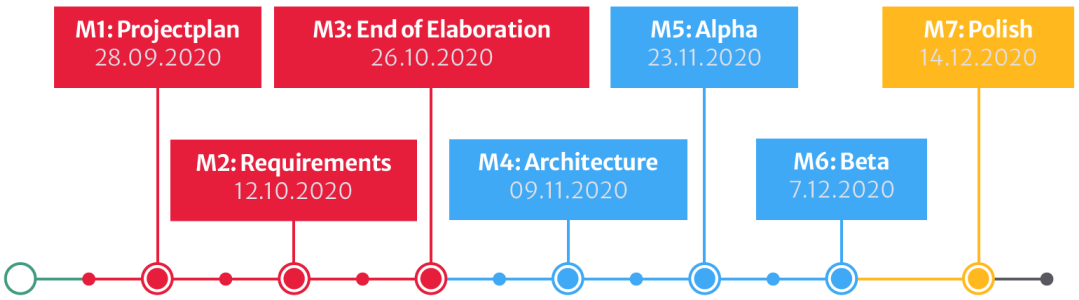


Figure 19: Project timeline

9.2.1 M1 *Project plan*

In this phase we set up a rough structure to plan the following work steps.

Included task list:

- Set up plan
- Define workflow
- Discuss goals
- Discuss necessary infrastructure
- Define scope

9.2.2 M2 *Requirements*

In this phase we define the mandatory and optional features of the application.

Included task list:

- Use Case Diagram
- Non functional Requirements
- Domain Model
- Screen views

9.2.3 M3 *End of Elaboration*

In this phase we see if our initial approach to the solution is a valid choice by making a prototype.

Included task list:

- Architecture prototype
- Demonstration of prototype
- Working tests for prototype

9.2.4 M4 *Architecture*

This phase will deal with the details of the architecture for the prototype. It is meant to assure that the solution will be valid in future scenarios as well as in the current situation.

Included task list:

- Physical architecture
- Logical architecture
- Interfaces (API)
- Database
- Component structure
- User interface
- Possible scenario for managing multiple pods

9.2.5 M5 *Alpha*

This phase will provide the use cases defined as “base use cases”. It will be dealing with the first version of the application and should provide a testable user interface that will be tested by the CEO, Noe Tüfer, or someone with direct authority to authorize necessary changes to the UI.

Included task list:

- Usability test cases
- Usability test results
- Unit tests for specified part of code base
- Specified system test cases
- Code review of system critical parts
- Code statistics

9.2.6 M6 *Beta*

This phase will be used to implement the changes elicited in the usability tests. We will choose the most critical changes and implement them. The changes we can not implement anymore, will be properly documented for future use.

Included task list:

- Implement changes according to usability test results
- Documentation log for usability test results
- Final updates to documentation
- Final system diagram
- Updated tests

9.2.7 M7 *Polish*

This phase serves as time to prepare documentation and also as a buffer zone for eventual issues, that could not be processed during the semester for whatever reason.

Included task list:

- Presentation material
- Presentation test run
- Documentation crosscheck
- Poster

9.3 Development Workflow

The official code of LifeDrop GmbH is hosted on their company GitLab https://gitlab.com/lifedrop. During the development phase of the term project, the relevant code repositories are forked into the *SA LifeDrop* group https://gitlab.com/sa-lifedrop. Results are delivered back via pull requests, which enables an independent and transparent development process.

Inside the *SA LifeDrop* group, development follows a straight forward *feature-branch based git workflow*[2] with mutual code reviews between Lukas and Robin.

The markdown based documentation is hosted on GitLab as well and is available as a generated static website on https://sa-lifedrop.gitlab.io/documentation. *Docusaurus* is the tool used to generate the static documentation website.

9.4 Project Communication

A weekly report is sent to the supervisor Thomas containing a brief update regarding the following points:

- Progress
- Fails/Open Issues
- Plan next week
- Questions/Decisions

This allows for an efficient weekly term project meeting on Monday at 2pm.

A WhatsApp group chat serves as a quick communication channel between all members involved in the term project.

- Students
  - Lukas
  - Robin
- Supervisor
  - Thomas
- LifeDrop GmbH
  - Noe
  - Enzo

E-Mail traffic is kept as low as possible, but permitted if deemed necessary.

9.5 Time Management

Students are awarded 8 ECTS points upon successful completion of the term project. This entails a total workload of 240 hours, which results in a weekly workload of roughly two eight-hour days, or 16 hours.

- Lukas Schiltknecht: 225h total
- Robin Elvedi: 222h total

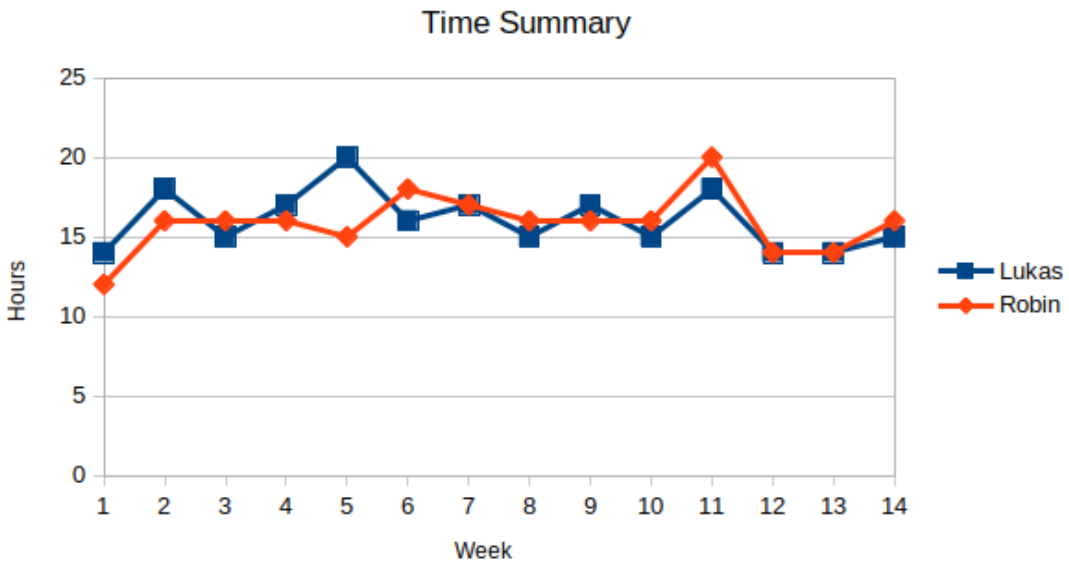


Figure 20: Time Graph

9.6 Repository

- **283** Commits
- **175** Pipelines

9.7 Lines of code

As measured with *cloc*

9.7.1 Media Manager

Language	files	blank	comment	code
JSON	9	0	0	9186
JavaScript	33	344	170	2405
CSS	3	88	4	528
YAML	3	18	1	186
Markdown	1	31	0	37
HTML	1	3	20	20
SVG	9	0	0	9
Bourne Shell	2	0	0	3
———	———	———	———	———
SUM:	61	484	195	12374

9.7.2 Media Service

Language	files	blank	comment	code
Python	5	126	19	402
Markdown	1	14	0	63
TOML	1	3	0	16
Bourne Shell	1	2	0	5
———	———	———	———	———
SUM:	8	145	19	486

9.7.3 Tablet Frontend

Language	files	blank	comment	code
JavaScript	19	148	72	1492
JSON	2	0	0	57
CSS	2	5	0	39
Markdown	1	31	0	37
HTML	1	3	20	16
YAML	1	1	0	12
SVG	3	0	0	9
Bourne Shell	2	0	0	2
———	———	———	———	———
SUM:	31	188	92	1664

9.8 Contributions

As reported by *git-quick-stats*

9.8.1 Media Manager

```
Lukas Schiltknecht:
lines changed: 36367  (73%)

Robin Elvedi:
lines changed: 13451  (27%)
```

9.8.2 Media Frontend

```
Lukas Schiltknecht:
lines changed: 1309   (5%)

Robin Elvedi:
lines changed: 966    (4%)
```

9.8.3 Media Service

```
Robin Elvedi:
lines changed: 2246   (100%)
```

9.9 Test Coverage

9.9.1 Media Server

82.96% Statements 516/622 **68.18%** Branches 45/66 **75.61%** Functions 31/41 **82.96%** Lines 516/622

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements		Branches		Functions		Lines	
server	91.67%	88/96	80%	4/5	100%	1/1	91.67%	88/96
server/controllers	86.36%	133/154	87.5%	14/16	75%	12/16	86.36%	133/154
server/models	100%	42/42	100%	3/3	100%	1/1	100%	42/42
server/services	76.07%	232/305	52.78%	19/36	72.73%	16/22	76.07%	232/305
server/utils	84%	21/25	83.33%	5/6	100%	1/1	84%	21/25

Figure 21: Coverage Media Server

9.9.2 Media Frontend

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements		Branches		Functions		Lines	
src	80.77%	105/130	77.14%	27/35	92.21%	71/77	81.45%	101/124
src/components	89.16%	181/203	76.34%	71/93	93.88%	46/49	97.16%	171/176
src/config	100%	2/2	100%	0/0	100%	0/0	100%	2/2

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements		Branches		Functions		Lines	
App.js	100%	8/8	83.33%	5/6	100%	4/4	100%	7/7
actions.js	100%	27/27	100%	0/0	100%	27/27	100%	25/25
api.js	54.9%	28/51	37.5%	3/8	86.21%	25/29	56%	28/50
index.js	100%	2/2	100%	0/0	100%	0/0	100%	2/2
reducers.js	95.24%	40/42	90.48%	19/21	88.24%	15/17	97.5%	39/40

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements		Branches		Functions		Lines	
CollectionSidebar.js	100%	41/41	95.83%	23/24	100%	15/15	100%	40/40
MediaCollection.js	100%	26/26	83.33%	5/6	90%	9/10	100%	25/25
MediaContainer.js	100%	1/1	55.56%	5/9	100%	1/1	100%	1/1
NavigationSidebar.js	100%	11/11	100%	8/8	100%	4/4	100%	11/11
StatisticsWindow.js	100%	9/9	100%	10/10	100%	4/4	100%	9/9
UpdateForm.js	73.75%	59/80	37.5%	9/24	81.82%	9/11	92.73%	51/55
UploadForm.js	97.14%	34/35	91.67%	11/12	100%	4/4	97.14%	34/35

Figure 22: Coverage Media Manager

10 Glossary

Term	Description
Raspberry Pi	Credit-Card sized multipurpose computer.
Pod	Name for the deck chair like device produced by LifeDrop GmbH.
Docusaurus	Static website generator used for the documentation.
LifeDrop	Name of the company behind the LifeDrop pod.
GitLab	Git repository hosting website, similar to GitHub.
RCE	Short for “Remote Code Execution”. Cyber attack, where code gets executed on a server by uploading malicious code to a machine.
ARM	Short for “Advanced RISC Machine”. Kind of microprocessor architecture. Incompatible to x86.
x86	Kind of microprocessor architecture. Incompatible to ARM.
UI	Short for “User Interface”.
Front-end	User-facing part of an application.
Back-end	Server-side part of an application.
omxplayer	A video and audio player specifically built for the raspberry pi.
omxplayer-wrapper	Python library for interacting with omxplayer.
mpd	Audio player daemon.
mpc	Command line client for mpd.
fbi	System utility. Used to output a static image over HDMI.
ZeroMQ	A message passing protocol for both synchronous and asynchronous communication.
zmq	Short for “ZeroMQ”
alsa	Short for “Advanced Linux Sound Architecture”. Used to configure sound devices on linux.
amixer	Tool for interacting with alsa.
Media catalog	Name for the database of the media manager.
asyncio	Python library for asynchronous programming.
deepl.com	Website for translating texts.
sh	Python library for interacting with system utilities.
System utilities	Programs installed on the operating system.
JSX	Javascript syntax extension for React.
React	JavaScript library for building user interfaces.

11 References

Reference
“Anleitung Dokumentation_HS20.pdf”, MS Teams. [Online]. Available: https://teams.microsoft.com/l/file/CD418C9E-2600-4A83-B1DC-DA59166FC4C4. [Accessed: 25.09.2020]
“Git Feature Branch Workflow”, Atlassian. [Online]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow. [Accessed: 25.09.2020].
“GitLab CI/CD”, GitLab [Online]. Available: https://docs.gitlab.com/ee/ci/. [Accessed: 14.12.2020].
“Poetry Introduction”, Poetry [Online]. Available: https://python-poetry.org/docs/. [Accessed: 14.12.2020].
“Raspberry Pi Documentation”, Raspberry Pi [Online]. Available: https://www.raspberrypi.org/documentation/. [Accessed: 14.12.2020].
“PyZMQ Documentation”, ZMQ [Online]. Available: https://pyzmq.readthedocs.io/en/latest/. [Accessed: 14.12.2020].
“omxplayer API Docs”, omxplayer-wrapper [Online]. Available: https://python-omxplayer-wrapper.readthedocs.io/en/latest/omxplayer/. [Accessed: 14.12.2020].
“systemd Service unit configuration”, Freedesktop [Online]. Available: https://www.freedesktop.org/software/systemd/man/systemd.service.html. [Accessed: 14.12.2020].

12 List of Figures

(images were created for this document where not otherwise declared)

13 Appendix

LifeDrop Developer Manual

Table of Contents

1 Developer Manual

This manual describes the set-up and usage of a new media pi. The first section describes how to install the dependencies as well as the media service software. The second section describes how to use the media pi.

1.1 Installation and configuration

- Using the default user `pi` with default home at `/home/pi`

1.1.1 Operating System

- Raspberry PI OS 32bit
- Can be installed onto the SD-Card using the `rpi-imager` utility

1.1.2 Hostname

- The hostname has to be set to `lifedropmedia`
- This can be done using the `raspi-config` utility

1.1.3 System Dependencies

- Required by the media service and other system utilities

```
sudo apt-get install libglib2.0-dev libdbus-1-dev libffi-dev libssl-dev
fbi omxplayer mpd mpc
```

1.1.4 Install Docker

- Used to run services inside containers
- Requires a reboot afterwards

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
sudo usermod -aG docker pi
sudo pip3 -v install docker-compose
```

1.1.5 Install Docker Compose

- Used to manage the containers and their dependencies among each other

```
sudo pip3 -v install docker-compose
```

1.1.6 Install Poetry

- Used to manage the python environment

```
curl -sSL -o get-poetry.py https://raw.githubusercontent.com/python-
poetry/poetry/master/get-poetry.py
```

The install script requires a manual patch on line `641`:

```
- allowed_executables = ["python", "python3"]
+ allowed_executables = ["python3", "python"]
```

- The installer can be run with `python3 get-poetry.py`

1.1.7 Install Media Service

- Used for listening to incoming messages and executing the appropriate system commands to control media playback

```
cd
git clone git@gitlab.com:sa-lifedrop/media-service.git
cd media-service
poetry install
./install_service.sh
```

1.1.8 Install Media Manager

```
cd
git clone git@gitlab.com:sa-lifedrop/media-manager.git
```

1.1.9 Install Media Frontend

```
cd
git clone git@gitlab.com:sa-lifedrop/frontend.git media-frontend
```

1.1.10 Configure MPD

```
cd /etc
sudo ln -s /home/pi/media-service/mpd.conf mpd.conf
```

1.2 Usage

- The media service runs as a `systemd` process due to the tight integration with operating system utilities such as `omxplayer`
- The other services (media manager and media frontend) run as docker containers
- Logs from containers can be inspected with `docker logs -f <container-name>`

1.2.1 Run Media Service

- Starts automatically after the raspberry pi boots
- The logs can be inspected with `journalctl -u media_service.service`

1.2.2 Run Media Manager

- Starts automatically after the raspberry pi boots
- Can be manually controlled with `./start.sh` and `./stop.sh` scripts

```
cd media-manager
./start.sh
./stop.sh
```

1.2.3 Run Pod Control UI

- Starts automatically after the raspberry pi boots
- Can be manually controlled with `./start.sh` and `./stop.sh` scripts

```
cd media-frontend
./start.sh
./stop.sh
```

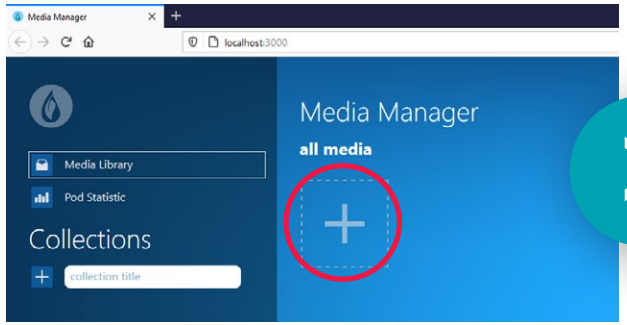
Life-Drop

Media-Manager

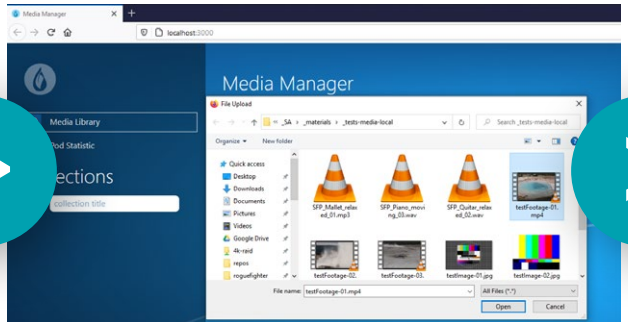
This is the user manual to the media-manager of life-drop.
For further information please consult ***life-drop.ch***

Contents

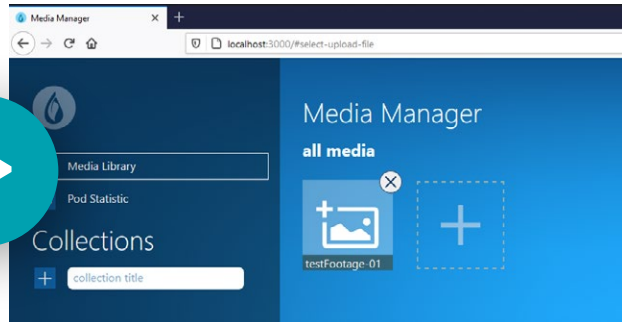
Uploading media to the pod



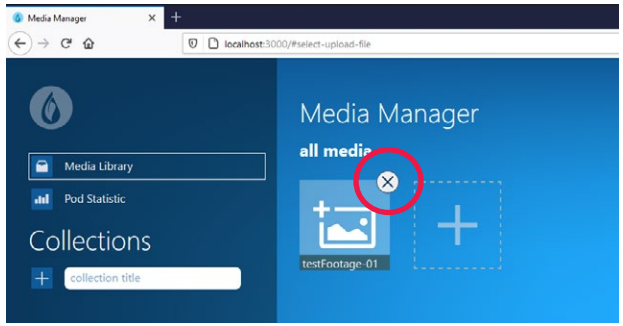
Step 1: Select add button



Step 2: Select Media to add



Step 3: Upload Media



Warning: To **delete** media from pod click on the "x"-button. This button will **always delete the media from every collection on the pod.**

Note: Following file sizes and resolutions are recommended for the media upload. Higher resolutions and bigger file sizes may lead to **unexpected issues.**

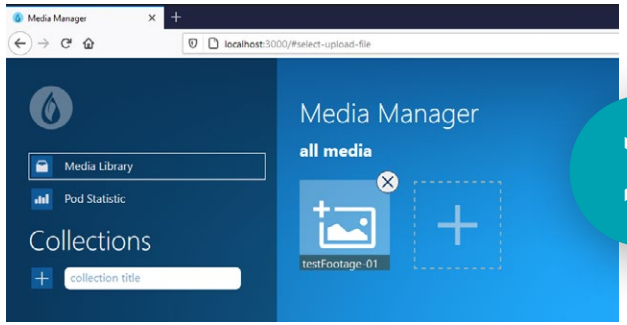
File size: 2–4 GB

Resolution: 1280 × 720 (pixel)

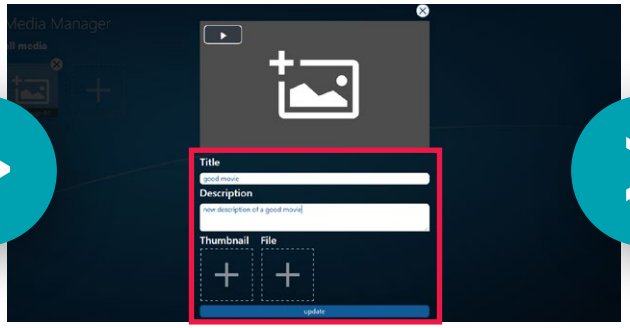
Codec: H264 - MPEG 4 AVC (recommended)

(all other media specifications can be tried but there is no guarantee for appropriate processing due to memory limitations)

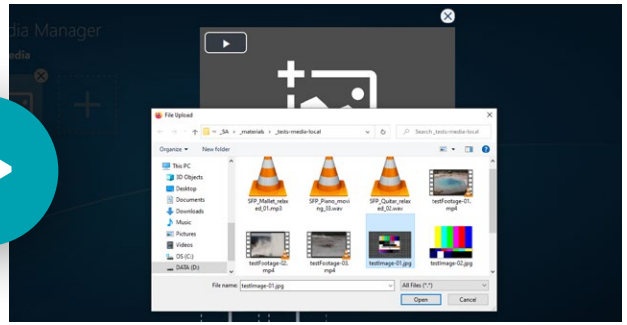
Updating media info



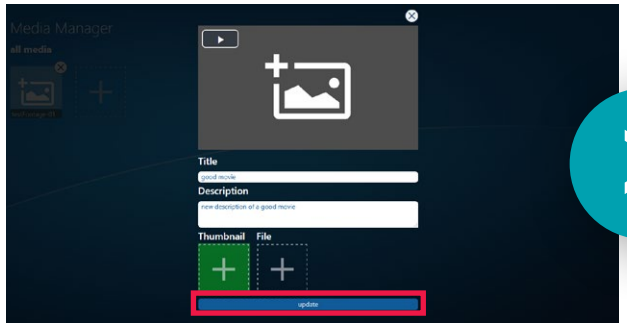
Step 1: Click on media tile



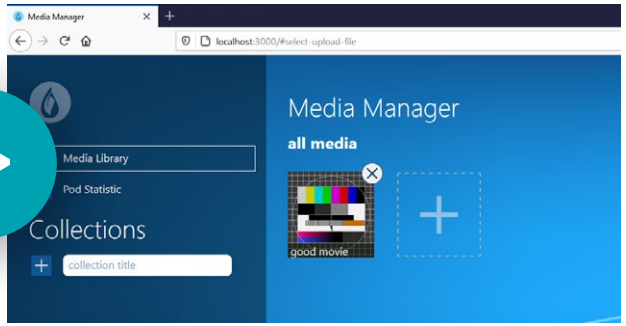
Step 2: Update media info



Step 3: Upload thumbnail

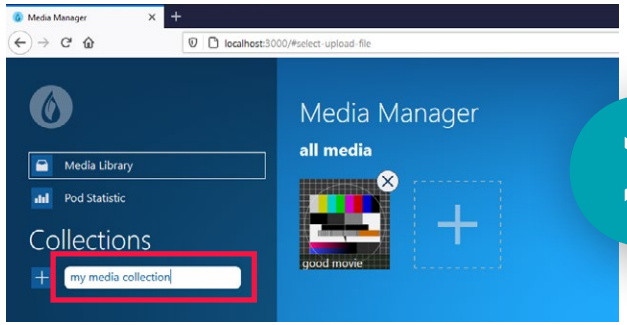


Step 4: Click update button

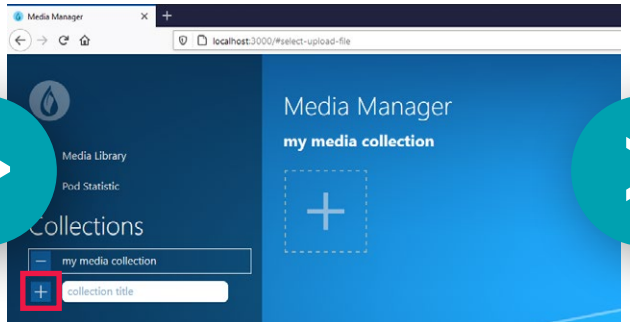


Step 5: Check updates

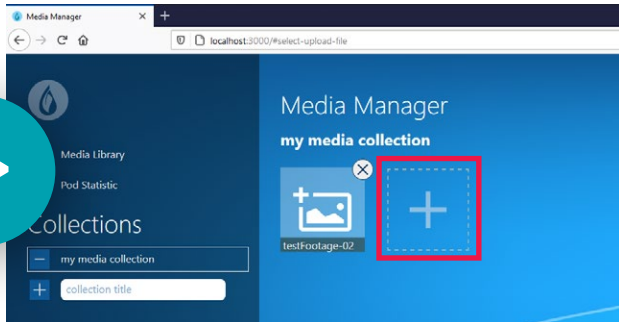
Create collection



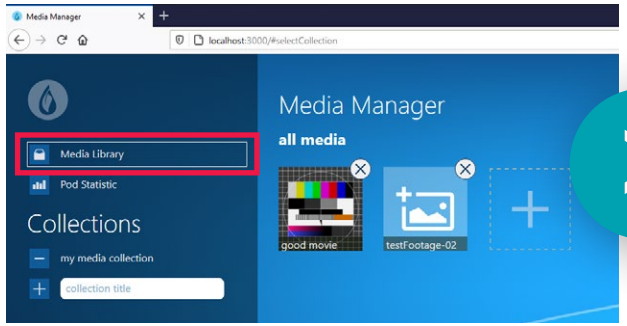
Step 1: Enter collection title



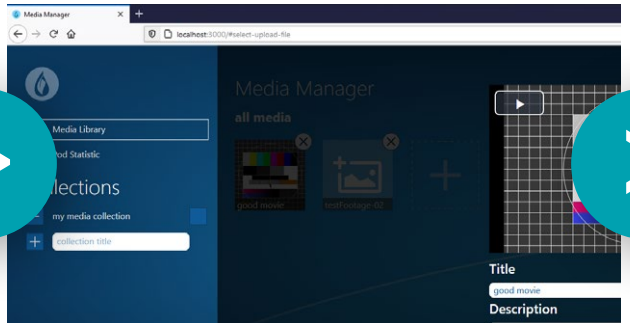
Step 2: Press add button next to title



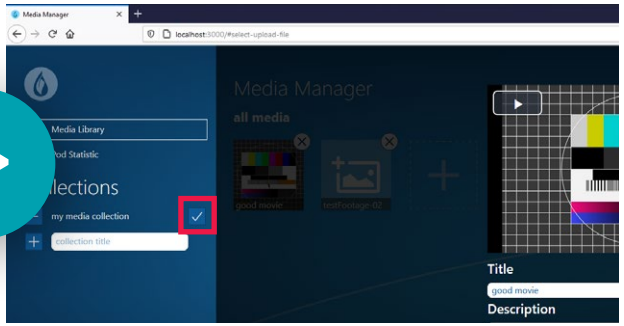
Step 3: Add media to collection directly



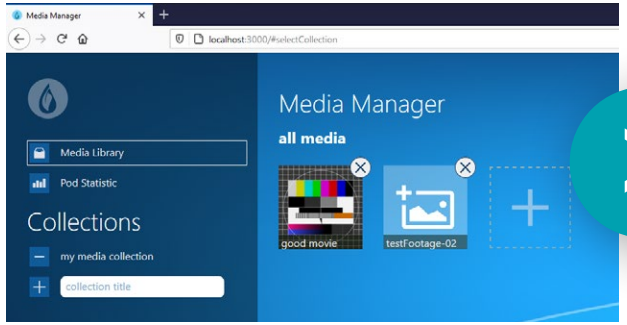
Step 4: Or select “Media Library”



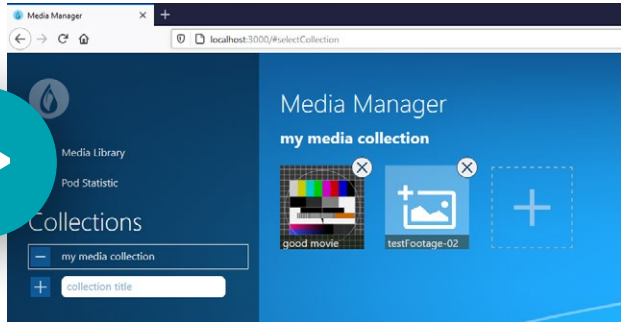
Step 5: Open update form



Step 6: Check collection checkbox

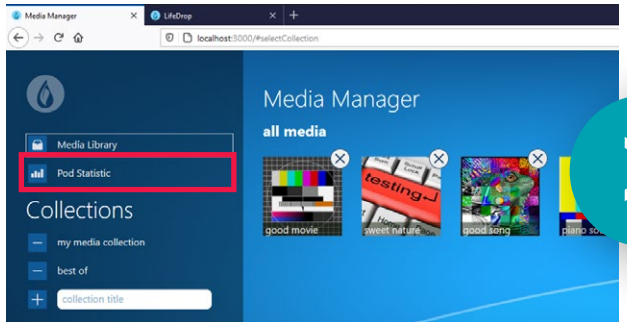


Step 7: Close update form

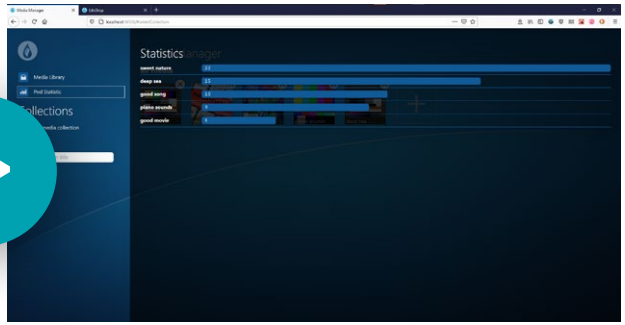


Step 8: Check if media is added

Check player count



Step 1: Select Pod Statistics



Step 2: Check the play counter