# [Title of Dissertation]

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

John Young

[SUBMISSION DATE]

NOTES FOR DISSERTATION WRITE UP

* Clear, concise, fluent, well organised.
* Quality over quantity
* All figures, tables, and appendices should be given numbers and headings
* Do not speak in first person
* State what was **done** rather than what I did.

REQUIREMENTS

## 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 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: |  | | |
|  |  |  |  |

## Abstract

100 words max

As technology continues to power the modern world, it is becoming more and more integrated into individuals lives. So much so that it is not uncommon to discuss ‘*smart’* fridges that inform its owner of items that are passing their use-by dates. Consumer demand for ‘*smart*’ devices has resulted in many brands entering the marketplace with their offerings – many to lock consumers into their ‘ecosystem’. Due to a lack of open communication standards for these devices, brands are creating their own, making it challenging to have devices from multiple brands ‘just work’ together. The software developed within offers a solution to both problems.

## Acknowledgements

## Contents

[[Title of Dissertation] 1](#_Toc80858459)

[Declaration of Academic Integrity 3](#_Toc80858460)

[Abstract 4](#_Toc80858461)

[Acknowledgements 5](#_Toc80858462)

[Contents 6](#_Toc80858463)

[Chapter 1: Understanding the Problem 7](#_Toc80858464)

[Chapter 2: User interface design 8](#_Toc80858465)

[Chapter 3: Architecture design and algorithm explanation 9](#_Toc80858466)

[Chapter 4: Experimentation (if relevant) 11](#_Toc80858467)

[Chapter 5: Testing 13](#_Toc80858468)

[Chapter 6: Evaluation and Conclusion 14](#_Toc80858469)

[References/Bibliography 15](#_Toc80858470)

[Appendices 16](#_Toc80858471)

## Chapter 1: Understanding the Problem

The student must clearly describe the perceived problem and the target audience. It should be evident from the chapter that the student has a thorough understanding of the problem domain and current applications used (if available) to address the problem at present. This section should demonstrate a good understanding of possible languages, libraries and frameworks that could be used to develop the project. This chapter should explain and justify the process by which the requirements have been elicited. It should then also clearly identify the requirements of your project, which can be later tested. Depending on the chosen development strategy this chapter may be written retrospectively. The student’s academic supervisor will provide additional advice on this where required. The development strategy must be clearly described, adequately justified in terms of the problem and appropriate to the project.

### Background / Current Landscape

Technology has continued to seep into every facet of our lives. Since the first ‘real’ smartphone, the iPhone, came on the scene in 2007, there has been a relentless pursuit to try to ‘optimise’ our lives. This lifestyle optimisation is happening using electronic devices to capture data on our health, daily patterns, environment, any many other attributes of our lives. The goal of which is to ultimately *try* to make our lives easier by using the data captured to provide data driven insights into ourselves. This is an area of massive year-on-year growth, as devices become smaller, cheaper, and more capable, but with this growth comes a myriad of brands, each bringing their own ‘better than the competition’ protocols. Which brings us to the crux of the problem with this innovative industry – the lack of agreed open standards, interoperability between brands and devices, and the requirement to lock yourself in to a particular brand if you want your devices to ‘just work’.

It’s understandable that brands want consumers to be wedded to their ecosystem, ranging from smartphones to smart speakers, to internet of things (IoTs) devices. The problem with this is that it restricts consumer choice and limits the potential of their home/lifestyle automation. It stifles innovation and artificially constrains consumer expectations.

Some brands provide ecosystems that are very open and allow the consumer to integrate devices in whatever way suits them [LINK BRAND EXAMPLES], other brands restrict their systems under the guise of providing convenience for a ‘small’ monthly subscription, for them to have 24/7 access to their own personal data.

From my perspective, I don’t want the data from my home, leaving my home, to be stored on ‘in the cloud’ somewhere, so that I have to pay a monthly fee to have the cloud then return this data to me, so that I make use of it. I find this to be very invasive and restricts my options, as I don’t want to be paying multiple subscriptions to multiple brands, not on top of all the other subscriptions that come with living a technology enabled life (e.g., internet service provider, on-demand TV service, photo storage, on-demand music, content subscriptions to creators I enjoy, etc.) – it all adds up very quickly.

[Mention royalty fees to become part of some ecosystems, e.g. HomeKit? This stifles small independent brands, as they either must shell out excessive amounts of money, or they have to be standalone, in which case for the average consumer they may not be all that useful because it doesn’t integrate – even if the device itself is state of the art].

There’s a real problem in the IoT market that is limiting the real potential for consumers.

Thankfully, a project is underway to achieve interoperability among smart home and IoT devices and it is being spearheaded by some of the biggest technology companies (Amazon, Apple, and Google) alongside some IoT specific brands (Zigbee Alliance). The end-goal of this project is a new standard known as **Matter** [source: <https://buildwithmatter.com/>] which aims to roll out compatible devices from 2022 onwards, hopefully removing a lot of problems listed above. Ultimately, empowering consumers to make whatever choices work best for them, rather than what works best alongside the devices they currently have.

So, from next year onwards, the smart home technology sphere looks great. But what about today? What about all those devices purchased over the past decade? Given potential technology constraints it’s unlikely most devices will be able to operate with Matter. This doesn’t make them defunct, as they will likely still be supported into the future on their current platforms, but it does make them less useful and again, possibly constraints future device choices if the consumer is already heavily invested.

This brings me to my solution, a smart home hub, that is open-source, modular, and enables end-to-end communication between devices – for example, a motion sensor triggering notifications on your phone and turning your outside lights, all whilst recording this data and making it available on-demand to the end-user. This is what I set out to achieve.

### Readily available solutions

There are a number of open-source solutions that have set out to solve the problems set out above. A high-level analysis of each is outlined below:

#### Home Assistant

This project goes back to November 2013 (source: <https://en.wikipedia.org/wiki/Home_Assistant>) when it was first published to GitHub. It is built using Python. Over time it has become one of (if not the) most popular home hosted IoT device and automation manager.

Home Assistant has an active development community that keeps it at the forefront of automation and encourages the community to create new ‘integrations’ with unsupported brands – providing standards that must be adhered to before integrations will be merged into the application.

Another big component of the Home Assistant platform is the mobile app, which supports both iOS and Android, making it very accessible. It enables the control and monitoring of smart devices, providing similar capabilities.

Whilst this is a fantastic product, there are several issues that push it beyond the average user’s capability. The user interface is not very intuitive and can feel somewhat clunky – especially when unfamiliar. Due to the extensive nature of its feature set, it can feel arduous to complete simple tasks, for example, to add a new device and setup a simple timed routine can require an understanding of YAML (YAML Ain’t Markup Language – see <https://yaml.org/>), Home Assistant configuration options, and have some technical capability to diagnose any problems as they arise. None of this is to say that this is not a good option, in fact it’s a great option for users who have technical acumen and are willing to spend the time to learn and understand the platform.

#### Node-RED

This was originally developed by IBM and describes itself as a “flow-based development tool for visual programming.” It wires “together hardware devices, APIs and online services as part of the Internet of Things.” (source: https://nodered.org/) All of this to say, it attempts to simply the automation of various smart devices using visual tools, somewhat akin to low-code platforms (see <https://en.wikipedia.org/wiki/Low-code_development_platform>).

This platform is also open-source and developed with JavaScript using the Node.js framework meaning it provides a very responsive and dynamic user experience. It is a much more user-friendly experience, thanks to the flow-based programming, which enables the user to drag-and-drop ‘nodes’, connecting them together in whatever way the user sees fit to achieve their desired automation. The nodes can be thought of as building blocks, each which a very specific function and purpose. This makes the connection of these nodes (building blocks) easy to understand.

Where Home Assistant requires YAML, Node-RED opts to use the more widely used JSON (JavaScript Object Notation) for advanced configurations. These configurations (also known as flows) are readily shared amongst the community (see <https://flows.nodered.org/>), which showcases the strength of the community that has developed around this platform. A strong draw to this platform is the fact it is ‘no-code’ by default, with code only required to build custom functionality.

The user interface is much more intuitive and encourages the end-user to have a ‘play’ creating different flows to see what can be achieved. It is very minimalist and more akin to a blank canvas.

Node-RED has great functionality but it is very concentrated in the automation sphere. It does not have the ability to manage devices, data, or dashboards. It can facilitate each of these but relies on third-party platforms to provide this functionality. For example, Node-RED can be integrated with Home Assistant for a ‘best of both worlds’ but this quickly starts to rule out the simplistic, it ‘just works’ end goal.

#### Others

There are many other smaller open-source platforms innovating in the IoT sector but in the interest of brevity I have focused on only the two main platforms with functionality overlap with the Smart Hub platform that will be developed.

### Requirements

Researching current solutions has clarified some of the key requirements that Smart Hub will need to deliver to in order to make it a platform worth using.

* Easy to install and configure
* User authentication
* Records IoT device data
* Makes it easy for the end-user to create automations using devices they have setup
* Enables grouping of devices so that device groups can be treated collectively
* Enables notifications to be sent to the user via e-mail and/or to their smart phone
* Enables the user to setup devices that can be used to ‘trigger’ automations (e.g. motion sensor) and devices that can be ‘triggered’ by automations (e.g. light bulb).
* Once configured, the platform will run 24/7 recording, parsing, and communicating with all devices and/or users are prescribed by the configuration.
* The platform should be modular and enable future development by the open-source community.
* The platform should make it easy for user’s to create accounts by, for example, enabling accounts to be created through social accounts (using OAuth – see <https://en.wikipedia.org/wiki/OAuth#:~:text=OAuth%20(Open%20Authorization)%20is%20an,without%20giving%20them%20the%20passwords.&text=OAuth%20is%20a%20service%20that%20is%20complementary%20to%20and%20distinct%20from%20OpenID>)
* The platform must be accessible outside the user’s LAN (local area network), so that they can monitor and/or configure their system remotely.
* The platform must be able to work with different protocols (e.g., Zigbee and API based devices) and manufacturer devices.
* The system must be relatively inexpensive to setup and to run.
* The system must be well documented so that if any problems arise, even non-technical end-users can self-diagnose and resolve issues.
* The user interface must be clean, simple, and encourage the end-user to explore the platform, rather than shy away.
* It must be interactive, providing users with feedback when they have completed actions successfully and when they haven’t.
* The platform must be fast and responsive – any sluggishness is discouraging to end-users.
* The platform must hide the complexity from the user and present only what is useful and actionable.

### Potential Languages and Frameworks

* PHP
* C#
* NodeJS and React
* Python
* Django

### Development Language

* Python vs C# vs PHP
* COAP (Constrainted Application Protocol) (RESTful) vs MQTT (includes QoS)
  + Elaborate on QOS
* Synchronous (HTTP) vs Asynchronous (Websockets)
* Raspberry Pi & Zigbee2MQTT

## Chapter 2: User interface design

This chapter should describe, illustrate and justify the user interface design of your proposed system. Not all projects will have a significant user interface component, for example if they are back end algorithms or experimental projects. For projects without a GUI a short overview of the interfaces to the software should be outlined. For projects that have more substantial graphical user interfaces there should be an explanation for how the design has been developed including any feedback that shaped the design. The goal being to ensure that anyone building on the system understands the reasoning behind the UI and the feedback of users that led to its design.

* Websockets and MQTT
* Considerations for users browsing by screen readers – icons with sr-only tags to describe icons
  + Leveraging title attributes in font-awesome to autopopulate sr-only tags

### Multi-device platform

Mobile first philosophy

### Inspiration

* HomeKit?
* HomeAssistant?

### Feedback

* Layers were adding complexity, e.g. accessing device from within location made it difficult to know where exactly the end-user was
  + Result: added breadcrumbs so user is always aware of their location relative to the starting position
* Creating triggers
  + It was not immediately clear that only specific values could be provided with certain trigger types (e.g. values only for >, >=, <, <=) and without any form of validation or user notification, user events were not being triggered. User could not understand why
    - Solution: provide validation
  + Restrict metadata fields to only those that the device actually uses

### Goal

* Hide complexity
* Simplify user experience
* Make it easy to navigate
* Flexibility is important, but more important is that the system automatically constrains user inputs so that the process is as simple as it can be.
* Made the UI predictable – no sudden deviations. Easy for the user to know what to expect and how to use the website.
* Kept button colours consistent for each task, e.g. delete always red, add always green.
* Used icons related to the task at hand so the user was able to get a sense of what each component was for, before they had to go digging.
* Provided a help section to help resolve some of the show-stopping issues user might encounter.
* Provided about section so the user had context to what this platform was aiming to solve.
* Add coloured icons on navbar to help draw attention to these items, so the user becomes very aware of them, and can access them IF they need to.
* Mindful of users navigating the website with screen readers – provided contextual information to all icons so that even without the icon, it’s very clear what purpose they are serving.
* Breadcrumbs make it easy to return to ‘home’.
* Constraining user choice means the user is always clear on their options.
* Providing links throughout to help users quickly jump to other areas of the platform that are related, e.g. adding a location from the devices list view.
* NavBar delineates between smart hub functionality and account/user settings, to keep them very segregated, so neither gets lost in the noise.
* Notifications on errors and successes – colour coordinated to provide an emotion response appropriate, e.g. green for success, red for errors.
* Used dropdown navigation with headers to continue the clean, functional and consistent theme of the UI – keeping everything predictable and easy to navigate.
* Interactive feel to the website by changing table row background colours so it’s easy for the user to see which item they’re about to click.
  + Made table rows clickable and changed the cursor icon to invoke that trained response, from accessing all websites – maintaining the predictability
  + Also provided options in the same place every time, should a user prefer to select exact options or be unfamiliar with the pointer cursor icon indicating the user has hovered own a link.
  + JavaScript.
* Discuss Mobile First – e.g. Navbar hides when on smaller screens so that it is not taking up unnecessary space.
  + Mobile devices (not including tablets) account for over 50% of web traffic globally (see <https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/>)
  + It was important to ensure the website catered to the needs of these devices, i.e. space is at a premium.
  + The UI was created with a mobile-first ethos, i.e. catered to mobile devices by default, with it being adapted to large devices (such as desktops) afterwards. This ensures it has been optimised to work as expected on these devices.
* Made heavy use of margins and padding to create space – given that the platform captures and displays a lot of data and information, it was important the user had a positive experience and a key piece of this was not to clutter and overwhelm the user – achieved using spacing.
* XML Http Requests for toggling device … (AJAX)
* CSRF Tokens to stop Cross Site Request Forgery
* Automatic detection of device type, making use of Zigbee MQTT exposes topic – this details what the devices capabilities are – these are parsed and stored when the device is created.

## Chapter 3: Architecture design and algorithm explanation

A high level overview of the architecture of the code should be provided in this section. The overview should be designed to help another person seeking to adapt and maintain the software and should refer to the source files used in the project. Where relevant this section should also explain the design of any api interfaces that have been designed for the project to enable others to easily interface with the project. This chapter should also include a description of any complex algorithms that may be hard to understand simply by reading the code and its comments. Where algorithms and architecture are based on other work this should be clearly explained as well as any references to external explanations of algorithms or architecture used in developing the software.

### Hardware

#### Operating System

For this system to be useful it will need to be running 24/7 so that it can parse and record messages, in addition to sending notifications/triggering devices. With domestic electric prices at an all-time high, and more importantly, the environmental cost of running the hardware, power consumption was a key consideration for deployment. The Raspberry Pi 4 was an obvious choice. It is portable, requires nothing more than a power cable, SD card and network connection. The memory and processor are relatively powerful and power consumption is significantly lower than a desktop computer, using 2.85 watts per hour idle and increasing only slight to 2.98 watts per hour under moderate load. [source: <https://raspi.tv/2019/how-much-power-does-the-pi4b-use-power-measurements>] This compares very favourably to the average desktop computer which consumes 100 watts per hour [source: <https://smarterbusiness.co.uk/blogs/how-much-energy-do-my-appliances-use-infographic/>] and takes up significantly more space, also generating more heat and noise.

The Raspberry Pi 4 also comes with built in Wi-Fi, Bluetooth, and USB 3 – all of which add to its suitability as an IoT hub (with some devices using Bluetooth to communicate, and USB being a universal port working across all platforms). Another plus is the ability to use Power over Ethernet (PoE), meaning the device can be placed anywhere there is an ethernet cable – with no requirement for a mains electric connection.

#### Zigbee device communications

* Using a Zigbee CC2531 packet sniffer which I flashed with a custom firmware to enable it to be used with Zigbee2MQTT
  + Had to use Raspberry PI GPIO pins with jumper cables connected to the CC2531 board. See <https://www.zigbee2mqtt.io/information/alternative_flashing_methods.html>
* Once flashed, I connected it to the Raspberry Pi
* Installed Zigbee2MQTT on Pi and configured the network to permit devices to join
* Setup devices so that communications were picked up by Zigbee2MQTT
* Researched MQTT protocol to understand how I could interact with this.

### Software

#### Environment

Smart Hub is built using Docker as its foundation. This enables the system to be deployed rapidly on any hardware device with Docker installed, abstracting away environment variables from the development process.

Docker works by building *containers* which live inside of a docker image. A Dockerfile is used to define the exact specifications of the image it should run – the Dockerfile for Smart Hub is shown below.

Text

Description automatically generated

[include figure]

The above code informs Docker what base image it should use as the operating system (**Python:3**). Once the image has been downloaded and is running, the commands from lines 3 onwards specify commands that should be run each time the image is started. Here, it outlines Linux system dependencies and Python dependencies that should be installed as part of the build process.

The power of Docker is that it enables these images to be created effortlessly with the knowledge they will run on any system that supports Docker – this eliminates the development versus production environment risk. The development of the application was carried out on a Macbook Pro through Docker. Which meant that deploying it to the Raspberry Pi, whilst a completely different system, with different hardware and capabilities was not an issue, because it ran inside the exact same Docker environment.

In order to create a platform with distinct separations of concern, i.e. the database is completely isolated from the Python application, and so forth, each application was built in its own container. Docker has another tool, *docker-compose* which allows for container level specifications written in YAML – see the *docker-compose.yml* created this applicationbelow.

Text

Description automatically generated

Each of the indentations below *services* is a container name, with the code specifying the container requirements indented below the header. Within each are the specifications outlining what image should be run for the container, any **environment variables** that need to be passed through from the host machine. The use of environment variables is an important security feature, it enables the passing of sensitive information, such as passwords or application programming interface (API) keys, without them being visible when the code is shared (e.g. via GitHub). The actual secrets are saved to a file that is not shared, named **.env** – see below.

Text

Description automatically generated

Whilst all the containers are important, the main container is **web**. The YAML has been scripted so that *web* will only start up once the containers it depends on start – this is important because it is not able to operate without access to the database; and, Django has been configured to use **memcached** (the name of the container and the application) for caching data. It stores key-value pairs in-memory – more on this later.

The Dockerfile installs all the Python dependencies I have specified in the *requirements.txt* file. This happens while the image is building – this is important because the containers rely on these dependencies to run. The key example here is *web* which is configured to start a **Django** application. Mention that Docker containers are ephemeral and all data stored is lost when the container is stopped – therefore it’s important to setup a local directory mirror, for example it would be terrible if the database data was lost every time the container stopped.

Django is a web framework built using Python. It is considered a ‘high-level’ framework as it comes with all the ‘low-level’ functionality built-in – some examples are database abstraction, authorisation and authentication, handling of the HTTP request-response axis’s, among many others. I chose this framework as it enabled me to jump start into development, focusing on the specific functionality of my application, rather than reinventing the wheel by developing my own user management system, and the like.

The backend database used is PostgreSQL – this is a free open-source database, with a Python language driver, and can store many of the modern development fields, such as, Geometry, JavaScript Object Notion, primitives and many others. It was initially released in 1986 and has stood the test of time, using it for development I have the confidence that it is not going to disappear overnight.

* Python
* Django
* PostgreSQL
* Memcached
* Explored the use of Celery and RabbitMQ

#### Data Models

Django provides an Object Relation Mapping (ORM) system for database abstraction. This is very powerful as it means that the database Structured Query Language (SQL) code is handled by Django and no SQL is required for development. Instead, data models are specified, which include relationships, data types, and validation – using Django’s own ORM syntax.

Text

Description automatically generated

Above is an example of a simple data model. This specifies the data fields and types – note the models, **CharField** and **ForeignKey** – and validation – note the attributes, **max\_length**, **blank,** and **null**. Django abstracts away the SQL implementation required to build this table. Instead, I perform to commands *makemigrations* and *migrate* – with these commands, Django builds the SQL code to create the table **DeviceLocation,** and then runs that code in PostgreSQL so that the tables are created.

This model is not *just* a database model though, it also contains methods (a function in Python) that can be called on a specific **instance** of the model. In some instances there were methods that I wanted to perform on more than one instance of the model, for these situations I created a custom **model manager**. You can see an example below.

This also enabled me to create custom queries that I could re-use, which would be applied to the model as a whole, rather than a specific instance of the field (which is equivalent to one record in a SQL database).

The framework is quite refined and enables the abstraction of queries from model managers, known as **querysets** which leverage class-based development by using inheritance and polymorphism. The querysets can then be inherited by the model manager(s), which is then inherited by the data model. This provided me a neat and modular platform to work from, it was especially helpful as the system became more complex.

Text

Description automatically generated

This is the complete end-to-end *model* for the object DeviceLocation. The items in paranthesis beside the class name, are the classes being inherited by that class. This structure was very helpful.

Even though this model is very simple, there is a lot happening here. [EXPLAIN self, Django double underscores, filter with commas is essential AND criteria in SQL, type checking in method signature.

I developed the application using a class-based paradigm which Django supports, this meant I was able to leverage built-in functionality and supplement with my own using inheritance, abstraction, and polymorphism.

For example,

* CustomUser model to capture home\_location and permit use of email address without need for separate username
  + Discuss using Django allauth and the ability to interact social apps – which I’ve integrated using Google
* BaseAbstract inheritance to add common fields to all models
* (User) Device to (hardware) ZigbeeDevice
* Device and DeviceLocation
* Device:
  + DeviceStates
* ZigbeeDevice
  + ZigbeeMessage
  + ZigbeeLog
* Events:
  + EventTriggers
  + EventResponse
  + Notifications
* Notifications
  + NotificationSetting
  + Pushbullet
  + Email
  + Notification (Log)

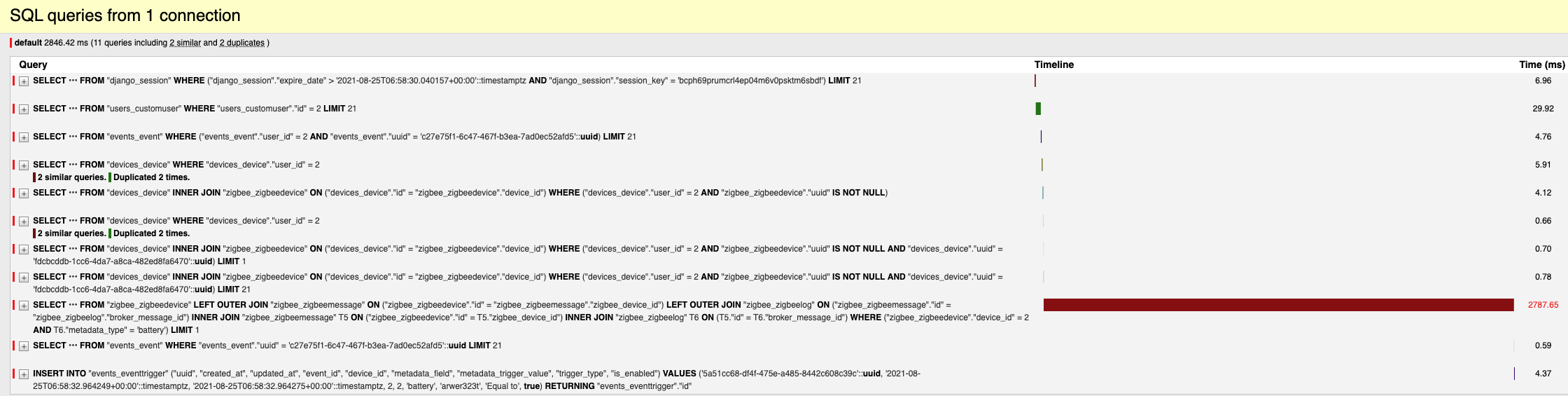
#### MQTT protocol

##### Subscribing

##### Publishing

Raspberry Pi

* Docker
* MQTT
* Django (python framework)
* PostgreSQL
* Zigbee protocol
* API protocol
* Using pip freeze to export requirements
* Celery
* Memcached
* Django management commands
* Query analysis - Optimising model design through SQL Explanations, e.g.



Identified the red time period as an issue – assessed further using SQL explanations to understand where this time was being spent and how I could optimise it.

* 
* Decision to add UUID field and use for URL params – security rationale – discuss mixin usage
* Creating an EventResponse model was a challenge, because the architecture I have designed means that hardware devices with different protocols are stored in different models. For invoking an event response, it did not make sense to link a hardware device to the EventResponse model because it would require a ForeignKey to be added for every type of device that is or will be supported.
  + For example, my model may start with just a ZigbeeDevice FK. Then I develop the ApiDevice model, so now I need to update the EventResponse model and include ApiDevice. In future I add the MatterDevice object, now I need to add this to EventResponse. This is quite laborious and adds a very manual component, it also requires database migrations each time. It introduces redundancy too, as only one of these FK fields will be populated per database entry, the rest will be NULL values – so what’s the point of them?
  + This architecture whilst function, wasn’t optimal. Instead, I used Django’s **GenericForeignKey** and **GenericRelation** classes. They required more setup and appreciation of their nuances, but they gave a lot more control over the relationships – enabling creation of a **GenericForeignKey** field which stores a model for triggering. This is much more flexible, enabling any of the aforementioned models to be stored in one field. Excellent.
  + To then enable reverse relationship calls, e.g. providing the ability to see what EventTriggers a ZigbeeDevice is connected with, a **GenericRelation** field was created in the ZigbeeDevice and associated models, which provides the association to the EventTriggers model.
  + Using this a device or a location can be specified. If a location is specified, then all devices in the location will be triggered. – bit complex to implement.
  + This enabled a polymorphic Device field within EventTriggers.
  + This provides a great deal of flexibility and provided a mechanism by which any device can ne triggered without an overly complicated model.
  + One other downside was the inability to use ModelForms, however, it would not have been possible to use a generic form anyway – in fact, it simplified the process by reducing the amount of form validation required.
  + See <https://docs.djangoproject.com/en/3.2/ref/contrib/contenttypes/#generic-relations>
* Django’s ForeignKeys are a one-to-one relationship.
* Discuss abstract classes – want to use them but Django’s implementation is not sufficient – allows inheritance of fields but no way to specify methods that must be created. Thought about using Python ABC but decide it was not worth the additional complexity with Django ORM.

## Chapter 4: Experimentation and Implementation

This chapter should be included for projects that include experimentation, such as projects with a machine learning component. This section should outline how the experiments conducted in the project have been designed to accurately measure likely performance in a realistic usage of the completed system. It should also highlight why each experiment was run, visually show the results e.g. in graph, discuss what was learnt from each experiment and how the project was adapted based on what was discovered. Appropriate experimental scripts, data and result files should be included with the developed software and referenced in the text so that the experiments can be reproduced and the results can be easily analysed.

* Access to Zigbee device information
  + Zigbee listener with custom firmware
  + Zigbee2MQTT installed on RPi which integrates with Zigbee listener
* Integrating MQTT listener/parser with Django so it has access to models and can create necessary objects
* Investigating the Django CBV system to understand the injection points for passing through request objects, success/fail messages, and restricting access dependent on user attributes
* Using reverse foreign key relationships in Django
* Getting access to a data field in another object so that it can be used in a view to provide information
* Deploying the architecture so that it’s as user friendly as possible (docker on RaspberryPi)
* Populating metadata field options on a per device basis – using XHR, model methods, views and urls
  + Plus validate values – especially validating!
* Creating Django filters using variables as the column name – needed to use a dict in the format {column\_name: column\_value}. These could then be unpacked using Python \*\*dict\_name format. Which essential substitutes the variable values in its place.
* To understand the properties, I had available within each view, I ran a terminal window and followed the Django logs, after attached print statements to the specific view and methods, for example, for Events detail view:
* Event Triggering:
  + I noticed early on that there are two categories of device: 1) those that publish an update when one of their values changes, and; 2) those that publish an update at regular intervals regardless of whether the device values have changed or not.
  + This caused some issues.
    - Constant event triggering even though the event had already been triggered and it was still ‘active’
    - With the above came constant loops invoking all event functionality at regular intervals – consuming a lot of unnecessary resources
    - Resulted in SPAM notifications
    - Resulted in EventResponses being triggered continuously
  + Had to figure out a way to store the last device state and only invoke the event trigger sequence if this state had materially changed.
  + Implemented this using caching.
    - Parsed messages – stripping fields that were irrelevant (e.g. last\_seen) in the context of event triggering
    - Compared against the parsed version of the last cached message
    - This was the first step in removing unnecessary tasks, if the message content hadn’t sufficiently changed, then I did not check for event triggers/invoke event responses.
    - The second phase is whether the actual tracked values had changed – this is an important check, because a message may have been deemed to have changed if the battery level has changed, or link quality (device attributes rather than sensor data points).
    - These could trigger an event if the non-trigger fields have changed, while the triggered value has remained the same as it was, but the event would be triggered again if the value matches the saved trigger value.
    - I was able to check these by saving a copy of the cached message, and passing it through to the event triggering process, then checking the exact trigger fields to determine if they have actually changed.
* Created my own template tag for status icons to avoid many manual copy paste if statements
* Text

  Description automatically generated

The terminal output of which is:

* Text

  Description automatically generated
* Understanding MQTT nuances: e.g. untidy disconnections from MQTT server – server thinks client is still connected, therefore when trying to reconnect the client with the same name, the broker does not permit this – ends up in a loop cycle until manual intervention.
* Duplication due to MQTT rebroadcasts -> needed a way to reduce data logging.

Graphical user interface, text, application, email

Description automatically generated

* Docker web container starting before DB container – causing issues with starting the application
  + Solution to implement bash scripting that uses ‘wait-for-it’ command to query ip and won’t run the command until it’s available
* Memcached
* Celery

## Chapter 5: Testing

This section should include a justification for the approaches used to test the resulting system. These may include such approaches as unit tests, manual and/or automated end 2 end testing, performance testing etc. The section should explain what parts of the project are at greatest risk of having uncaught bugs within them, for example because of their complexity, and how the approach to testing has been developed to analyse those areas in detail. For particularly complex algorithms for example this section can outline the design of the test cases to ensure good test coverage. The section should clearly reference any automated testing code, manual testing plans and test results included with the project that provides clear evidence that testing has taken place and can be easily used by anyone further developing the project.

* Created model factories using **factory\_boy** and **Faker**
* **Pytest** to execute unit testing suite
* Structured approach to testing, testing views, models, and forms separately – where customised code justified its own testing.
* Implemented continuous integration system (**CircleCI** or **TravisCI**) to ensure tests were run on every git commit.
* Implemented git hooks to prevent direct committing to the main branch, ensuring that only pull requests that passed the integrated test suite could be merged.
* Results from testing:
  + Some permission mixins weren’t working as expected – meaning users could modify objects they did not own.
* Built out test helper functions to perform repetitive tasks, e.g. converting response content to lower case / creating a list of objects and looping through to make sure they ALL appear in the response
* Mention subTests() for looping and how it enables extra detail from tests by passing through extra params to identify the specific instance(s) that failed
* Mock objects to override CSRF token while testing – ensuring a predictable value

## Chapter 6: Evaluation and Conclusion

This section should include a general evaluation of the success of the project measured against the criteria stated in the introduction and/or requirements. An evaluation of the hardware/software environment and language used may also be presented. Suggestions for further work should also be discussed. Do not be afraid to be critical or to draw a negative conclusion; not all projects will be successful. This section should provide a thorough and honest reflection on the process followed in the project and the results of that process. To do this well, the student should not leave any blind spots in their reflection and should identify the most and least successful aspects of the project. It should be written in such a way as to be helpful to a person seeking to adapt the project or to create a similar project in the future.

Weaknesses

* Matching zigbee messages to a user device is done via MQTT topic to user device friendly\_name field – bit clunky, if either changes the match is broke.
  + ZigbeeDevice is separate from ZigbeeMessage – these are linked via
* Need to invoke caching – too many queries
* Select notification channel per event instead of default to all
* Implement better event triggering – enable triggers to be stacked with AND OR conditions – at the minute they are all individual triggers, any trigger the event – no way to perform advanced logic
* Implement “on change” triggers, e.g. trigger event when state changes (heating on/heating off)

## References/Bibliography

Languages used

* Python
* HTML
* JavaScript
* CSS

Technologies and Libraries

* PostgreSQL
* Docker
* MQTT
* Websockets
* MVC pattern
* Django factories
* Django allauth
* Django crispy forms

Hardware

* Raspberry Pi
* Zigbee Listening Module
* Zigbee Sensor Temperature & Humidity Sen
* Yeelight wifi bulb

Tools for linting and testing

* PyLint
* Black
* Isort
* PyTest
* Factory
* Faker
* AutoPEP8

## Appendices

Test results

Printout of code for section/sub-sections of the application developed for the submission