DESIGN SPECIFICATION

for

BLC's Automated Light Guide

Prepared by
Emil Hu, Bastian Kramer, Magnus Tang, Jens Fisker, Daniel Biørrith

Department of Engineering, Aarhus University

May 25, 2021

Contents

Revision History

Name	Date	Reason for Changes	Version
Magnus Tang, Emil Hu, Daniel Biørrith, Jens Fisker & Bastian Kramer	13/03/2021	Initializing the document and making deployment diagram	01
Magnus Tang, Emil Hu, Daniel Biørrith, Jens Fisker & Bastian Kramer	20/03/2021	Description of deployment diagram and making sequence diagram	02
Magnus Tang, Emil Hu, Daniel Biørrith, Jens Fisker & Bastian Kramer	22/05/2021	Adding development and logical view	03

1 Introduction

The purpose of this report is to document the architecture and the design of BLC's automated light guide, as well as give a description of the implementations build upon said design. The designs are based on the requirements set in our Requirements Specification document [1]. The architecture and design of our system will be presented using UML diagrams and the implementation will be explained with the use of code snippets.
Like the requirement specification, this report is developed iteratively to ensure that the costumer demands are met. Thus, updates to this document are approved by a common meeting between the costumer and the developers. The changes can be found in the *Revision History* section.

1.1 Kructhen's 4+1 Model

To describe the architecture, Krutchen's 4+1 view has been applied [2] in order to obtain the best description of the architecture as it is not possible to represent the whole system in one diagram [3]. This also allows us to look at the design from different views, which allows for a greater variety of issues to be addressed.

The Kructhen's 4+1 view is depicted in figure 1.1.

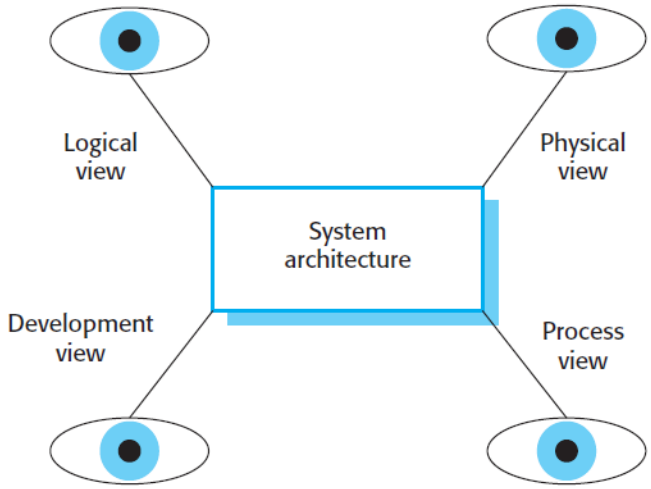

Figure 1.1: Krutchen's 4+1 view [3]

The logical view is concerned with the functionality of our system. It will be showcased in the form of a UML class diagram and a state transition diagram. The process view will be presented with a sequence diagram, which shows the dynamic aspects of our system, as well as the intended behaviour of the system. A UML deployment diagram is highlighted in the physical view in order to visualize the relationship between the hardware and the software of our system. Lastly, the development view illustrates the system from a programming perspective and this view will be presented using code snippets from the implementation of BLC's automated light guide.

2 Physical View

This section will present the high-level system architecture of BLC' automated light guide, illustrated in the form of a deployment diagram to show how the hardware and software interact in BLC's automated light guide as well as a description of the pattern the system is structured around.

2.1 Architecture Pattern

The architecture of BLC's automated light guide is based on the Model-View-Controller (MVC) design pattern, where an illustration of the pattern is depicted in figure 2.1. MVC is build around the three classes: **model**, **controller** and **view**.
The model is the data source and describes how data is structured, which corresponds to the data from the PIR motion sensors. The view is the the user interface, which displays the information about the model to the user. For this system the web application acts as the view. Finally, the controller is the link between model and the view and therefore there is never direct communication between the model and the view. This is handles in the software implementation and the MQTT broker, which will be elaborated further in the next sections [4].

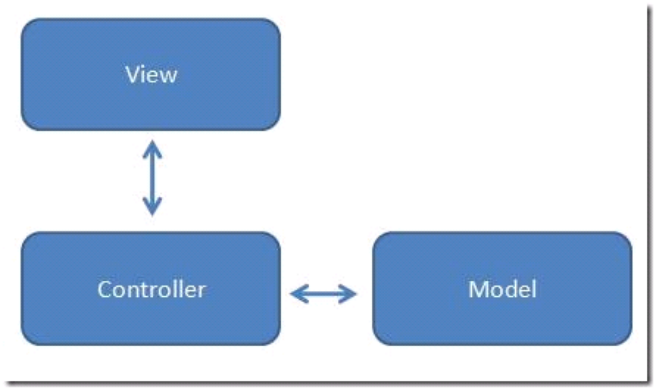

Figure 2.1: Model-View-Controller design pattern [4]

2.2 Deployment Diagram

The deployment diagram of the system is depicted in figure 2.2. As seen on the diagram, it consists of the following components:

- Raspberry Pi 4 (see NR3 in section 3.4 in [1])
- NodeJs webserver, running `BLCWebapp.js`
- Windows client that accesses the application via a browser
- Aqara motion sensors and LED actuators
- A Zigbee USB controller

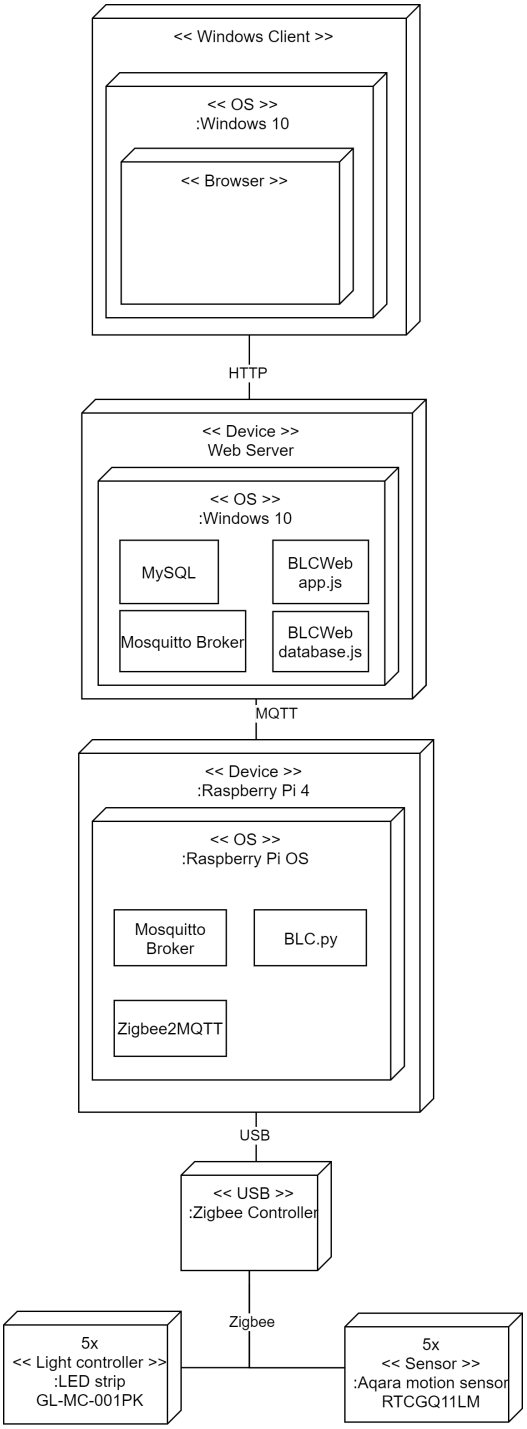

Figure 2.2: UML Deployment diagram of the whole system

Starting from the bottom of the diagram, the motion sensors and LEDs are connected to the Raspberry Pi 4 via the USB controller, which is used to interact the devices. The devices are communicating over a Zigbee network, where the Raspberry Pi works as a hub in the network [5]. The use of Zigbee allows to connect a larger amount of devices to the same network while still obtaining low power consumption. Unlike if the system were to run on either a WiFi or Bluetooth network, where there either is a large energy consumption or a limited amount of devices allowed on a single network. The trade-off is the data rate, but as the system is not required to run at a extremely high-speed, Zigbee is the optimal network for this system [6].

The data from the sensors are sent to the Raspberry Pi 4 [7]. The data are handled by our software implementation, our Python program `BLC.py`, as well as the MQTT broker, which is an broker-based model [8]. Zigbee2MQTT, a NodeJS application, is used to connect the Zigbee network and devices to the Mosquitto broker [9].

The system shall include a web application (NR5 in [1]), where the stored monitor data shall be accessible to the user. In order to do so, the captured data from the Raspberry Pi 4 will be sent to a MySQL database, which will store the data (NR6 in [1]).

The data itself will be sent from the Raspberry Pi to the databse using MQTT and two Mosquitto brokers, where the RaspBerry Pi 4 acts as the publisher for the database (see NR2 in [1]). The web application will be assembled to get a graphical interface (GUI) for the user, where the data from the MySQL database is transferred using the Hyper Text Transfer Protocol (HTTP) [10].

3 Process View

This section will present the process view of BLC's automated light guide, to give a better understanding on the behaviour of the system. This will be showcased in the form of a sequence diagram.

3.1 Sequence Diagram : User Movement

In the sequence diagram, depicted in figure 3.1, it can be seen how movement in a room shall result in a turned on LED and stored data. First, the motion sensor registers movement and sends the event to Z2M. Z2M publishes the event to the broker, on a topic subscribed to by our BLC App. The app processes the information, publishes the data on the broker located at the BLC Webserver and publishes a state change on the local broker. The BLC Webserver logs the received data in the MySQL database, which can be accessed at a later time.

4 Logical View

This section will present the logical view of BLC's automated light guide. Here, a class diagram and a state transition diagram has been made to showcase the functionality of the system.
How the classes and the state machine are implemented will be covered in the *Development View* section.

4.1 Class Diagram

Figure 4.1 shows the UML class diagram of BLC's automated light guide.
With regard to the MVC pattern, it can be seen that system has a model class and a controller class, whereas the webserver acts as the view class. In addition to these, the system also consists of the following classes:

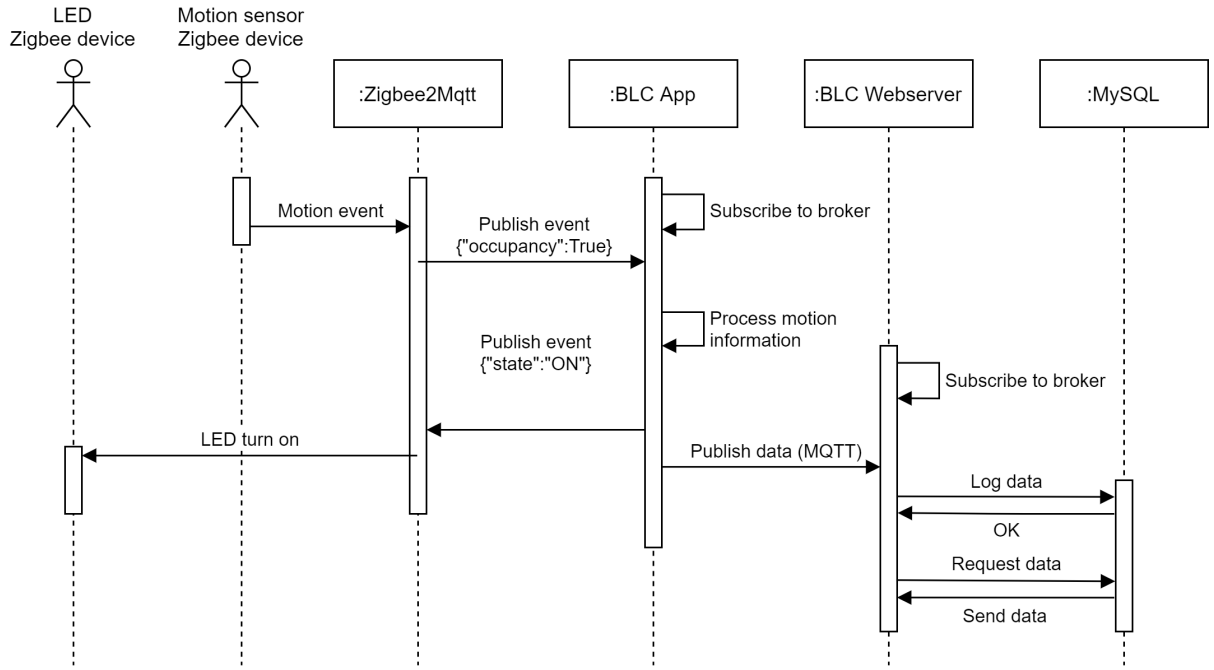

Figure 3.1: UML Sequence Diagram showing the system behaviour in case of movement

- BLCZigbee2MQTTClient
- Statemachine
- Database
- Publisher
- HEUCOD

Model

As stated in section 2.1, the model class is responsible for representing and structuring the data. Thus, the class consists of two dataclasses, which represent the LEDs and the PIR motion sensors. The two functions `add()` and `find()` makes it possible for the system to add multiple devices and finding existing devices already in the system, respectively.

BLCZigbee2MQTTClient

The BLCZigbee2MQTTClient class represents a client, which allows for the controller to interact with the Zigbee2MQTT events. It is in this class where the structure of a Zigbee2MQTT message is implemented as well as all the different types of messages.

Controller

The controller is the focal point of the BLC's automated light guide since it manages all the interactions between the model and the web application. Therefore, the controller class is directly connected to the Model, BLCZigbee2MQTTClient and StateMachine classes as well as the broker as seen on figure 4.1.
The controller registers the events from Zigbee2MQTT that have been published to the MQTT broker and handles them. The controller processes the messages by forwarding the messages to the state machine.

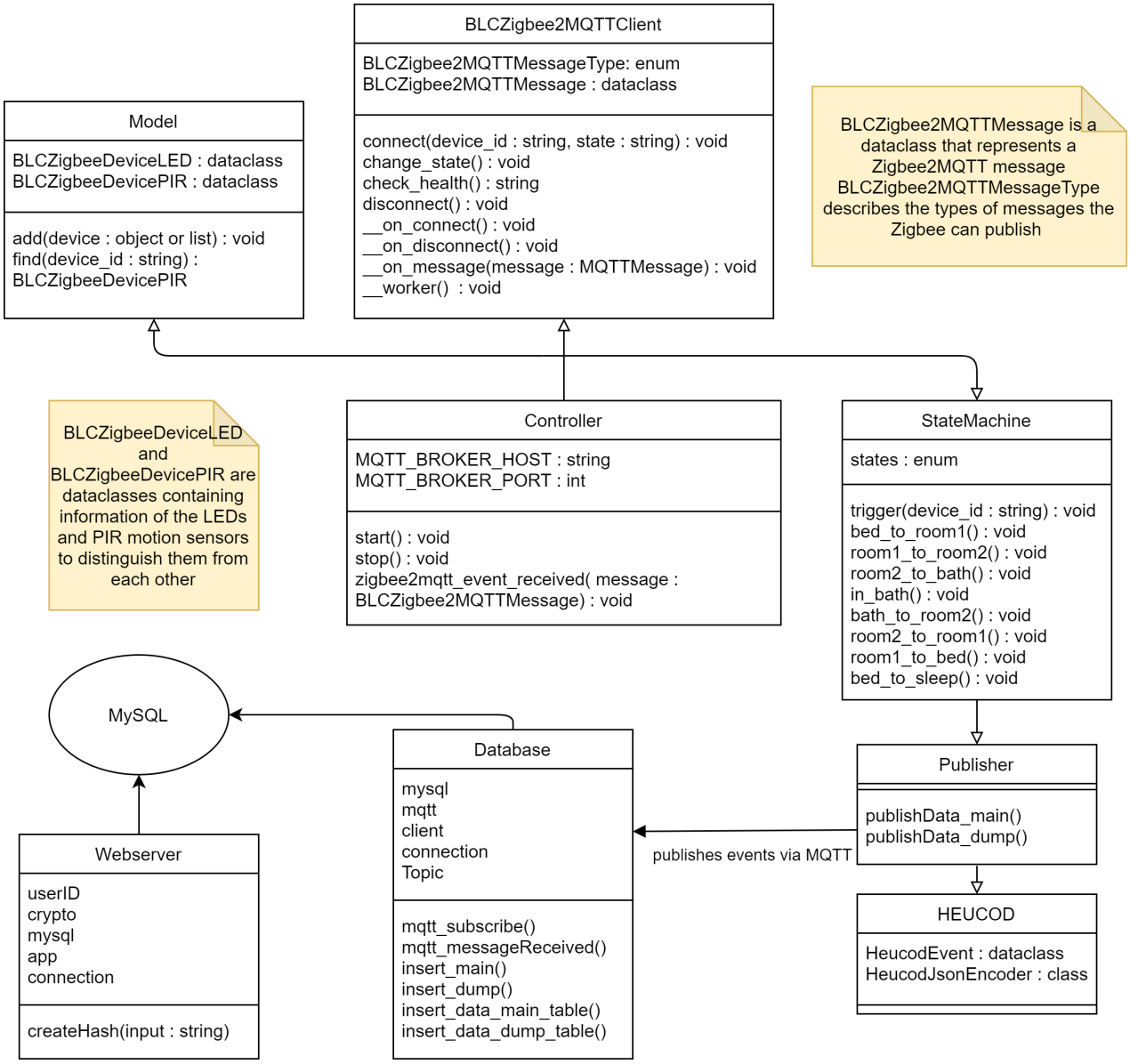

Figure 4.1: UML Class Diagram for the BLC application

State Machine

This class implements the transitions from the user in case of movement from one room to another aswell controlling the state of the LEDs in every room. Therefore, the state machine class consists of functions that describe each transition.
A illustration of the state machine is shown in figure 4.2. As seen on the state diagram each room is interpreted as a state in the system. For every transition between the states there are two conditions. The first condition is the registration of movement from the PIR motion sensor in the given room and the second condition is whether the user has been to the bathroom or not. The value of these conditions triggers events, which is a change in the state of the LED and thus determines which LEDs that turn on or off.
The state machine gets called from the controller, which knows which PIR sensors that have registered movement. More specifically, the controller calls `trigger()`, which is the function in the state machine that is responsible for changing the state of the LEDs accordingly.

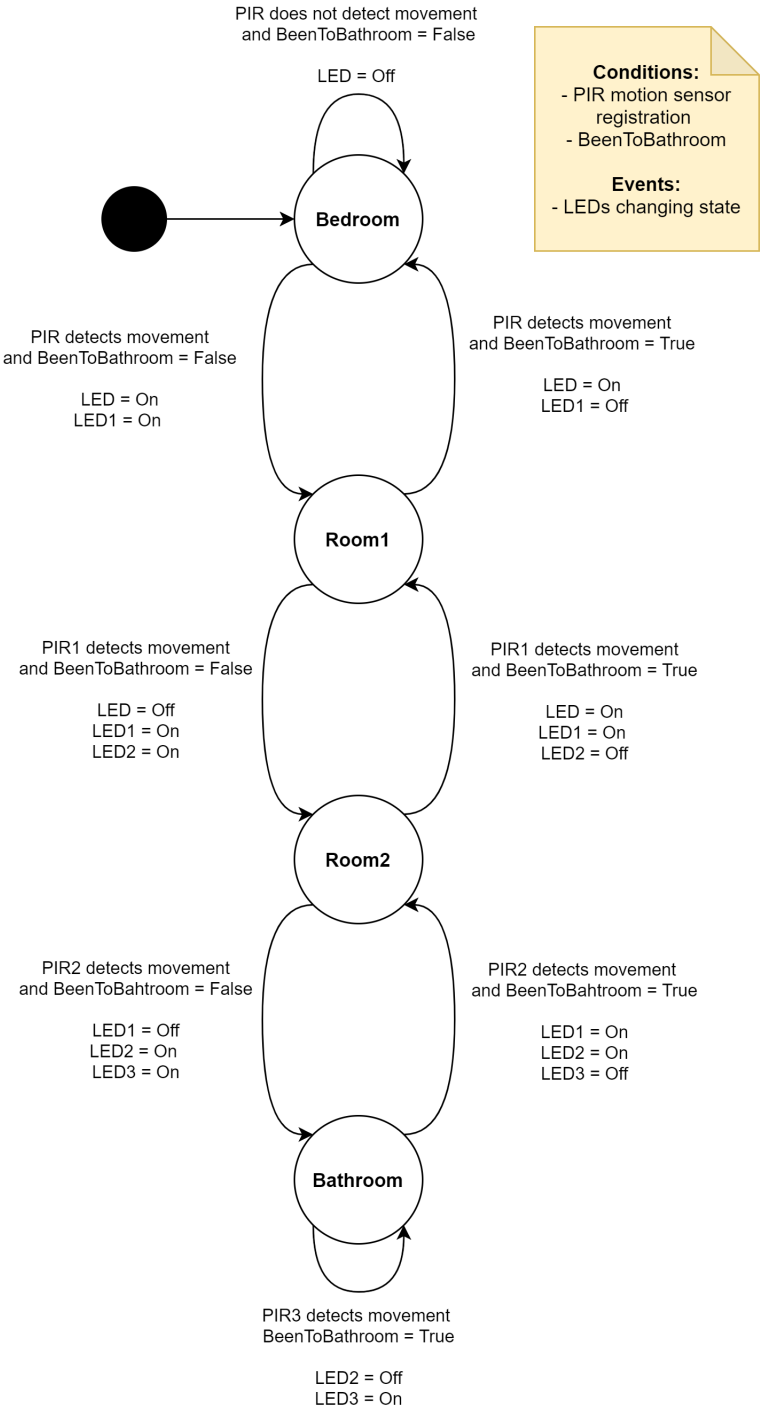

Figure 4.2: UML State Diagram

Publisher and Heucod

The publisher is responsible for publishing the data to the database in order for the user to be able to view it on the webserver. As seen on the diagram, there are two types of data that can be published, `main` and `dump`, which are called in the state machine class. This class acts as a publisher, since it communicates with the database through a MQTT broker. In order for the database to understand the messages published by the publisher, it is necessary to convert the messages to JSON objects. This is done in the HEUCOD class, which has a JSON encoder class as well as many different types of events. Thus, the publisher calls this class before publishing the data.

Webserver and Database

As mentioned previously, the project follows the MCV architectural pattern; this is also the case for the webserver. The model is located in `BLCDatabase.js` and the MySQL database. They are in control of the business logic, handling the received data and making sure it is stored in the database. The controller, located in `BLCwebsite.js`, handles the HTTP requests and the application logic, and gets data from the model through query calls to the database [10]. The two Javascript files work independently with MySQL, which is the only link between the two. Finally, the view is the HTML code, where all the files are located in the `views` folder, which is the generated page the user sees. The view is based on the HTTP requests and handled by the controller.
As stated in the requirement specification, the system uses a local instance of a MySQL as the database. In MySQL, the database is set up to consist of four tables, as depicted in figure 4.3.

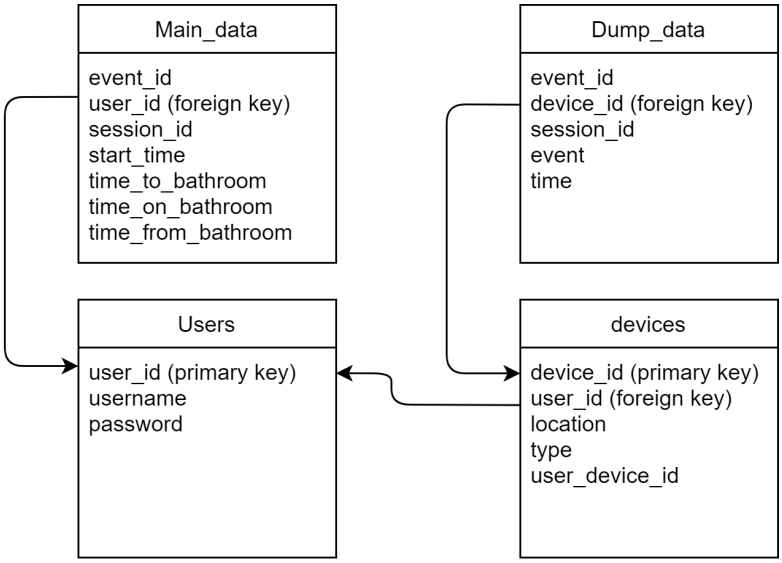

Figure 4.3: Design of BLC's database.

The database is designed to follow the normalization standard [11], making use of primary- and foreign keys to avoid storing repeated data. An example of this is the `device_id` foreign key, stored in the `Dump_Data` table. It is desired to store the location and type of a given device, but if stored in `Dump_Data` it will be repeated each time the device sends information. Instead, `Dump_Data` simply stores a reference (foreign key) to the specific device ID in `Devices`, where its location and type is stored only once and can be easily accessed.

Most of the data stored in each table is self explanatory by looking at the names, except for the `user_device_id` and `session`. The `user_device_id` serves as an abstraction for the PI, making the system more scalable. Imagine there are 200 different devices stores in the devices table, where a arbitrary Rasberry Pi has sensor 161, 162, 163, 164 and 165. If a change was made in the devices and the devices ID were changed to 163, 164, 165, 166 and 167, the IDs would have to be changed on the Pi manually, as well as all other Pi's affected. This would be undesirable and potentially very complicated if the system is scaled up. Instead, every Pi believes their sensor ID's to be 1, 2, 3, 4 and 5 instead. This is their 'believed' sensor ID, stored in the `user_device_id` column. Every combination of `user_device_id` and `user_id` is unique, by which the web server can find the specific sensor needed to present to the user. This also makes changes in the table possible, without having to change the ID's on all the different light guide systems.

The `session` value binds data from the `Dump_data` table to one specific entry in the `Main_data` table. The data stored in the `Main_data` is the data requested by the client, time to, on and from the bathroom and when the trip began. The data stored in `dump_data` is all the data recorded underway, where it is stored each time a user enters a new room. In other words, the main data is the overall trip, while the dump data is the details attached to said trip. With this, we have to connect the dump data to a single main trip, which is done with the `session` value. Lastly, the passwords stored in the `Users` table are stored using SHA256 encryption for protection of the users passwords, meaning the password stored must be encrypted when inserted.

5 Development View

This section will present the system from the developers perspective and focus on the main aspects of BLC's automated light guide, which will be showcased in the form of code snippets. The whole source code can be seen in [12].

5.1 Model

The code snippet below shows how the PIR motion sensors and LEDs are implemented in the system. The LEDs is structured such that each LED has their own ID and type. So, if there is multiple LEDs in the same room, they will have the same type but different IDs in order for the system to distinguish them from each other. The implementation of the PIR motion sensors follows the same structure as the LEDs. Furthermore, because the LED should automatically turn on in the next room in case of movement and turn off if no movement is detected, then a PIR motion sensor also knows the LEDs of the previous, current and next room.

```
@dataclass
class BLC2ZigbeeDeviceLed:
    id_: str
    type_: str

@dataclass
class BLC2ZigbeeDevicePir:
    id_: str
    type_: str
    ledPre: BLC2ZigbeeDeviceLed
    ledOwn: BLC2ZigbeeDeviceLed
    ledNext: BLC2ZigbeeDeviceLed
```

To add the devices into the system, the function `add()` is called in `main()`. The input of the function is a list of all the PIR motion sensors along with their corresponding information, where the required information is showed in the dataclass above.
The code snippet shows the adding of 4 PIR sensors and their relative LEDs:

```
devices_model = BLC2Model()
devices_model.add([BLC2ZigbeeDevicePir("PIR", "pir0", None,
BLC2ZigbeeDeviceLed("LED", "led0"), BLC2ZigbeeDeviceLed("LED1", "led1")),
BLC2ZigbeeDevicePir("PIR1", "pir1", BLC2ZigbeeDeviceLed("LED","led0"),
BLC2ZigbeeDeviceLed("LED1", "led1"), BLC2ZigbeeDeviceLed("LED2", "led2")),
BLC2ZigbeeDevicePir("PIR2", "pir2", BLC2ZigbeeDeviceLed("LED1","led1"),
BLC2ZigbeeDeviceLed("LED2", "led2"), BLC2ZigbeeDeviceLed("LED3", "led3")),
BLC2ZigbeeDevicePir("PIR3", "pir3", None, None, None)]) #Bathroom PIR
```

5.2 Controller

As stated before, the controller has the responsibility of registering and handling the incoming events from Zigbee2MQTT. This is done in `zigbee2mqtt_event_received()`. This function is given as callback to BLCZigbee2MQTTClient, which is then called when a message from Zigbee2mqtt is received.
Upon receiving the message, the controller first checks whether the current time is between 10 PM and 09 AM as this is the time interval the system is required to be valid in. Afterwards, the device ID is retrieved as well as the device itself using `find()` from the model class. When the device, which has registered movement, is found by the controller, then the controller checks for a occupancy message. If there are an occupancy message, then `trigger()` in the state machine is called with the device ID as argument. Otherwise an error is returned. Thus, only occupancy messages are accepted and forwarded to the state machine. The occupancy message can either be true or false. If true, then the PIR motion sensor has detected movement and vice versa.

```
def __zigbee2mqtt_event_received(self, message:BLC2Zigbee2mqttMessage) -> None:
    self.currentTime = datetime.datetime.now()
    if self.currentTime > self.today10pm and self.currecntTime < self.today9am:

        ...

        device_id = tokens[1]
        device = self.__devices_model.find(device_id)

    if device:
        try:
            occupancy = message.event["occupancy"]
        except KeyError:
            pass
        else:
            self.stateMachine.trigger(occupancy, device_id)
```

5.3 State Machine

The `trigger()` function implements the main logic of the state machine. It gets called from the controller, which states which PIR sensor that have registered movement. The function is implemented with an if-statement that goes through every state. The code snippet below shows the implementation for the `Bedroom` state, but the rest of the states follows the same structure. For each state it checks for occupancy and if the device ID matches with the corresponding room. Furthermore, a variable describing if the user has been to the bathroom or not is also checked. Based on the values of these variables, a suitable transition is called.

```
def trigger(self, occupancy, device_id):
    if self.state == 'Bedroom':
        print("Bedroom")
        if device_id == "PIR" and occupancy and not self.Been_to_bath:
            self.bed_to_room1()
        elif device_id == "PIR" and not occupancy and self.Been_to_bath:
            self.fun_bed_to_sleep()

    ...

    else:
        print("Fail in state machine")
```

The code below shows three of the transition functions. It is inside these functions, where the state of the LEDs are changed. The structure of each function is almost similar. The first function describes the transition from the bedroom to room 1. In this transition the state of

LED in the current and next room is changed by using the controller to call the client. Time is also tracked, which is used as data for the user as well as the occupancy message.
The second transition takes place when the user is in the bathroom. Again time is measured and data is published. Here the `Been_to_bath` variable is also set to `true`, which reverses the state machine logic.
The third transition describes when the user is back in the bedroom and thus marks the end of the session. Therefore, here is where the final time data is published to the database and the `Been_to_bath` variable is set to `false` again.
The rest of the transition functions, as seen on the UML class diagram, are implemented the same way as `fun_bed_to_room1()`.

```
def fun_bed_to_room1(self): #start
    device = self.__devices_model.find("PIR")
    self.__z2m_client.change_state(device.ledOwn.id_, "ON")
    self.__z2m_client.change_state(device.ledNext.id_, "ON")
    self.awake = datetime.now().strftime("%D %T")
    self.start_to_bath = time.time()
    self.pub.publishData_dump(1, self.sesion, 1, "Movement detected")

def fun_in_bath(self): #middle
    self.finish_to_bath = time.time()
    self.start_in_bath = time.time()
    self.Been_to_bath = True
    self.pub.publishData_dump(1, self.sesion, 4, "Movement detected")


def fun_bed_to_sleep(self): #end
    device = self.__devices_model.find("PIR")
    self.__z2m_client.change_state(device.ledOwn.id_, "OFF")
    self.__z2m_client.change_state(device.ledNext.id_, "OFF")

    to_bathroom = int(self.finish_to_bath - self.start_to_bath)
    on_bathroom = int(self.finish_in_bath - self.start_in_bath)
    from_bathroom = int(self.finish_from_bath - self.start_from_bath)

    print("to bathroom = ", to_bathroom, ", on bathroom = ", on_bathroom, " and
     from bathroom = ", from_bathroom)
    self.pub.publishData_main(1, self.session, self.awake, to_bathroom,
    on_bathroom, from_bathroom, "User went to bathroom and back to bed")
    self.sesion += 1

     self.Been_to_bath = False
```

5.4 Publisher

Every time data need to be send to the database, it gets done by calling either one of the Publisher class' two functions. The code snippet underneath shows the implementation for `publishData_main()`. The function makes an instance of a HEUCOD event in order to convert the message to a JSON object, which the database can understand. To communicate with the database, the publisher needs to know the IP address of the database and its port number, so it can to publish to the MQTT broker. In the code snippet, the IP is an example, whereas the port is the MQTT standard port.

```
class publisher:
    def publishData_main(...):
        event = HEUCOD.HeucodEvent()

    ...

```

```
        publish.single(hostname="192.168.2.204",
                       port = 1883,
                       topic=f"server/main_table",
                       payload = event.to_json() )
```

5.5 Webserver

The following subsection describes and shows how the webserver is implemented by providing NodeJS code snippets. This includes how to connect to MQTT and the MySQL database and how the website is made using Express.

5.5.1 Storing the data

The model of the webserver, `BLCDatabase.js` is responsible for getting and storing the data received from the PI. For this, it needs to subscribe to the Mosquitto broker and connect to MySQL. The following code snippet shows how this is done:

```
var mysql = require('mysql'); //Library
var mqtt = require('mqtt'); //Library
var client = mqtt.connect('mqtt:localhost:1883') //Connect to broker

var connection = mysql.createConnection({ //Connect to MySQL
    ... //host, user, password etc.
});
client.on('connect', (err) => {
  client.subscribe(Topic, mqtt_subscribe); //On connect subscribe
})
function mqtt_subscribe(err) {
    ...
};
```

First, the libraries for MySQL and MQTT are included and a connection to MySQL is made. Upon a connection to a Mosquitto broker, the MQTT client subscribes to a given topic, in this case '`server/#`'. With this up and running, the following code snippets shows what happens when a message is published on the topic:

```
function mqtt_messsageReceived(topic, message) {
  const obj = JSON.parse(message)
  ...
    if(topic === "server/main_table"){
      split_data_main(obj)
    }
    else if (topic === "server/dump_table"){
      split_data_dump(obj)
    }
  ...
};

function split_data_main (obj){
  var user_id = obj.patientId
  ...
  insert_main(user_id, ...)
}

function insert_main(user_id, ...) {
  var query = "INSERT INTO main_data (user_id, ...) VALUES (?, ...)"
  connection.query(query, [user_id, ...], (err) =>{
    ...
  });
};
```

As seen in the code snippet, the action depends on which topic the message is published on. For this example, it is published on `server/main_table`. This results in a call of the `split_data_main()` function, which splits up the JSON object into their respective values. It then goes to the `insert_main` function with all the values as input. In here, a query is created and executed on the MySQL connection, which results in data published on the MySQL database unless an error occurs.

5.5.2 Displaying the data

The display of data happens in the controller and the view. As stated earlier, the controller handles the HTTP requests, application logic and the data validation. Based on the requests the views is send to the user. For setting up the frame network and network application, the library express JS [13] has been used. The following code snippet displays how the website is set up.

```
const express = require('express')
var app = express()

app.use(session({
  ...
  cookie: {
    expires: 3600000
  }
}));

app.listen(3000, () => {
})
```

First, the express library is included and saved in the variable `app`. The following two calls sets up sessions such that the webserver is able to differentiate between logged in and non-logged in users, and make express listen on port 3000. With this code, the server is up and running.
Next, it needs to be able to get the HTTP request with `POST` and `GET` methods. An example of such an implementation is given below:

```
app.get('/', (req, res) => {
  res.render('login');
})
```

The snippet shows a `get` method for when a user accesses the main '/' page. When this is accessed, the login HTML screen will be rendered and presented to the user. The different functions for the requests vary in length and complexity, and if personal data needs to be displayed, the data is send with the render. However, this is the overall functionality they all follow. For the different views, the website has the following pages accessible to the user:

- Homepage
- Login
- Data
- Details
- Dump

Not all of these are accessible to the user, if they are not logged in. In order to check for this, sessions are used. A session is created in the login method and ends/is destroyed either when a user logs out, or if an hour goes by. This can be seen in the snippet below:

```
app.post('/login', (req, res) => {
  var username = req.body.username
  var password = createHash(req.body.password)
  connection.query(query, [username, password], (err, rows) => {
  ...
  req.session.username = rows[0].user_id
    req.session.loggedin = true;
  })
})

app.get('/logout', (req, res) => {
  req.session.destroy();
  res.redirect('/');
  })
```

In '/login', the input username and password are stored, followed by a check if a match is found in the database. If there is, the username is saved in the session and `loggedin` is set to true. When logging out, the session is simply destroyed, which sets `loggedin` to false and removes the stored username. In order to keep 'intruders' out, a check of an existing session is done. Every `POST` and `GET` function, except for '/', have a check as seen in the following code snippet:

```
if (!req.session.loggedin) res.redirect('/')
else{
    ...
}
```

Thus, if a user tries to access a page they aren't allowed to, they will be redirected to the login page.
The EJS files all contain simple HTML code along with CSS styling [14]. These will not be explained in depth.

In order to make a new user on the website, one must directly access the database and manually insert the user. This is a part of the setup process, along with manually inserting the devices being in a given system of a user. Examples of how to insert such can be found in the `README.txt` file in Github [12].

5.6 Support

For the implementation of BLC's automated light guide application, there has been used some support for different parts of the code. The following points highlights where the implementation support has been applied:

- BLC's automated light guide is required to be active in the hours 22 PM to 09 AM. Therefore, the `date.time` library has been applied in order to do so [15].
- The classes BLCZigbee2MQTTClient and Model has been has been used as the framework for our application and has been implemented with the help of tutorials made by instructor Jorge Miranda [16].
- The database and the webserver has been implemented with help from different online sources. This includes NodeJS code for a MQTT and MySQL setup, login systems, setting up sessions and more [17] [18] [19] [20] [21] [22] [23].

References

[1] Emil Hu, Bastian Kramer, Magnus Tang, Jens Fisker and Daniel Biørrith, "Requirements Specification," 2021. [Online]. Available: https://github.com/Basaron/BLC_Auto_Light_Guide/tree/main/References

[2] P. Krutchen, "Architectural Blueprints—The "4+1" View Model of Software Architecture," IEEE, Tech. Rep., 1995. [Online]. Available: https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf

[3] I. Sommerville, *Software Engineering*, 10th ed. Pearson, 2016, pp. 173–175.

[4] S. Walther, "The Evolution of MVC," 2008, accessed 21 May 2021. [Online]. Available: http://stephenwalther.com/archive/2008/08/24/the-evolution-of-mvc

[5] K. Kanters, "Zigbee network," accessed 25 March 2021. [Online]. Available: https://www.zigbee2mqtt.io/information/zigbee_network.html

[6] G. Elinoff, "Bluetooth vs Wi-Fi vs ZigBee," accessed 23 March 2021. [Online]. Available: https://www.electronicproducts.com/bluetooth-vs-wi-fi-vs-zigbee/#/

[7] S. Rogers, "'Data is' or 'Data are'," accessed 20 May 2021. [Online]. Available: https://www.theguardian.com/news/datablog/2010/jul/16/data-plural-singular

[8] S. Cope, "MQTT Publish and Subscribe Beginners Guide," accessed 25 March 2021. [Online]. Available: http://www.steves-internet-guide.com/mqtt-publish-subscribe/

[9] ——, "Using Zigbee2MQTT- A Beginners Guide," 2021, accessed 25 March 2021. [Online]. Available: https://stevessmarthomeguide.com/using-zigbee2mqtt-beginners-guide/

[10] W. schools, "HTTP calls and requests," accessed April 2021. [Online]. Available: https://www.w3schools.com/tags/ref_httpmethods.asp

[11] Guru99, "Normalization of database," accessed May 2021. [Online]. Available: https://www.guru99.com/database-normalization.html

[12] Emil Hu, Bastian Kramer, Magnus Tang, Jens Fisker & Daniel Biørrith, "BLC_-Auto_Light_Guide," 2021, accessed 21 May 2021. [Online]. Available: https://github.com/Basaron/BLC_Auto_Light_Guide

[13] O. Foundation, "Express js," accessed April 2021. [Online]. Available: https://expressjs.com/

[14] E. developers, "Embedded Javascript Templating (EJS)," accessed May 2021. [Online]. Available: https://ejs.co/

[15] R. Pate, "How to compare times in Python?" 2021, accessed 18 May 2021. [Online]. Available: https://stackoverflow.com/questions/1831410/how-to-compare-times-in-python

[16] S. R. Wagner and J. Miranda, "Tutorial 6," 2021, accessed 18 May 2021. [Online]. Available: https://github.com/Basaron/BLC_Auto_Light_Guide/tree/main/References

[17] smching, "Node JS MQTT and MySQL application," accessed April 2021. [Online]. Available: https://gist.github.com/smching/ff414e868e80a6ee2fbc8261f8aebb8f

[18] W. school, "How to select data from MySQL in Node JS," accessed April 2021. [Online]. Available: https://www.w3schools.com/nodejs/nodejs_mysql_select.asp

[19] M. Nurullah, "How to display data from MySQL in Node JS," accessed April 2021. [Online]. Available: https://codingstatus.com/how-to-display-data-from-mysql-database-table-in-node-js/

[20] T. Point, "Tutorial for implementing Express sessions," accessed April 2021. [Online]. Available: https://www.tutorialspoint.com/expressjs/expressjs_sessions.htm

[21] D. Adams, "Login system for NodeJS using MySQL," accessed April 2021. [Online]. Available: https://codeshack.io/basic-login-system-nodejs-express-mysql/

[22] Z. B. Khalid, "How to decode JSON in Node JS," accessed April 2021. [Online]. Available: https://learncodeweb.com/javascript/how-to-decode-and-encode-json-data-in-javascript/

[23] S. Exchange, "How to access data using foreign key in MySQL," accessed April 2021. [Online]. Available: https://dba.stackexchange.com/questions/129023/selecting-data-from-another-table-using-a-foreign-key