**TODO:**

**Change to formal tense (don’t use I)**

**Music-Synced Smart Lighting System**

A dissertation submitted in

partial fulfilment of  
The requirement for the degree of  
**MASTER OF SCIENCE**

in

**Software Development**

in  
**The Queen’s University of Belfast**  
By  
**Ryan Urquhart**

**40099112  
06/05/2025**

**Acknowledgements:**

* Bee-Yen Toh – My supervisor who was with me every step of the way
* Cherise McClean – My wife, a constant (and welcome) distraction
* SoDes – On GitHub, created a package that made this whole project possible on windows (PyAudioWPatch)
* aiXander – On GitHub, created a repo that I adapted, it provided fast Fourier transformations that I built the beat detection algorithm on top of.

Abstract:

This report covers the development of a Music-Synced Smart Light System. The development of this system was successful, however, due to time constraints the system is missing many aspects that would be desirable should the project be continued on. Continuing on the development of this system is aided by the initial style of development of the system, the system has been developed in such a manner as to allow easy continued development. The system takes advantage of FFT of audio data from PyAudio to allow the system to detect the beat of music and change bulb brightness in time with that beat.

Table of Contents

[Introduction: 6](#_Toc196841949)

[1. Understanding the Problem 7](#_Toc196841950)

[2 User Interface Design 9](#_Toc196841951)

[3 Architecture Design and Algorithm Explanation 14](#_Toc196841952)

[3.1 API Endpoints 14](#_Toc196841953)

[3.2 Bulb Database Model 16](#_Toc196841954)

[3.3 Bulb communication 16](#_Toc196841955)

[3.4 Host Audio Parsing 18](#_Toc196841956)

[3.5 Beat Detection Algorithm 19](#_Toc196841957)

[3.6 Future Development Mindset 21](#_Toc196841958)

[4 Testing 26](#_Toc196841959)

[4.1 Automated Testing 26](#_Toc196841960)

[4.2 Manual Testing 28](#_Toc196841961)

## Introduction:

The project discussed in this report is hosted on a repository at, <https://github.com/RyanMcClean/wizlight-music-sync>, if you have trouble accessing the repository please contact, rurquhart01@qub.ac.uk.

This report is focusing on the development of a Music-Synced Light Control System (hereafter referred to as, Bulb Bop). The report will consist of six chapters, Chapter One, Understanding the Problem, Chapter Two, User interface-design, Chapter Three, Architecture design and algorithm explanation, Chapter Four, Experimentation, Chapter Five, Testing, Chapter Six, Evaluation and Conclusion. The content of these chapters will be explained in the rest of the introduction.

Chapter One will contain an exploration of the problem and the basis on which I started undertaking the development of the software system. This will include my initial understanding of the project, based on the title and the preliminary meetings with my supervisor. It will then progress into my initial investigations about the potential solutions to the problem at hand. Finally, this chapter will end with the decided upon plan of action for development.

Chapter Two will contain the plan and the development of the user-interface. This will include the early iterations of the user-interface, and how these early iterations influenced the final design. It should be noted however, that the bulk of the work in this project was the beat detection algorithm and therefore, there was less thought put into the user-interface than there was into the back end of the system.

Chapter Three will contain the plan and the development of the architecture and algorithm design. This section will explain the architecture upon which the system can be ran, and the algorithms that allow it to run. This is the chapter in which the bulk of the work of this project will be explained.

Chapter Four will contain the experimentation that I have conducted due to this project. This will include looking at different languages that I attempted to utilise, as well as the research and experimentation that went into the development of the beat detection algorithm. Various iterations and experiments will be present in this chapter.

Chapter Five will contain the testing carried out on the project, during its development, and before the submission of the project. This will include risk assessments of the project, and a discussion of the rationale of the tests that were designed, as well as an examination of the areas of the project that were not thoroughly tested and an explanation as to why.

Chapter Six will be the final evaluation and conclusion of the project. Discussing the success or failure of the development of the system. It will evaluate different technologies utilised and how they have impacted the final system shape. There will be a reflection on my own development of the project, highlighting areas that were handled well, and areas where there could have been further development.

# Understanding the Problem

The initial description of the project is as follows,

“Develop a system that integrates music with dynamic lighting displays to create an "instant smart disco light" function or a “smart lighting solution for improved well-being.” The system should control a Wi-Fi-enabled smart light bulb, adjusting its brightness and colour in sync with music rhythms, enhancing the atmosphere in any setting.”

As well as this, there were four main functional criteria for the development of the system.

1. The system should connect to and control a Wi-Fi enabled smart light bulb
2. The system should detect music rhythm and tempo from an audio source and synchronise the light settings accordingly
3. The system should provide a primary control interface
4. The system should offer multiple lighting modes

There was no restriction placed on the technologies to be utilised in this project. No specific bulbs, programming languages, or web frameworks. However, due to having to use third-party hardware (The smart bulbs) there may be a specific API dictated by the manufacturer that would have to be used.

The initial prompt has no definition of the user the application is supposed to be designed for. However, during meetings with my supervisor the intended user was defined in a varied way. From families, to DJ’s, to people hosting a party. The decided goal was that the user would be someone hosting a party in their house, this user would be either using a microphone attached to the computer running the system, or the computer running the system would be playing the music from its speakers.

From the initial description of the project, research was carried out, this research covered two main areas. The first was the bulb, which brand would be appropriate for the project. The second was the programming language, this programming language had to fulfil two criteria, it needed access to the audio from the operating system I was developing on (Windows and Linux), it also needed to be able to host a web server which would provide the user interface.

The research into the bulbs brought up many different options for use, Wiz, TP-Link Kasa, TP-Link Tapo, YeeLight, Shelly, Philips Hue, Athom.tech, and switchbot. From all of these bulbs I decided to use Wiz bulbs. The reasons for this were the cost, the open API (*bulbs like TP-Link Tapo have closed API’s so users must use their app*), the app does not require a hub to operate (*Philips Hue does*), and from my reading of various forums wiz was highly recommended as a bulb brand. Based on this research I purchased a WizBulb A60.E27 bulb (A colour changing bulb), I was also later give a non-colour changing bulb as a gift.

The research into the programming language to use initially started with Java, the reason for this simply being that much of the CSC7063 Computer Programming course has been taught in Java. Two libraries I looked at were TarsosDSP(Six, 2011/2025), and Jipes by Tagtraum(Schreiber, 2013/2024), both of which I was unable to gain access to audio data from the host machine using. Jipes was unsuited as it was not made for real-time processing of audio information. With regards to TarsosDSP, it used Gradle to build it, this was something that, given the time I needed to get the project started in, I was unable to figure out in time.

As such, I looked for another programming language that would allow me access to this host audio data. For this I turned to python, there is a python package names, PyAudio, and it reliably and easily allows developers access to the host machine audio data. The issue however, is that PyAudio does not allow access to the loopback api present on windows machines that allows the developer access to real-time speaker audio data. This was solved when I found a windows specific patch of the PyAudio library, called PyAudioWPatch. This allows PyAudio the ability to access the loopback api, this with the Django library that is able to serve web apps with a great deal of flexibility made python the perfect choice.

A big consideration to take into account was the security of the system. In the case of this project there is little-to-no system security, while it was considered to add in security it was decided against for the following reasons. There are no personal details saved in the system, nor would the addition of security, like a login page, or OAuth authentication actually protect the bulbs from being manipulated by someone who knows how to do so (as the bulbs themselves have no security on the local network). The only security for this system comes externally to the system, and comes in the form of the firewall on the local network.

# User Interface Design

A screenshot of a computer

AI-generated content may be incorrect.

Figure 1 - Index Page (PC View)

The user interface for this system is quite simple. There are a total of four different views that the user can access. Each of these views are dynamic to varying degrees, Django gives the option to develop html that will dynamically change based on a context provided by the developer. For instance, the bulbs on the index page render depending on if they are present in the database or not.

The first part of the user interface that we will discuss are the parts common to all views. The main part of this is the navbar, this contains the title, links to other views, buttons to activate functions of the system, and a light/dark mode toggle button. There is also a footer on the page, a line that contains the text, ‘2023-25 : Bulb Bop : Ryan Urquhart : 40099112’. Part of these aspects are dynamic, the audio sync option for instance, only shows when bulbs are recorded in the database.

The second view we are going to look at is the home page, the index page (figure 1 and 2), this is the main page that the user will be engaging with. This page displays icons for the bulbs that are recorded in the database, their names, as well as graphically displaying their state. There are tooltips that appear when the user hovers their mouse over the bulbs, this informs the user that should they click on the it will toggle the state of the bulb. Should the user not yet have connected the system with any bulbs then the index page will display a prompt to the user to use the discover function of the system (figure 3).

A black and white image of a person standing in front of a white sky

AI-generated content may be incorrect.

Figure 2 - Index View (Mobile View)

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3 - User prompts to add bulbs

A screenshot of a computer

AI-generated content may be incorrect.

Figure 4 - FAQ's Page (PC View)

The third aspect of the user interface we will discuss is the ‘FAQ’ (frequently asked questions) page (figure 4). This page covers some basic setup questions that a user may have, from connecting lights to troubleshooting, and a contact email for support (not a real email). This page is referenced in the tooltip to help new users get started with the application. It also links to the WizBulb app, to show users how to connect their bulbs to the local network (unfortunately using the system to connect the bulbs to the local network was impossible as it requires Bluetooth and is undocumented).

A screenshot of a computer

AI-generated content may be incorrect.

Figure 5 - About View (PC View)

The fourth page of the system to discuss is the ‘About’ page. This is a simple page that just displays information about the project, it’s goals, and some information about the author if the project. Like every other page, this page has the navigation bar which allows the user, should they have bulbs saved in the database, to activate the audio sync. As this page is mostly static, apart from the navigation bar, there is nothing else to say about it.

The final page to discuss in this section is the ‘Edit’ page, for the bulbs. This page allows the user to edit or delete the bulbs that they have saved in the database. The user can edit the bulbs name and Ip address, but that is all. As when the system has a bulb saved with a name and an Ip address, then it will query the bulb and update its status in the database itself. When the user clicks on edit, it will show a modal that the user can interact with to edit the bulb. When the user clicks on save edit, or delete, the page will display a success or error message in the bottom right corner.



Figure 6 - Edit View with two bulbs

A red sign with white text

AI-generated content may be incorrect.

Figure 7 - Example Error Message

A green screen with white text

AI-generated content may be incorrect.

Figure 8 - Example Success Message

# Architecture Design and Algorithm Explanation

In this section of the report the discussion will centre on the ‘backend’ of the system. This will include a description of the api endpoints, the bulb database model, how the system was configured to interface with bulbs on the network, a description of how access was gained to the audio data of the host machine, an explanation of the algorithm to detect beats in music, and finally a section on how the system was produced so with future development in mind. This section will encompass most of the work done on this project and therefore will be the longest section in this report.

# EXPAND ON SECURITY HERE #

## API Endpoints

This project has 14 different API endpoints, not including requests that get images and JavaScript code. Four of these endpoints are for different views or pages, of the user interface, one is reused for bulb form submission, therefore, these will not be discussed in this section, as the views were mentioned in chapter 2. There are three endpoints for bulb control, four (including the reused view endpoint) for database operations and, two for error and success handling.



Figure 9 - List of API endpoints

The endpoints for bulb control are as follows, “/discover/”, “/toggleBulb/”, “/queryBulb/”, and “/colorBulb/”. Of these four, only three are currently utilised in the system, “/colorBulb/” was developed for use, but when testing with a colour changing bulb, and a non-colour changing bulb this endpoint caused issues, so was left for future development, which ultimately ended up being scoped out of the project. “/discover/” is used to discover new bulbs on the network, it sends a broadcast UDP packet, to which bulbs connected to the network respond with their current state. This response is shown to the user, who can give the bulb a name, which is then sent to another endpoint, which will be mentioned later. “/toggleBulb/” is the main endpoint the user will trigger from the user interface, when they click on the icon of a bulb it will trigger this endpoint, which sends a packet to the specific bulb and instructs it to turn off, or on, depending on the state of the bulb recorded in the system. “/queryBulb/’ is similar to “/discover/”, except rather than sending a broadcast packet to all Ip addresses, it send packets to specific addresses to query the state of specific bulbs.

The endpoints that control database operations are, “/”, “/delete/<str:ip>/”, and “/edit/<str:ip>”. The “/” endpoint is the index endpoint, when this endpoint is triggered with a GET request, then it renders the home page, when it’s triggered with a POST request, then it submits a Django model form. This Django model form is how the system saves new bulbs in the database. When the form is submitted, it is checked to ensure that it contains no errors. The criteria for this check is that the Ip address is valid, and that the name of the bulb is not null. The other information for the bulb is filled in from the return from the broadcast packet that was sent during discovery. The next two endpoints are accessible to the user through the edit bulbs page, “/delete/<str:ip>/” and “/edit/<str:ip>/” both of these endpoints require the Ip address of the bulb in which they are editing, as a unique identifier. “/delete/<str:ip>/” will delete the bulb with the specified Ip address, and “/edit/<str:ip>/” will submit a form, that is verified, of a new Ip address or name for the bulb, this then updates the model in the database for that bulb.

There are two endpoints that control error and success handling, these clear the error and success messages, both are to prevent error, and success messages being displayed multiple times. These endpoints are never interacted with by the user manually, they are triggered through JavaScript that runs when every page loads. While this does not interfere with a message that is being displayed in the page that has just been loaded, it will prevent a message from being displayed in another page that the user navigates to.

There are two endpoints for controlling the audio synchronisation, “/activateSync/”, and “/stopsync/”. These are self-explanatory, “/activateSync/”, this endpoint accepts a POST request that contains the index number of the audio device selected by the user. This then activates the audio sync in the system for that audio device, the system waits thirty seconds to ensure that the sync program has activated, then it sends a status message to the user interface.

This concludes the endpoints of the API that are activated by the user, or the JavaScript the pages contain. The next step of this chapter is to discuss how communication with the bulbs was achieved by the system, documentation that was followed, and the technologies that were involved in its process.

## Bulb Database Model

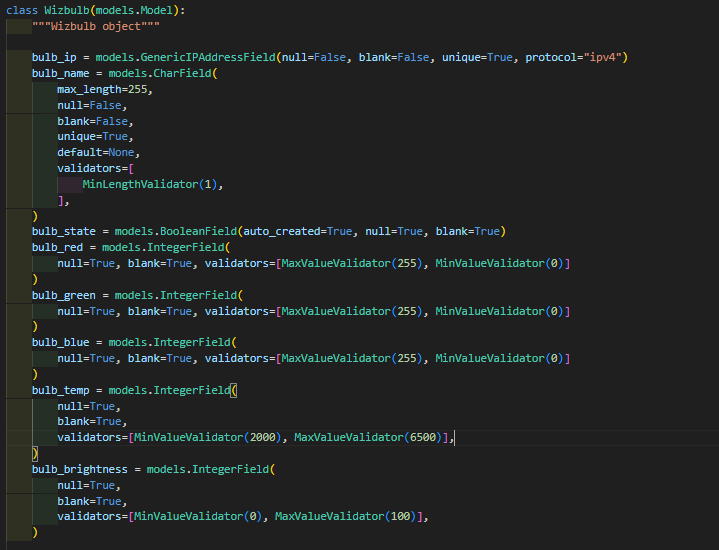


Figure 10 - Bulb Database Django Model

The database design in this project is rather simple, it only contains one model, the bulb. This model consists of seven fields, the IP address, name, state, red, green, blue, temperature, and brightness. While all of these fields are updated when a bulb is queried, only the IP address, the name, and the state are used in the system. For each field there are several Django validators applied, these prevent bulbs from being created unless the criteria are met. For example, the IP address must be unique, it cannot be null or blank, and it must be a valid IPV4 address.

## Bulb communication

One of the reasons that the Wiz brand of bulbs was chosen for this project was the level of documentation about the bulb (*WiZ Pro API Reference*, n.d.), to augment this, there is also a python package called, ‘pywizlight’ which utilises this documentation to provide communication to wiz devices (Traub, 2020/2025). I did not use this package, and instead primarily relied on the wiz documentation to enable communication to the bulbs.

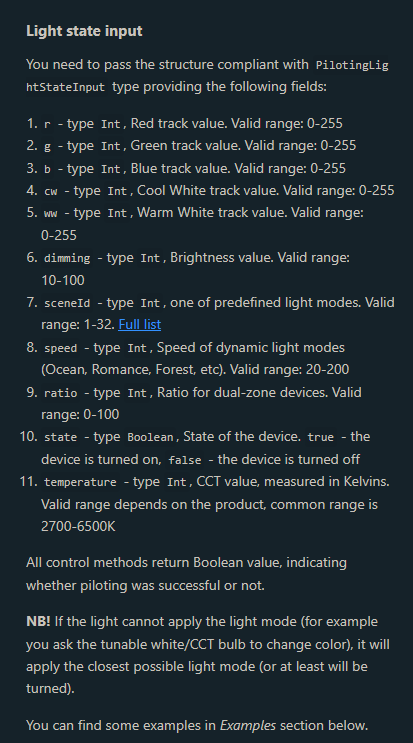


Figure 11 - Excerpt from API Reference Site

Utilising the API reference, I have implemented five UDP packets which are used by the system, I have also included two other UDP packets, which are not currently used, but can be implemented in further development. The seven total packets have the following functions, bulb discovery, turn on bulb, turn off bulb, turn to half brightness, turn to full brightness, bulb registration, and turn bulb to colour. Bulb registration and turn bulb to colour are unused by the system, while both were used in the development of the system, it was unfeasible to implement them properly in the time given for the project.

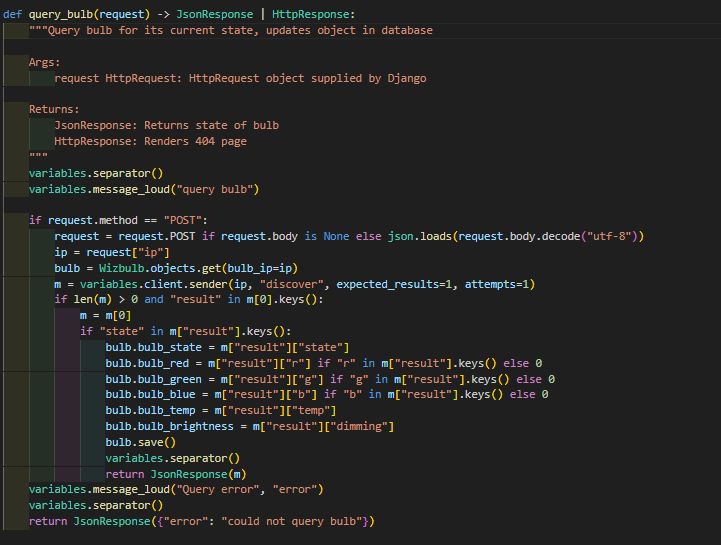


Figure 12 - Query Bulb Fuction that parses the bulb packet return

Functions that can read and parse the return UDP packets from the bulbs have also been implemented. These packets contain information that allows the system to update the bulb object and database model with their current state. This allows the index view for instance to change the colour of the bulb icons depending on the current state of the bulbs.

## Host Audio Parsing

The reason that Python was used over Java was that Python has easy access to the audio of the host machine. Initially only PyAudio was used, and the system was able to access microphones attached to the host machine, however, then PyAudioWPatch was discovered. PyAudioWPatch takes advantage of Windows Audio Session API (WASAPI) (drewbatgit, 2021), this allows the audio of output devices to be intercepted and parsed by the library. With the discovery of PyAudioWPatch I was then able to allow the system to parse music played from the system and control the lights accordingly, this is a considerable improvement over having to use the system with a microphone which would have picked up all stray noises, making the beat detection a much harder task (using a microphone is still as possible as using a speaker is). On Linux PyAudioWPatch does not work, as WASAPI is a windows only API, however, using PulseAudio on a Linux machine, it is easy to create a loopback of a speaker so that it registers as an input device.

So due to this I had access to the raw audio output of a selected microphone or speaker of the host machine. This however, was unhelpful when it comes to detecting beats, as from PyAudio (or PyAudioWPatch) I am just getting a stream of un-processed data. This un-processed data needed to be sorted into frequencies and amplitudes, necessitating the use of Fast Fourier Transformations. This however, is a complicated process, so to handle this for me, I found a repository on GitHub that would handle this calculation for me, Realtime\_PyAudio\_FFT (*aiXander/Realtime\_PyAudio\_FFT: Realtime Audio Analysis in Python, Using PyAudio and Numpy to Extract and Visualize FFT Features from Streaming Audio.*, n.d.). The repository itself was helpful; however, it was not designed for my needs, this meant that I had to modify it. This meant making it work with PyAudioWPatch, but it had to still work with PyAudio (to allow it to work on Windows and on Linux).

A screenshot of a computer

AI-generated content may be incorrect.

Figure 13 - List of commits to submodule

To manage this edit, I changed it so that rather than visualising the audio output the repository would instead just return the stream of frequencies and amplitudes that are being produced by the audio device. I also changed the repository so that it would use the ‘PyAudioWPatch’ library, if possible, but would fallback to ‘PyAudio’ if ‘PyAudioWPatch’ is not available (i.e. on Linux rather than windows). Another change that I made, was the editing of channels and audio rate, this was necessary to allow the use of speakers with the system. This final change was necessary, as speakers often use two channels, for left and right, they also tend to use a higher rate for the audio than microphones do.

With the libraries and changes made above once the user selects an audio device and clicks to start the synchronisation of music, then the system will start a stream of amplitudes and frequencies. This stream is then parsed by the system and using the beat detection algorithm will send UDP packets to the bulb(s) to turn them up, or down, in brightness in time with each other and the music. This beat detection algorithm is what will be discussed in the final section in this chapter.

## Beat Detection Algorithm

Now the system is getting a stream of computed frequencies and amplitudes from the audio device, as the final goal of the project is to have the beat of the music be what influences the control of the bulbs, the audio data will need to be manipulated to detect the beat. From my research this is an extremely difficult task, a lot of beat detection does not happen in real-time, rather it observes an audio file in it’s entirety and detects the beat that way (‘Beat Detection’, 2021). Therefore, I looked at an example of beat detection that is used in real-time (*Beat Detection Algorithm – Parallelcube*, 2018), this example comes with formulas that I was able to adapt to serve the purposes of the system.

When it comes to the system parsing the audio, the start was to figure out the frequencies that needed attention, and which ones could be ignored. I ultimately decided that all frequencies over one hundred and fifty hertz were to be ignored. The higher frequencies contain vocals and other instruments that are not indicative of a beat in the melody. The lower frequencies, however, are where most of the drum set is contained, this is where the beat will be most noticeable, and therefore these are the frequencies that the system will be using in the beat detection. The rest of this section will cover the input of frequencies and amplitudes and how it is manipulated to define a threshold for beat detection.

The above equation is how the system gets the sum of the amplitudes (E) for a particular set of frequencies (i to k+n). This is achieved by calculating the sum of the FFT amplitudes (FFT[i]) where ‘i’ is the frequency. This is done for the range i=k, i.e. the starting frequency of the sum, and k+n, where k+n is the max frequency of the sum. In the system this is done for three ranges of frequencies, where n always equals 50, but i is equal to, 0, 50, and 100. This leaves the system with a sum of amplitudes, from 0hz to 49hz, 50hz to 99hz, 100hz, to 149hz. Every frequency above 150hz is ignored.

This sum of amplitudes is then added to an array, one amplitude for each unit time. The unit of time in this case is dependent on how fast the host machine can compute the amplitudes of the audio data which it is manipulating. This means that an average can be calculated for each array of amplitudes, this is done as below.

This average can then be used to determine the fraction of the average compared to the sum of the amplitudes for that frequency range, this fraction can be considered the variance. The sum of these variances for the length of ‘buffersize’ can then be considered to be ‘F’.

This sum can then be used to calculate the average variance (V), however, to ensure that the average variance is greater than the one (this is needed so that the beat limit is greater than the average frequency range amplitude), when calculating the average variance, if the variance is less than one, then the inverse of the variance is taken.

Now that the average variance (V) has been calculated, this can be used to calculate a threshold value, this is the threshold which if the sum of amplitudes for a frequency is greater than, then it will be considered a beat. The equation to calculate the threshold is as follows.

Upon the detection of a beat by the explained algorithm above, the system will then send a UDP packet to all bulbs connected to the system and it will change the brightness of the bulbs appropriately.

## Future Development Mindset

Part of the requirements of this project was to ensure that the project was left in a position that would be conducive to future development. This means that the code is clean, organised, and written in such a way that should another developer with no experience of the system (but experience in the language and framework), pick up this project where it has been left off, they would have no difficulty understanding the development choices and code. To go beyond, this project also includes organisation of the GitHub repository that it is hosted on (McClean, 2024/2025).

With regards to clean code, this means that the system is organised into directories, each of which is named to clearly label the contents. For example, the templates folder holds all the Django template files that are used to render the webpages.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 14 - Display of Directory structure

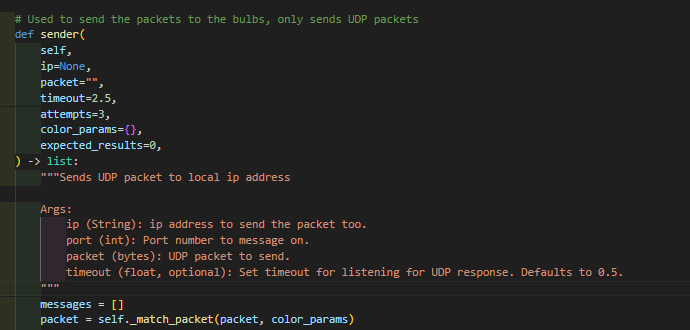


Figure 15 - Example of docstring’s and comments

While this is helpful, it is not enough if the content of these files is hard to read or understand. To combat this deficiency, two approaches have been taken. One, the code base is automatically formatted by one of two formatters. For python files, the code is formatted by the “Black” formatter (*Black*, 2025), if the file is an ‘html’ file, or a ‘JavaScript’ file, then it will be formatted by the ‘Prettier’ formatter (*Prettier · Opinionated Code Formatter · Prettier*, 2025). The second approach is the manual addition of comments, docstrings, and helper files.

The addition of formatters into the project means, that whoever is adding code into the project, the style will be consistent. ‘Prettier’ formatter, is manually configured with settings that will maintain consistent ‘html’ and ‘JavaScript’ formatting. The ‘Black’ formatter maintains python files in a format that is consistent with Pythons PEP 8 guidelines (*PEP 8 – Style Guide for Python Code | Peps.Python.Org*, 2025).

The project also makes use of linting, Pylint is used to provide this linting (*Pylint 3.3.6 Documentation*, 2025). This linting provides a score out of ten for how closely the code follows the PEP8 standards. This score is displayed on the README on the GitHub page.

The focus of the PEP8 style guide from Python is on readability. This is also the intended purpose of the addition of comments, docstrings, and helper files. Each module, function, class, and method have docstrings present on them, these simply explain the intended purpose of the following code.

There are specific parts of the system that are written but not in use. One example of this is a function that produces a packet that can turn a WizBulb to a specific colour. This is something that was hoped to be included in the final project, however given time constraints this was not implemented. These such sections of code were left in place to allow for their use without further research in the future. These sections contain comments to ensure that they are not misunderstood.

The GitHub repository for this project contains extensive organisational rules, guidelines, and continuous integration (CI) to ensure it maintains a consistent order for future development. Some of these rules have had to be ignored for the initial development of the system. Others, like the CI for instance were extremely helpful in catching bugs and maintaining code style.

This includes branch rules, to prevent anyone pushing git commits to the main branch without the pull request getting reviewed by another coder. For future and larger development teams this is integral to preventing the addition of ‘bad’ code to protected branches. With regards to pull requests I have also implemented a pull request template, and the addition of various labels to help organise pull requests and issues.

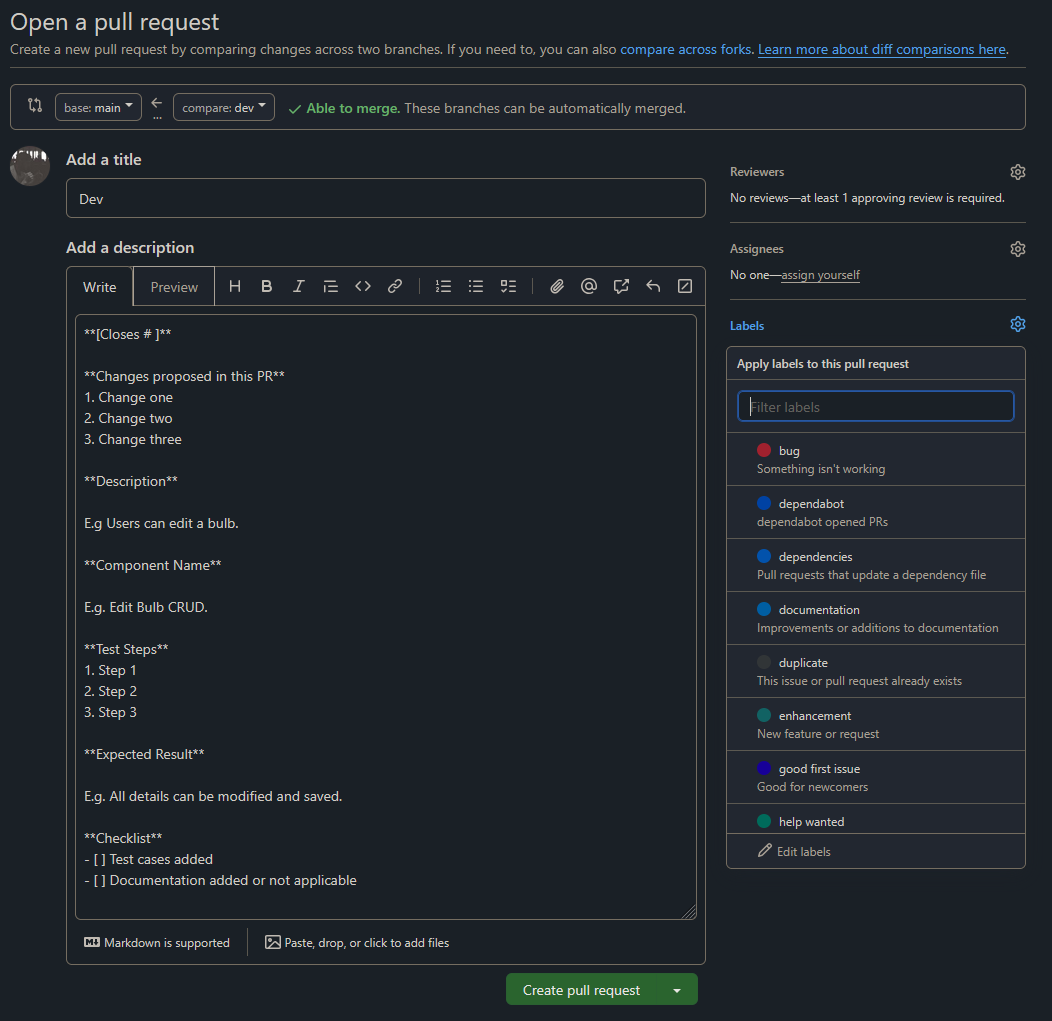


Figure 16 - Example of Pull Request Template

The most helpful feature of GitHub was the CI, this allowed me to test commits and pull requests made to the GitHub repository automatically. Using CI the repository was able to run the entire testing suite on each pull request, and scheduled each night at midnight, this then sends email notifications should a CI run fail. On top of that, the CI is able to handle other automations that are helpful to maintaining an awareness of the state of the code in the repository, such as the Pylint badge, the coverage badge, and a badge that displays the status of the last CI run.

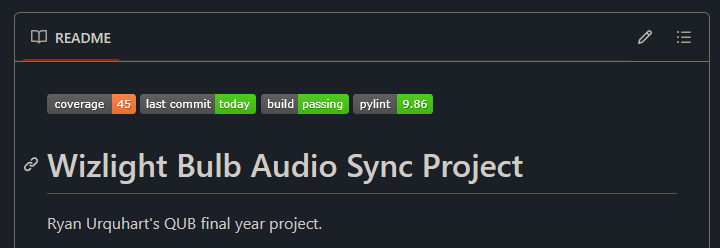


Figure 17 - Screenshot of GitHub buttons

# Testing

In this chapter the testing that has been carried out on the system will be discussed. This will cover two main sections, automated testing, and manual testing. Test cases for both will be discussed, their results, future testing that could take place, and weaknesses of the testing procedure will also be discussed.

## Automated Testing

Most of the testing that has been carried out on the system was automated, this is due to the CI on GitHub that allowed the automated tests to be carried out again and again. The automated testing was extremely helpful as it allowed for the ability to check on the status of code changes before committing to the repository. The development of the tests focused on the ability of the user to start the audio synchronisation, as this was the main objective of the project. This was also the main objective of the manual testing on the project, that will be discussed in the next section.

The automated testing in this project takes on three shapes, directly testing the code, its function, and the database, running a mock live server and testing that the UI functions as expected, and finally testing the API endpoints to ensure that the code functions as expected. With these three approaches, it is possible to cover most of the users experience with the system, as well as many of the functions that the user does not interact directly with. There are however, some functionality that is not tested in this approach, this can be due to several reasons. Code that is too complex, code that can only be ran on a live and active (has bulbs connected and live) system, or simulating the inputs is too complex, for example, simulating the input of an audio device into the system is too complex for the scope of this project.

To discuss the testing the format will follow the user journey to starting the audio synchronisation. The steps this includes are as follows:

1. Install the system
2. Load the index view
3. Discover bulbs
4. Save bulbs
5. Select an audio device
6. Start the audio synchronisation

There are other actions that the user can take, these are also tested, such as loading the FAQ view, or editing a bulb, these are also tested for, and the documentation for these can be found in the attached appendix.

There are two ways that the system can be installed from scratch, one is manual, one is automated. The automated way to install the system from scratch is via GitHub actions. The actions are configured to pull the GitHub repository and then to build it and run tests against it. Each time the GitHub actions run the system is installed on two separate windows virtual hosts, and two separate ubuntu virtual hosts, one of each of these sets is ran with Python 3.11 and the other with Python 3.12. This ensures that the system remains installable, but also that it works with Python 3.11, Python 3.12, Windows, and Ubuntu.

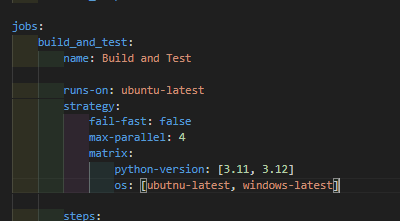


Figure 18 - Configuration of systems to run GitHub actions on

This test has been incredibly helpful in discovering bugs in the system, for instance, there was a bug that when the system was installed it would try and access the Wizbulb database model before the table was created. This caused an error that would prevent installation of the system, this bug was discovered by a run of the GitHub actions that failed. The system on the development host machine missed this bug as the database was already created on the machine.

For tests that involve loading and ensuring that the page has been loaded correctly Playwright was used to run these tests. Django has a feature that means it can load a live server with an empty database, this allows for Playwright to load the views of the system and ensure that elements are loaded and in the appropriate state (i.e. visible, attaches, etc). This is more valuable than just simply checking that the correct template was rendered, as it more accurately checks the users experience of the user interface.

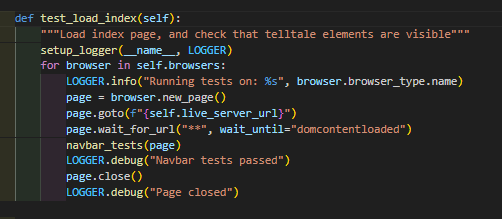


Figure 19 - Playwright load index test

The Playwright testing allows for some small compatibility testing as well. The tests can be run on multiple kinds of browsers, they are configured to run on Firefox, and Chrome. Due to this testing, it is proven that the system is not only cross OS compatible, but it is cross-browser compatible as well. There are a series of these tests that run, testing each view in turn, as well as navigation between views. Those not discussed here are documented in the appendix.

For the user to discover bulbs they must click the ‘Find Bulbs’ button on any of the views on Bulb Bop. This navigates the user to the ‘/discover/’ view. This page is almost identical to the index view; however, Django has templates that allow the HTML files to change dynamically based on the context. Navigating to the discover page changes this context and now the page will display un-saved bulbs connected to the local network on the index view.

There is a difficulty in automated testing of the bulb discovery function of Bulb Bop. When running these tests on GitHub runners, which are virtual machines with virtual networks, there are no bulbs that can be discovered. This means that the discovery function of Bulb Bop is unable to be fully tested by automated testing as it would fail when the bulbs were not discovered. This is something that must be handled via manual testing, however, successful navigation to the discover page can be tested as below (Figure 20).

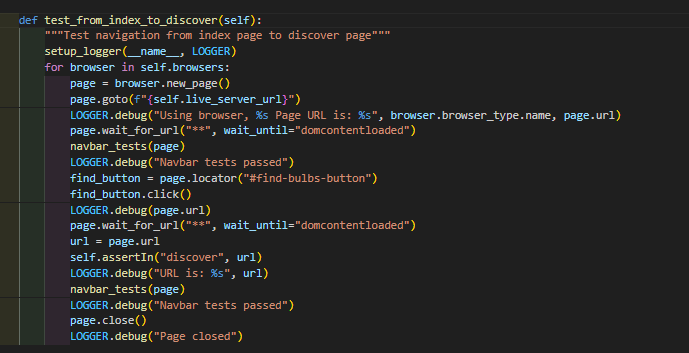


Figure 20 - Test Navigation from Index to Discover Views

The final three sections of the user journey to starting the audio synchronisation need manual testing to fully test. There is some small amount of testing that can be carried out via automated testing. Such as ensuring forms submitted properly or improperly are accepted or rejected as appropriate or testing the API endpoint to start the audio synchronisation. These are helpful but not complete until the manual testing is finished.

## Manual Testing

To be able to completely test the user journey of Bulb Bop manual testing is needed. These covers testing the ability to discover bulbs, to toggle bulbs, to start the audio synchronisation, and to stop the audio synchronisation. There are many functions of Bulb Bop that are covered by the manual tests, the ones discussed here will cover the user journey for the main functionality, other tests not discussed here will be available in the appendix.



Figure 21 - Discover Bulbs Manual Test

The above test covers the first step in the user journey to activating the audio sync, discovering the bulbs, this can only be done manually because the automated tests don’t always run on a network that has bulbs connected, and therefore, the tests cannot be configured to expect them to be present. Documentation for the tests will all follow the format of Figure 21 and will be present in the appendix.

Figure 22 - Steps for saving a bulb to Bulb Bop

The next step of the user story is the ability to save bulbs, this has been tested successfully multiple times. The next step is then to test the ability to select an audio device, this step had to be done manually because when using GitHub runners they are virtual machines which have no access to an audio device, therefore the dropdown menu in Bulb Bop has nothing to be populated with.

**SCHOOL OF ELECTRONICS, ELECTRICAL ENGINEERING and COMPUTER SCIENCE**

**CSC7058 – INDIVIDUAL SOFTWARE DEVELOPMENT PROJECT**

A signed and completed cover sheet must accompany the submission of the Individual Software Development dissertation submitted for assessment.

Work submitted without a cover sheet will **NOT** be marked.

**Declaration of Academic Integrity**

Before signing the declaration below please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.
6. Software and files are submitted via Canvas.

**I certify that that the submission is my own work, all sources are correctly attributed, and the contribution of any AI technologies is fully acknowledged.**

**I declare that I have read both the University and the School of Electronics, Electrical Engineering and Computer Science guidelines on plagiarism - http://www.qub.ac.uk/schools/eeecs/Education/StudentStudyInformation/Plagiarism/ - and that the attached submission is my own original work. No part of it has been submitted for any other assignment and I have acknowledged in my notes and bibliography all written and electronic sources used.**

**I am aware of the disciplinary consequences of failing to abide and follow the School and Queen’s University Regulations on Plagiarism.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name: (BLOCK CAPITALS)** |  | | |
| **Student Number:** |  | | |
|  |  |  |  |
| *Student’s signature* |  | *Date of submission* |  |