



Developing a web application to Visualise IoT Data for Abacws Building

Author: George Grainger
Supervisor: Charith Perera
Moderator: Martin Caminada

GitHub Repo: <https://github.com/randomman552/Abacws-Data-Vis>
Final Commit: 6a74921ac4081d0be72ee93989624bfdb684e1a4

Visualiser Application: <https://abacws.ggrainger.uk>
API Docs: <https://abacws.ggrainger.uk/api/>

1 CONTENTS

2	Abstract	4
3	Introduction.....	5
4	Background.....	6
4.1	Existing Systems/Applications	6
4.1.1	Newcastle Urban Observatory System	6
4.2	Building Automation Systems.....	10
4.2.1	BACnet (Building Automation and Control Network)	11
4.3	Utilised Technologies.....	12
4.3.1	Common	12
4.3.2	Front-End Visualiser Application	13
4.3.3	Back-End API.....	13
5	Specification and Design	16
5.1	Specification.....	16
5.1.1	Functional Requirements	16
5.1.2	Non-Functional Requirements	18
5.2	Design	19
5.2.1	Separation of Concerns	19
5.2.2	Visualiser Service Design (Low Fidelity)	20
5.2.3	API Router Structure	22
5.3	Agile Development	22
6	Implementation.....	23
6.1	Enhancing 3D Model Performance.....	23
6.1.1	Reducing Detail.....	24
6.1.2	Problems With Exported Model.....	25
6.1.3	Results	26
6.2	Medium Fidelity Prototype.....	27
6.2.1	Loading the 3D model	27
6.2.2	Initial API Development.....	28
6.2.3	Basic API Endpoints	29
6.2.4	Initial Visualiser Development – React Hooks.....	31
6.2.5	Further Three-js Work.....	32
6.2.6	Generating Test Data	33
6.2.7	Graphing Data	34
6.2.8	Survey.....	35
6.2.9	Design Changes Based on Survey Responses.....	37
6.2.10	Improved Medium Fidelity Design	39

6.3	High Fidelity Prototype	40
6.3.1	Docker	40
6.3.2	Self-Documenting API.....	44
6.3.3	Advanced Query Features	45
6.3.4	Device Selection	49
7	Results and Evaluation	51
7.1	Results.....	51
7.1.1	Visualiser	51
7.1.2	API.....	52
7.2	Evaluation	53
7.2.1	Evaluation against Requirements.....	53
7.3	Reproducibility.....	61
8	Future Work.....	62
9	Conclusions.....	63
10	Reflection.....	64
11	References	65

2 ABSTRACT

The Internet of Things (IoT) is a recent emerging technology that provides substantial amounts of data previously inaccessible to humanity. The aim of this project is to develop a system to make it easier to access the data produced by/for the Cardiff University Computer Science Abacws building. Making this data more accessible opens the door to a range of possible applications and people wishing to access, process and utilise sensor data for the building. During this report I will explain the technical decisions I have made during the production of this system and evaluate the final product in relation to our software specification. The final aim of this project is to produce an API (Application Programming Interface) to control access to the data, with an accompanying 3D visualiser to show the data on a model of the Abacws building.

3 ACKNOWLEDGEMENTS

I would like to thank my supervisor, Charith Perera, for providing me with the 3D model of the Abacws building used throughout this project.

4 INTRODUCTION

The aim of this project is to develop a web application to visualise data gathered from IoT sensors in the Abacws building. This application will be like an existing application hosted by the University of Newcastle, which can be found here: <https://3d.usb.urbanobservatory.ac.uk>.

The purpose of this application is to make it easier to access and visualise IoT data within the Abacws building. Displaying the data on a 3D model like this is useful as it lets the user know visually where the sensor/device is located within the building. This is useful within a university setting as some studies may, for example, require access data such as the temperature in a particular room at a particular time. This application will empower people to easily collect and use data from these sensors with little technical understanding. The target audience of the application is professors and students at Cardiff University who want or need to access the data as a curiosity, for information, or as part of a wider study.

This application will present a navigable 3D model of the Abacws building with devices clearly shown inside of it. Upon selecting a device, the information and data related to that device should be fetched from a data store and shown to the user, the user will then be able to view historical data for that device to easily identify trends. For more advanced use-cases, the application will expose several endpoints to make it easy to query the dataset of all devices. This will provide a powerful aggregation of IoT data in a single source.

The application will be extensible to support the aggregation of data from various sources, such as Building Automation Systems responsible for handling building functions. This leverages the power of the IoT and will allow smart systems to make informed decisions based on the data provided by this application. For example, the data provided by this application could provide data to reduce the environmental impact of the Abacws building. This can be done by taking actions such as switching off air conditioning when the building is empty.

During the production of this web application, I will utilise modern web development technologies to increase the speed and quality of the final product. I will produce an easily understandable and maintainable codebase for future developers to look over for inspiration, fork the code base and improve it themselves, or to develop further for a real-world application.

Over the course of this report, I will be describing my approach to the development of this application alongside justification for the technical decisions I have made throughout. This will begin with me creating a software specification and end with the evaluation of my final product against this specification.

5 BACKGROUND

5.1 EXISTING SYSTEMS/APPLICATIONS

5.1.1 Newcastle Urban Observatory System

Newcastle University has a system with a similar aim to the one I am producing. In this section I will investigate their application which is hosted online at <https://3d.usb.urbanobservatory.ac.uk/> and see any ways in which it can be improved to better carry out its task. It also has an accompanying API with documentation hosted at <https://api.usb.urbanobservatory.ac.uk/>.

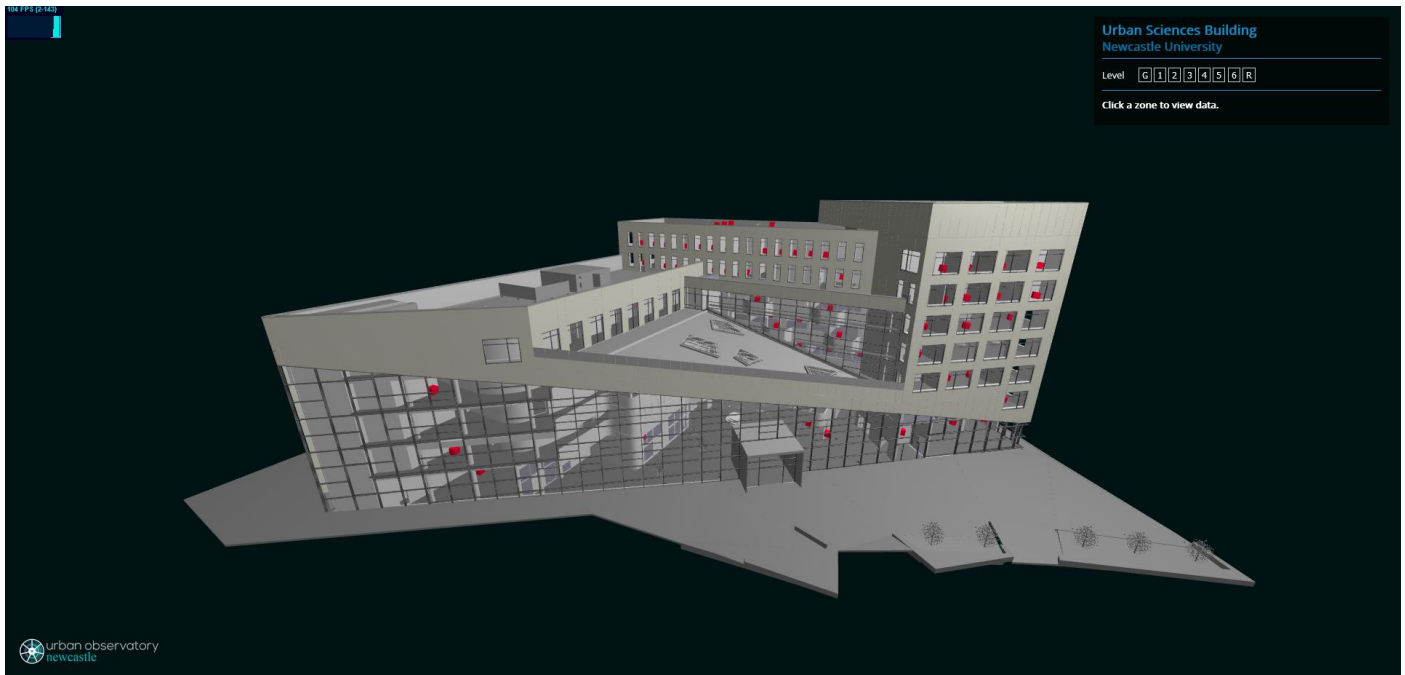


Figure 1 Screenshot of Newcastle Urban Observatory Application

The user interface (UI) of this application is split into two components. A model view in which you can navigate the 3D model of the building and select devices within it, and a data panel (on the right) in which you can slice through the floors of the building and view data for the device you have selected.

The application is split into two components. A data explorer (shown above), and an API hosted on a subdomain which is used by the data explorer to fetch information about the state of sensors in the building. The application utilises a protocol called BACnet which is commonly used in many Building Automation/Management Systems globally. I will investigate BACnet as part of this next section as it is an interesting point of consideration for future work.

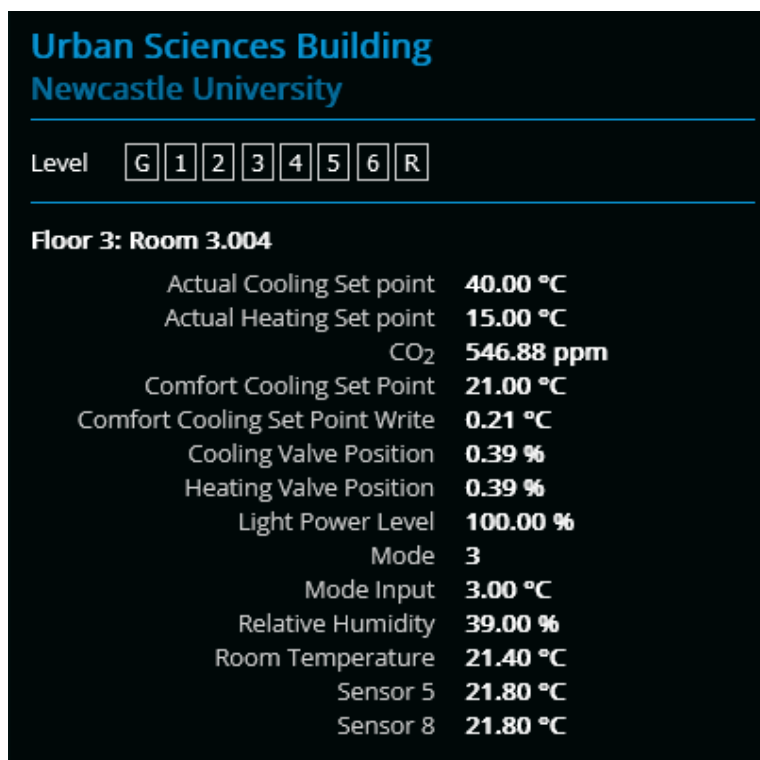


Figure 2 Data view upon selecting a device in Newcastle application

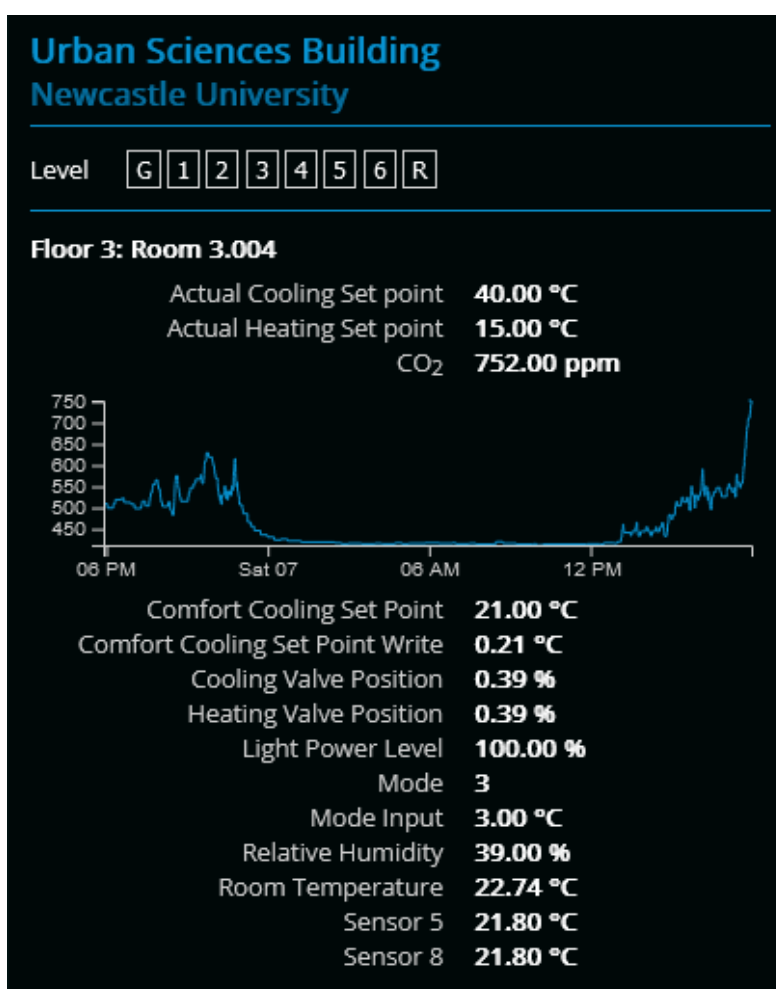


Figure 3 Data view upon selecting a field within a device in Newcastle application

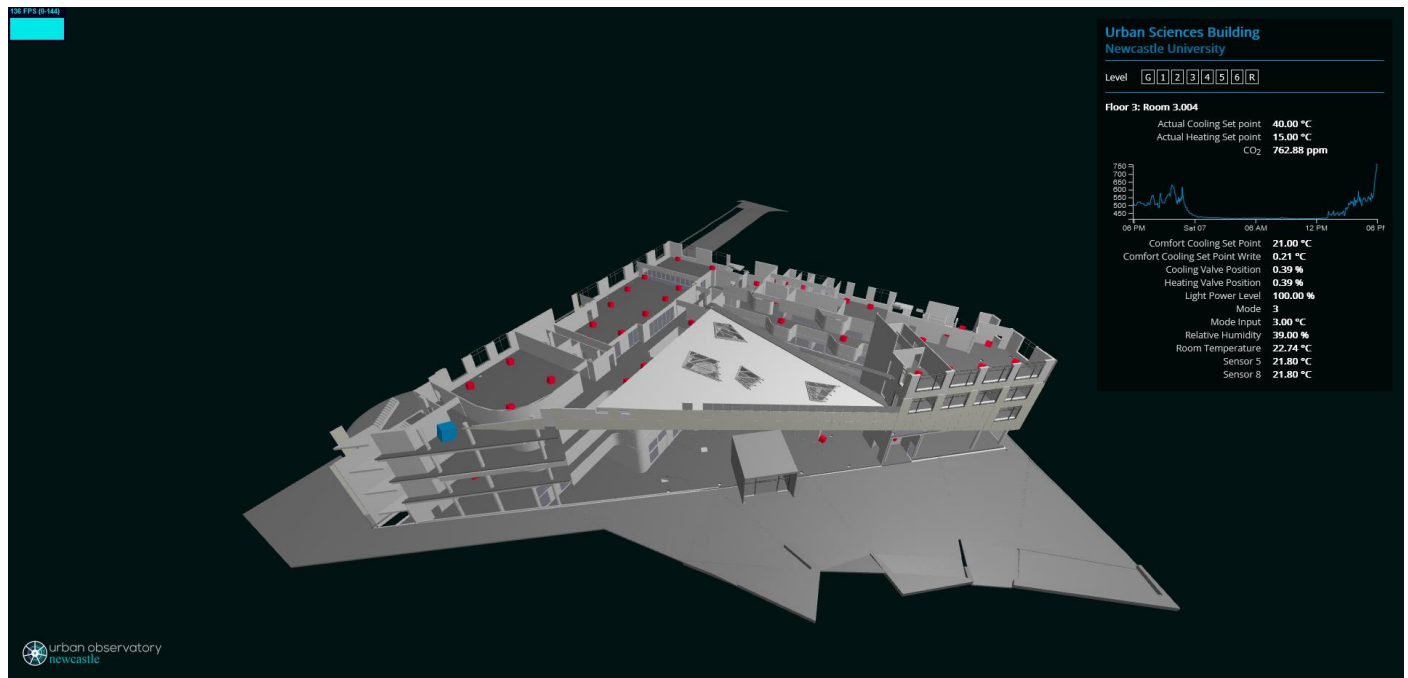


Figure 4 Sliced building model in Newcastle application

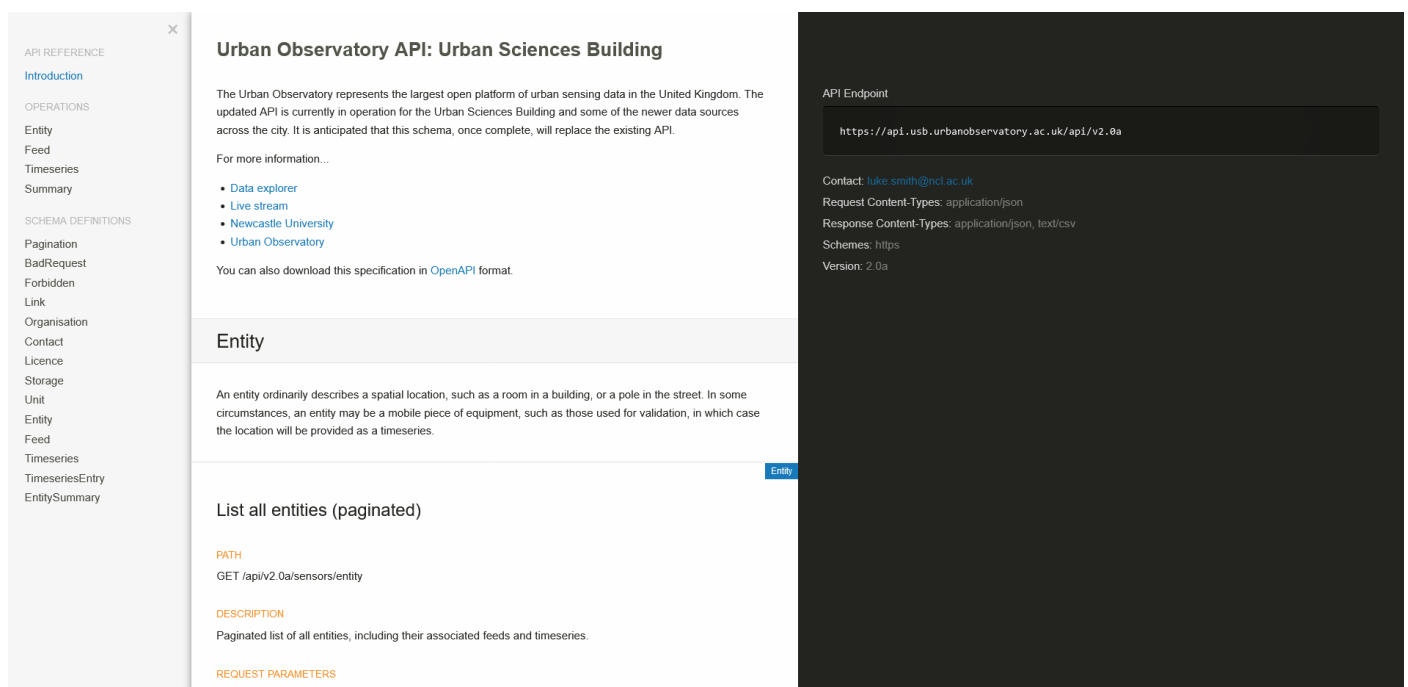


Figure 5 Newcastle application API Documentation

The Newcastle API is documented using the OpenAPI specification which provides machine and human readable documentation. I have also decided to utilise this specification in the development of my project as it makes the application more accessible to future developers/users and provides a way to automatically generate test suites and visually appealing documentation.

5.1.1.1 What it does well

- The application makes it clear when a node is selected from the model
- API data is very well tagged with metadata
- API documentation is clear and well maintained

- Control scheme for model navigation is intuitive
- Utilises existing data from the environment by using BACnet protocol to integrate with the existing Building Automation System

5.1.1.2 Potential Improvements

User feedback is lacking in some areas. For example, it is not clear which floor of the building the user has currently selected in the data panel on the right. It is also not clear how to display the graph for each data field after selecting a device. The cursor does turn to a pointer when you hover over these fields, but that in and of itself is not enough to properly suggest that the user can interact with them. A button would better indicate this functionality.

The data panel is very small on larger screens; and on mobile devices is largely unusable due to its fixed, small size. On smaller screens the data panel occludes the view of the model - this could easily be fixed with the ability to hide it.

The view of the 3D building model is not adjusted when you re-size the window. This does not majorly impact the functionality of the application, but it can be annoying.

Whilst this application has a very strong API and back-end system, the front-end application presenting the data to the user has some problems which need to be addressed. I will take these issues into consideration when designing the specification and UI for my application.

5.2 BUILDING LIFECYCLE MANAGEMENT (BLM)

It should be apparent that the application I am intending to produce is not unique. Building information systems are used throughout the world, this is especially common on new systems as 3D BIM models are readily available for use. Abacws is one of these buildings, and as such it is important that I look to existing applications created for similar purposes. Two such applications are referred to in a report (Huynh & Nguyen-Ky, 2020) which we will be discussing in this section.

The applications they discuss are used to improve the facility management (FM) process of the building. They argue that applications such as these can be used to *“harness the data-driven monitoring and automation in buildings to the greatest extent”*. I agree that the accessibility of the wealth of IoT data produce by buildings such as these can be used to enhance buildings in areas such as energy efficiency, which is an area for potentially massive savings in money and emissions for large commercial or academic buildings.

In this article, the authors seek to visualise data from buildings using *“building modelling and progressive web apps”* (PWA). A PWA is any web application which functions like an installed, native application (MSEdgeTeam, et al., 2022). Discord (a voice and video service) is a popular example of one of these applications. PWAs are based on the Chromium browser using a framework called electron, and while they act like real applications, they function as web applications under the hood.

During the article they prototype two applications, one is for a newer building (like the Abacws building) and the other is introduced for a legacy building with no existing 3D models. For the legacy system they must build the BIM model from scratch using floor plans of the building. Luckily as we are dealing with a much newer building, there is already a model available for us to use.

One of the main reasons to choose a web application is that it provides much better cross-platform support than other software solutions. The use of PWA or web applications means that a user on a mobile device such as a phone can access the data just as easily a desktop or laptop user. This makes the harvested data available to anyone, anywhere.

In the article (Huynh & Nguyen-Ky, 2020), during the development of the project, they utilise NodeJS, the TICK stack and Grafana to create the application. NodeJS is used to create an API, while TICK and Grafana provide the approach to data storage and visualisation. The TICK stack includes Telegraf (data collection agent), InfluxDB (high-performance database for large data handling and compression), Chronograf (InfluxDB visualising tool) and Kapactor (monitoring and alerting system for InfluxDB).

Their application was deployed using Docker which I have also decided to utilise in the deployment of my application. The advantages of Docker and why it is used will be discussed in the utilised technologies subsection of this background section.

The authors conclude that the two cases have *“depicted the potential of employing a web-app-based FM system in the BLM process”* (Huynh & Nguyen-Ky, 2020). The application they have produced binds together many IoT devices from different vendors and collates the data together. This application collates data from many sources and packages it together in an easy to access way, the use of web applications makes this even easier as a user can access data from any device, in any location.

5.3 BUILDING AUTOMATION SYSTEMS

“A building automation system (BAS) consists of a system installed in buildings that controls and monitors building services responsible for heating, cooling, ventilation, air conditioning, lighting, shading, life safety, alarm security systems, and many more” (Domingues, Carreira, Vieira, & Kastner, 2016). Building automation systems regulate the flow of information between various useful systems in buildings such as air conditioning and lighting. It would be possible to utilise a BAS to collect sensor data from the existing network within the Abacws building.

A major issue pointed out in the article (Domingues, Carreira, Vieira, & Kastner, 2016) is that despite their being several standards of communication to facilitate BAS, there is little agreed field-knowledge and there are gaps in functionality between major standards. This causes *“developers to repeatedly redefine basic concepts, creating their own solutions bottom up instead of relying on the existing body of knowledge”* (Domingues, Carreira, Vieira, & Kastner, 2016).

All the systems mentioned in the article (Domingues, Carreira, Vieira, & Kastner, 2016) refer to devices on the BAS network as “entities”. These “entities” then expose a series of “data points” which can be read-only or write-only. The read-only fields are sensors, and the write-only fields are actuators (devices that affect environment, such as a relay or switch). The “entities” can receive commands to change their internal state or use an actuator to affect their environment. For our purposes we are only interested in the read-only data points, as we do not aim the control these “entities” from our system.

These systems are relevant to this project as we can utilise them to harvest data from the sensors, which could then be displayed in the application. There are several protocols and communication standards which can be utilised to help integrate these systems together. The article (Domingues, Carreira, Vieira, & Kastner, 2016) discusses KNX, LonWorks, ZigBee, and BACnet.

The BACnet is of particular interest to us as the standard aims to provide good interoperability between devices manufactured by different vendors. It is also used by the Newcastle University application discussed in the previous section of this report.

5.3.1 BACnet (Building Automation and Control Network)

5.3.1.1 *What is BACnet?*

BACnet is “a data communication protocol for building automation and control networks” (Newman, 2022). BACnet provides a way of representing the functions of any device in a standardised model. It was originally an American standard developed by ASHRAE (American Society of Heating, Refrigerating and Air-Conditioning Engineers) in 1987. It has since grown to become an international ISO standard (ISO 16484-5 in 2003).

It provides interoperability between BAS devices made by different vendors, allowing these devices to communicate with each other easily. It can function over ethernet using IP or can use communication standards such as ZigBee. Devices are represented objects; these objects have multiple fields representing the real-world state of the device.

5.3.1.2 *How is it Relevant?*

This standard is relevant to this work as it provides a method of querying existing devices in the Abacws building for useful information, instead of requiring the installation of hundreds of redundant sensors. The existing application in use by Newcastle University’s Urban Sciences building utilises this protocol to treat the BACnet devices as sensors and display them on the website. While we did not have time during the development of this project to introduce this protocol, it would be a potentially interesting consideration for future development.

5.4 UTILISED TECHNOLOGIES

During the development of this project, I will utilise modern web development techniques and technologies to improve the quality of my final solution. These technologies make my solution more maintainable. In this section I will go over technologies utilised and why I selected them.

5.4.1 Common

There are several technologies we use in both the visualiser and API services. These will improve the developer experience and ease of deployment.

5.4.1.1 TypeScript (Microsoft, n.d.)

TypeScript is *“a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale”* (Microsoft, n.d.). TypeScript offers several advantages over JavaScript, namely:

- TypeScript is a superset of JavaScript.
 - This means that it adds additional features, without changing the fundamental language.
 - No need to re-learn the entire language.
 - All JavaScript is valid TypeScript
- Adds static types to JavaScript, making it easier to catch errors in development before they occur.
 - A major problem with JavaScript is that the variables are not statically typed. This is a source of many headaches during development and debugging where JavaScript implicitly changes the type of a variable, without informing you.
- Provides better type hints during development than JavaScript.
 - Makes development easier.
 - Easier to understand for people viewing the project for the first time.
- TypeScript can be compiled to any version of JavaScript, making supporting older or less compatible browsers easier as the TypeScript compiler makes all the code work with whatever platform we wish to support.
- JavaScript implicitly performs type coercion for comparisons, and TypeScript informs you where this is occurring. Type coercion can lead to hard to debug problems, TypeScript improves this.

TypeScript is widely used in the industry as a drop-in replacement for JavaScript. Due to the advantages outlined above, I will be using it instead of JavaScript wherever I can.

5.4.1.2 Docker (Docker, n.d.)

Docker is an industry standard virtualisation tool; it uses the power of the Linux kernel to run applications in small, virtualised environments called containers. It allows for the creation of easily portable “images” which contain their own dependencies. This allows for simple and easy deployment on any machine with Docker installed, making our application highly portable.

A file called a “Dockerfile” is used to configure a Docker “image”, like a setup script a “Dockerfile” is a series of commands used to create an “image”. These Docker “images” are then used to form the basis of a Docker “container” in which the application then runs. Once the application is no longer required, the container is destroyed.

Once the container is destroyed, all changes to the file system within the container are lost. To store data when containers restart, we use external “volumes” mapped to the inside of the container to store the data permanently within the host file system.

5.4.1.3 Docker Compose (Docker, n.d.)

Docker Compose is *“a tool for defining and running multi-container Docker applications”* (Docker, n.d.). This will be useful as we are building two separate components which can be run in their own respective containers.

Using Compose involves building a YAML (Yet Another Markup Language) file to configure the services for the application, the services can then easily be started from the command line. We will be using compose in conjunction with a container-based proxy system called Traefik (Traefik, n.d.), which handles routing of HTTP (Hyper Text Transfer Protocol) traffic for our containers.

5.4.2 Front-End Visualiser Application

5.4.2.1 React (Facebook, n.d.)

React is described by Facebook as *“A JavaScript library for building user interfaces”* (Facebook, n.d.). It makes it easy to build declarative, component based, responsive UI for websites using just JavaScript. It can be easily extended with additional functionality using NPM (Node Package Manager) modules.

I elected to use React instead of raw HTML and JavaScript for the visualiser. This is due to my previous experience with the library and how much easier it is to develop with. The extensive components available through NPM also make it a good choice, as we will be able to add additional functionality without having to develop it from scratch. Using React also improves the ease of future maintenance, as it is commonly used in the wider web development community.

5.4.2.2 Sass (Sass, n.d.)

Sass stands for *“Syntactically Awesome Style Sheets”* (Sass, n.d.) and is an extension of standard CSS which improves the developer experience. Sass, like TypeScript is a superset of JavaScript, is a superset of CSS. Meaning that all CSS is valid Sass. Sass is compiled into CSS and automatically handles differences between browsers. On top of this, Sass provides useful additions to the CSS standard such as for-loops.

5.4.2.3 Three.js (mrdoob, n.d.)

Three.js is a graphics library for web browsers. It provides an abstraction over the standard WebGL implementation. As defined in the requirements for this application, we need to be able to show a 3D model of the building with IoT devices plotted. Three.js is the obvious choice here, as it will simplify the final implementation and save development time as we will not have to write everything in raw WebGL.

5.4.3 Back-End API

5.4.3.1 Node.js (nodejs, n.d.)

Node.js is *“a JavaScript runtime built on Chromes V8 JavaScript engine”* (nodejs, n.d.). I have elected to use Node.js with express to build the back-end API for this application. The main reason I picked this over something like Flask and Python is to keep the programming language across the stack the same, this means we can use TypeScript in the front and back end which means that the only language that will need to be understood to read the codebase is TypeScript.

Using Node also gives us the vast library of functionality provided by NPM, from which we can easily leverage additional functionality such as database connectors. The static typing of TypeScript also beats out the dynamic typing from JavaScript and Python, and as such is a superior choice.

5.4.3.2 MongoDB (MongoDB, n.d.)

As we are dealing with unstructured data which can contain any number of fields, an obvious choice is MongoDB as it is a NoSQL database with schema. This will allow devices to provide whatever data they want, without the need for additional setup on the API side. MongoDB deals with unstructured data very well and allows us to query it in a natural way using JSON (JavaScript Object Notation).

Another reason for picking MongoDB is that I already have previous experience with it as part of my course, and in one of my own side projects, this familiarity with the database will reduce the amount of learning I would need to do with a new system.

5.4.3.3 Express (expressjs, n.d.)

Express is a *“fast, unopinionated, minimalistic web framework for Node.js”* (expressjs, n.d.). For building an API, express is the obvious choice as it allows us to easily create API endpoints by registering functions as endpoint handlers. As our API will be simple, using another more complicated web framework would overcomplicate the system.

5.4.3.4 OpenAPI 3.0 (swagger, n.d.)

“The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to the source code” (swagger, n.d.).

The OpenAPI specification allows us to define our API by writing a YAML file which defines the endpoints of our API. This is useful for us as it can be used to provide documentation and a testing suite automatically. This means our API can be self-documenting.

5.5 PRIMARY PERSONA

Tommy Shaw	
Picture	About
	Tommy Shaw is a computer science student at Cardiff University. Technically skilled, interested in the IoT and programming his own tools and apps. Currently in his third year of study. Running a study as part of his final year dissertation that requires him to source information from IoT devices in the Abacws building.
Information	Goals
Age: 20 Occupation: Computer Science Student Interests: Programming, IoT	1. Collect IoT data from the Abacws building as part of his dissertation 2. Access high quality documentation for the system used to access the data 3. Be able to request data over a particular time range for his dissertation 4. Be able to filter the data easily to narrow down his search window and reduce the amount of redundant data he obtains

6 SPECIFICATION AND DESIGN

6.1 CONSTRAINTS

The main constraints we must deal with are due to this application being web based. To maintain reasonable load times and performance, any 3D rendering of a model is going to be constrained and the file(s) containing the model should be small to reduce load times. Any 3D model used will have to be simplified so that it can be rendered at reasonable quality on most devices, including modern mobile devices and laptops with integrated graphics. The 3D model will also have to be simplified so that it can be rendered on mobile devices, which sport significantly less processing power.

6.2 SPECIFICATION

After studying the Newcastle University application covered in the background section above, I compiled a series of requirements I wish my application to fulfil. These are split into a list of functional and non-functional requirements.

6.2.1 Functional Requirements

6.2.1.1 Devices must be able to add data to the application

Acceptance Criteria

- Device must be able to send a HTTP request to a web endpoint to add data to its representation within the application.
- An authorisation token must be provided by the device to carry out this task.

Assumptions

- There are devices with data to send.
- Devices are connected directly to the internet.
- Devices can communicate via HTTP/HTTPS.
- The application is hosted at a reachable IP address or domain.

6.2.1.2 User must be able to see a model of the building

Acceptance Criteria

- A user viewing the web application can view a 3D model of the Abacws building.
- The building should be navigable using orbit controls.

Assumptions

- User has an internet connection.
- The application is hosted at a reachable IP address or domain.

6.2.1.3 User must be able to see devices represented within the building model

Acceptance Criteria

- User viewing the application can see IoT devices represented within the model.

Assumptions

- There are devices to represent within the model.

6.2.1.4 User must be able to slice through the 3D model to view different floors

Allowing the user to slice between floors makes the interior of the building easier to access when trying to find devices within it. Otherwise, the user would be forced to find the devices through the windows on the model, which would not be user friendly.

Acceptance Criteria

- Upon selecting a floor, the 3D model must be sliced to show no floors above the selected floor.

Assumptions

- N/A

6.2.1.5 User must be able to view the data provided by a device

Acceptance Criteria

- Upon selecting a device, the user will be able to view the most recent data entry for that device.
- This data should be displayed in tabular format, with options to view historical data for fields.

Assumptions

- There are devices for the user to select.
- Devices the user selects have valid data.

6.2.1.6 User must be able to visualise historical data for a field on a chart/graph

Acceptance Criteria

- While viewing the most recent data entry for a device, the user should have the option to view historical data of that field, for that device.
- User could also have the option to plot multiple data fields onto many charts.

Assumptions

- There is enough historical data to plot.

6.2.1.7 User should be able to export data in JSON format

Acceptance Criteria

- With a device selected, the user can select an "Export" option.
- They will then choose a location to save a file and be able to download the data in JSON format.

Assumptions

- There is data to export from the given device.

6.2.2 Non-Functional Requirements

- The 3D model of the building should be of sufficient quality to be easily navigable.
- The 3D building model should be rendered at 30 FPS minimum.
- Representations of devices in the 3D scene should be easily distinguishable from the model.
- It should be clear to the user which floor they have selected when slicing through the model at different heights.
- It should be clear to the user what device (if any) they have selected.
- Historical data should be plotted in an accessible, usable, and visually appealing way.

6.3 DESIGN

6.3.1 Separation of Concerns

Whilst designing this application, I have attempted to adhere to the Separations of Concerns design principle (SoC). The SoC design principle is the *“idea that a software system must be decomposed into parts that overlap in functionality as little as possible”* (Makabee, 2012).

For this reason, I have made the decision to split the application into three separate services. A visualiser service and a data handling service, where the visualiser (frontend) will display the model of the building and make HTTP requests to an API (backend) which will handle the processing of all data from IoT devices. The storage of all data will be handled by a MongoDB database.

This gives us three distinct services which together form the application. These services can then be networked together using Docker.

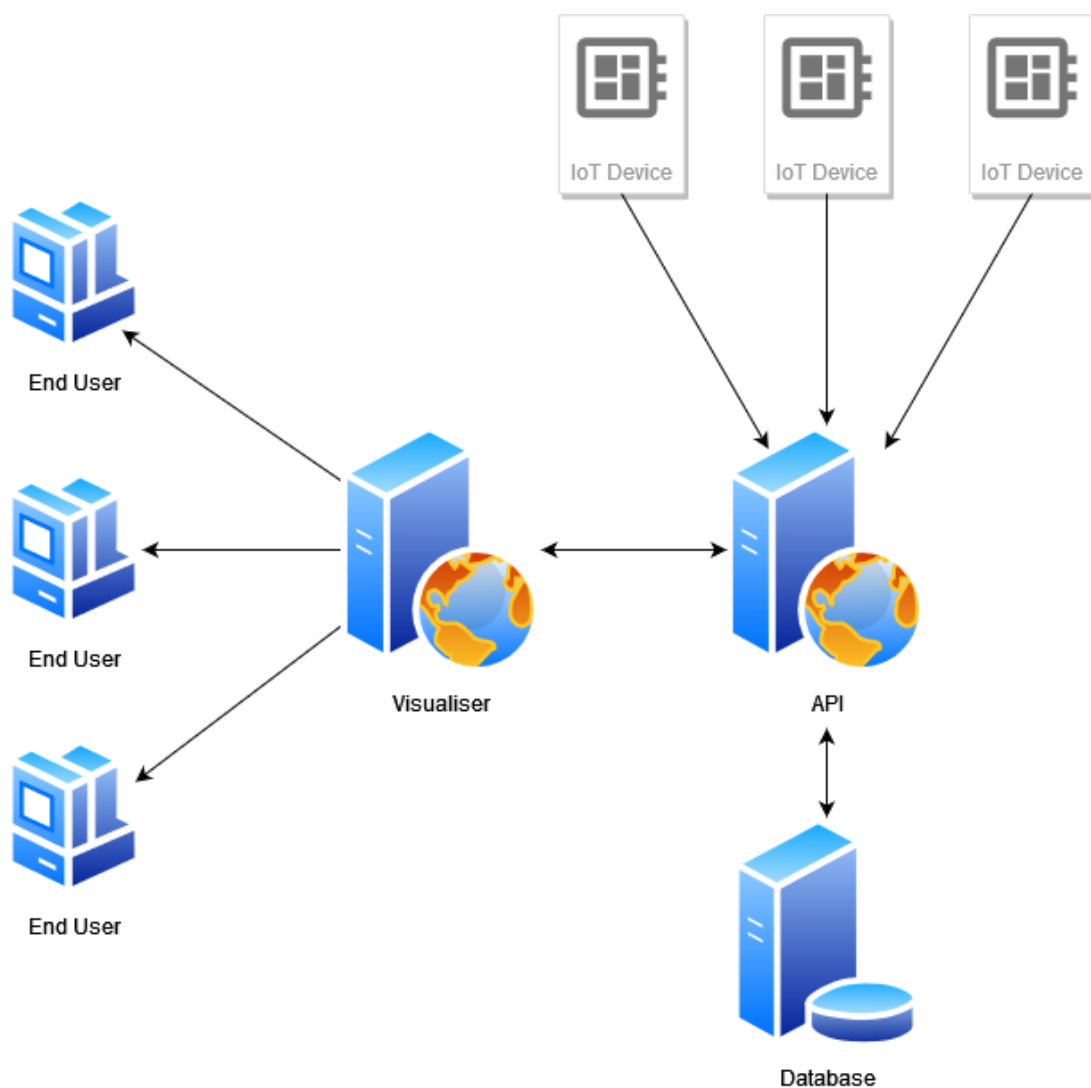


Figure 6 Basic software architecture for the application. Each machine shown in the diagram will be virtualised using Docker.

6.3.2 Visualiser Service Design (Low Fidelity)

I made some simple wireframe designs of the application before proceeding. These designs form the basic layout of the application.

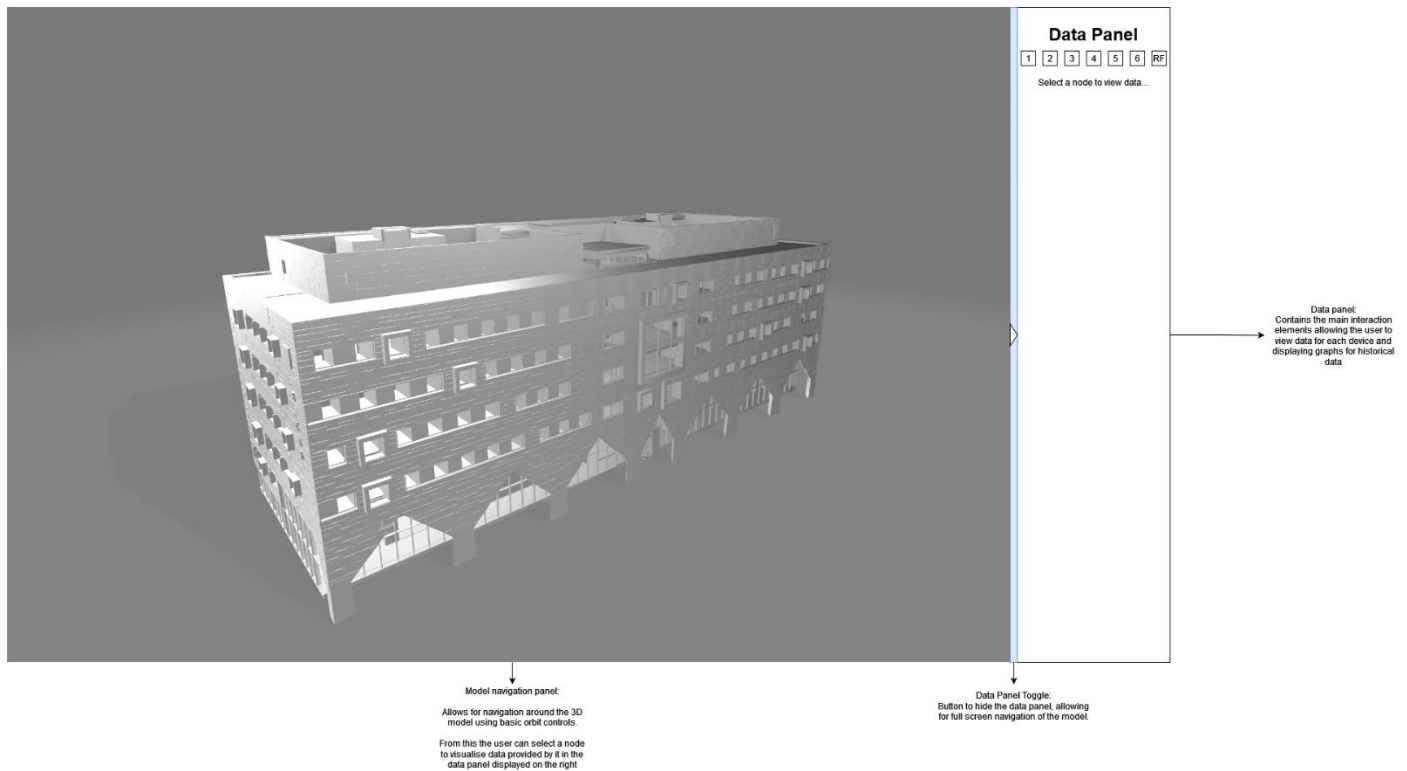


Figure 7 Low fidelity wireframe of the application

The design consists of two major components. A model navigation panel, and a data panel.

- The model navigation panel allows the user to navigate a 3D building model on which IoT devices are shown. The devices on the model will need to be easily distinguishable from the model itself, as such they should be in a different colour as to be easily visible.
- The data panel is displayed on the right of the screen and can be hidden to allow the user better access to the model on demand. It is split into several further subcomponents. The first of these is a floor selector that allows the user to slice the building model at each floor individually. This is useful as it allows the user to access devices that may be hidden within rooms which do not have windows or other openings.

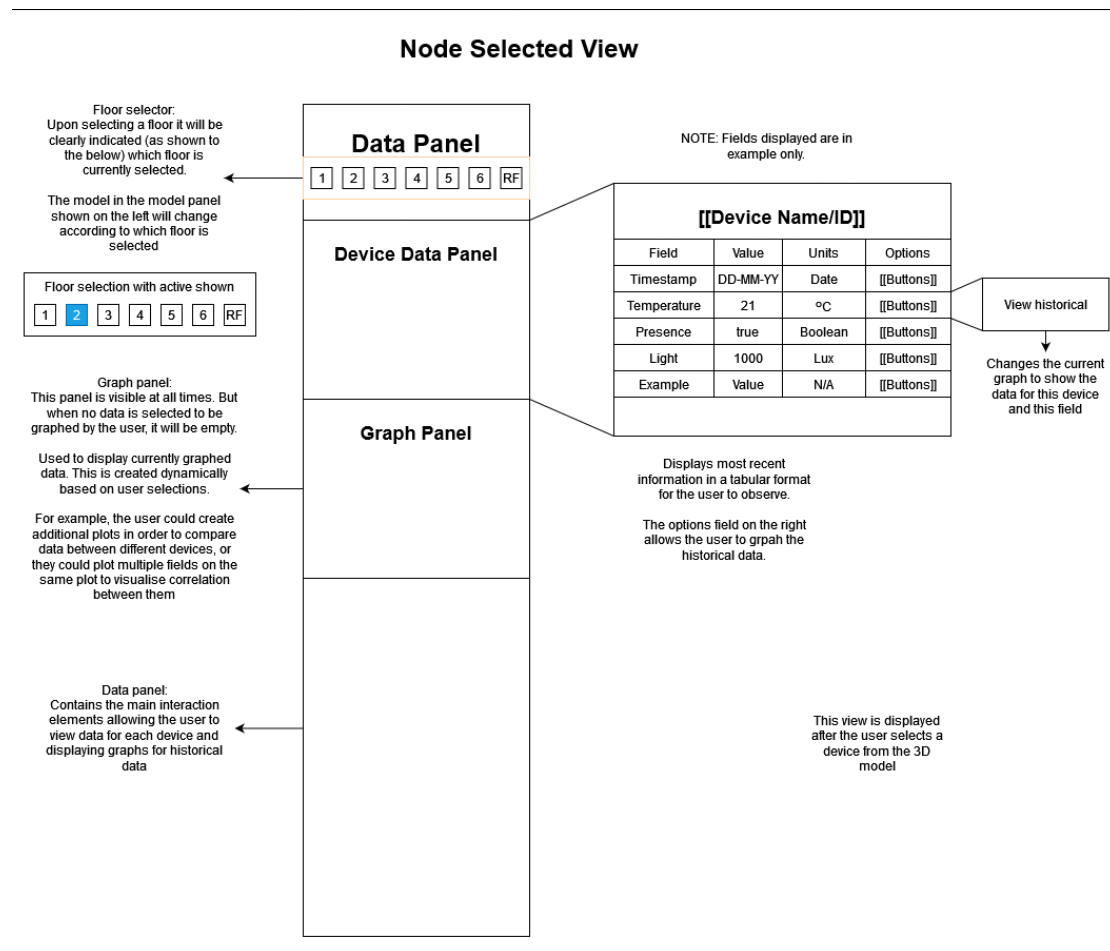


Figure 8 Low fidelity wireframe design for the data panel component

The device data panel contains information about the currently selected device and the option to view historical data for a specific field. While designing this interface I have attempted to minimise the number of clicks for a user to achieve any task; the fewer clicks it takes to perform a task the less likely users are to get confused.

The device data is displayed in a table, and each data field can have a value and units. The graph panel contains a graph of the currently selected historical data and, when the view history button is used, the graph is repopulated with new data.

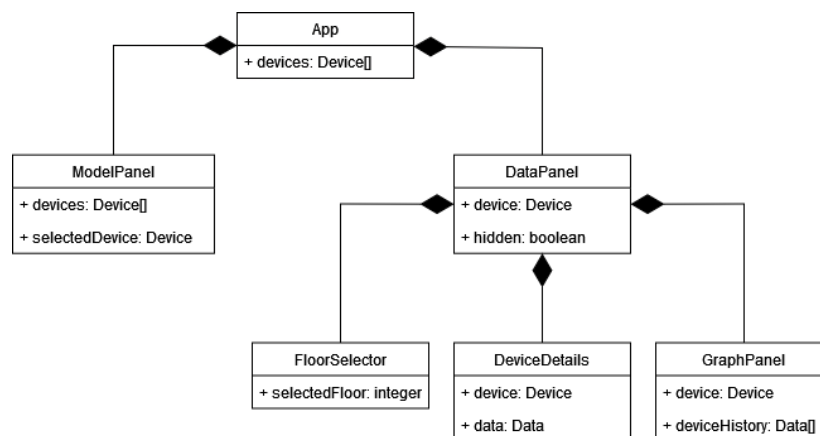


Figure 9 UML diagram representing the components that make up the visualiser UI

6.3.3 API Router Structure

The API is built as a hierarchy of request routers. Each request router checks whether it can handle the request, and if not passes the request further down the hierarchy to be processed by another router. This leads us to a hierarchy of routers organised like a tree. The leaf nodes of this tree are functions which generate a response to respond with (response generator).

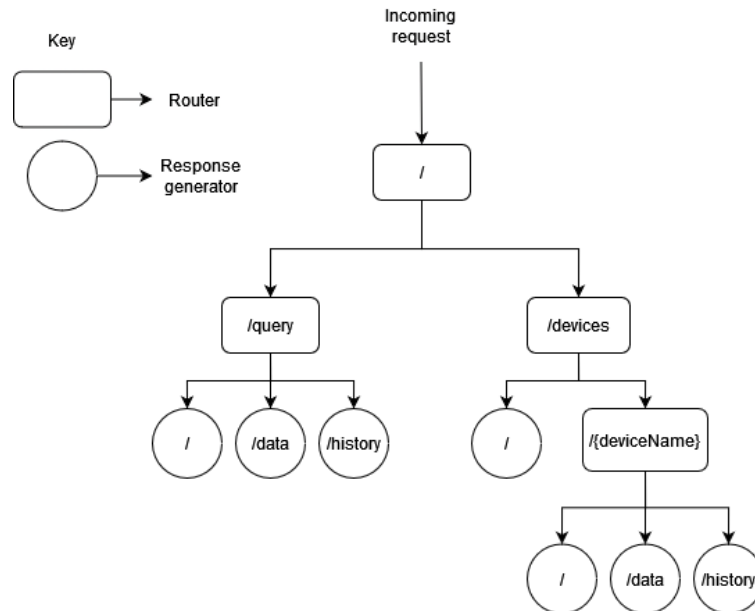


Figure 10 API request router structure.

6.4 AGILE DEVELOPMENT

I have decided to utilise a Kanban (a style of agile development) board to organise development of this project. This ensures that I finish the basic required functionality before applying polish and additional features. I used GitHub projects for this purpose. This is especially useful as it automatically adds any issues which I have added on GitHub to the board, allowing me to better understand my progress through the development process.

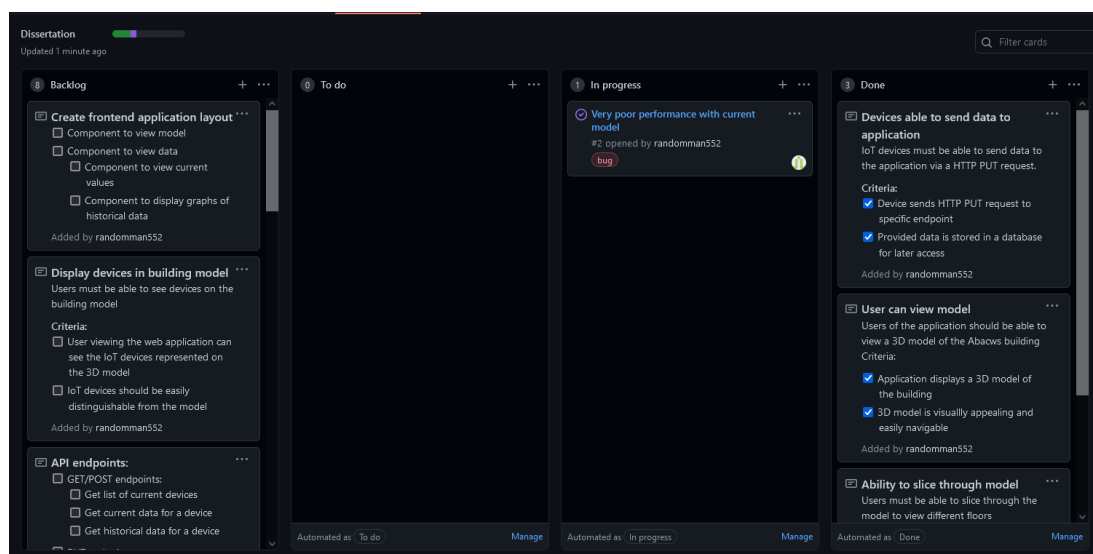


Figure 11 GitHub Kanban board screenshot

7 IMPLEMENTATION

7.1 ENHANCING 3D MODEL PERFORMANCE

At the beginning of this project, I was provided with a 3D model of the Abacws building by my supervisor. This model was incredibly detailed and included things such as pipes and electrical outlets which were extraneous for the purposes of this application.

The detail of the model resulted in poor performance, and it was apparent that I would need to simplify it before I continued. This model was a constant cause of performance issues until I established the best way to reduce detail. In this section of the report, I will discuss how this problem was solved.

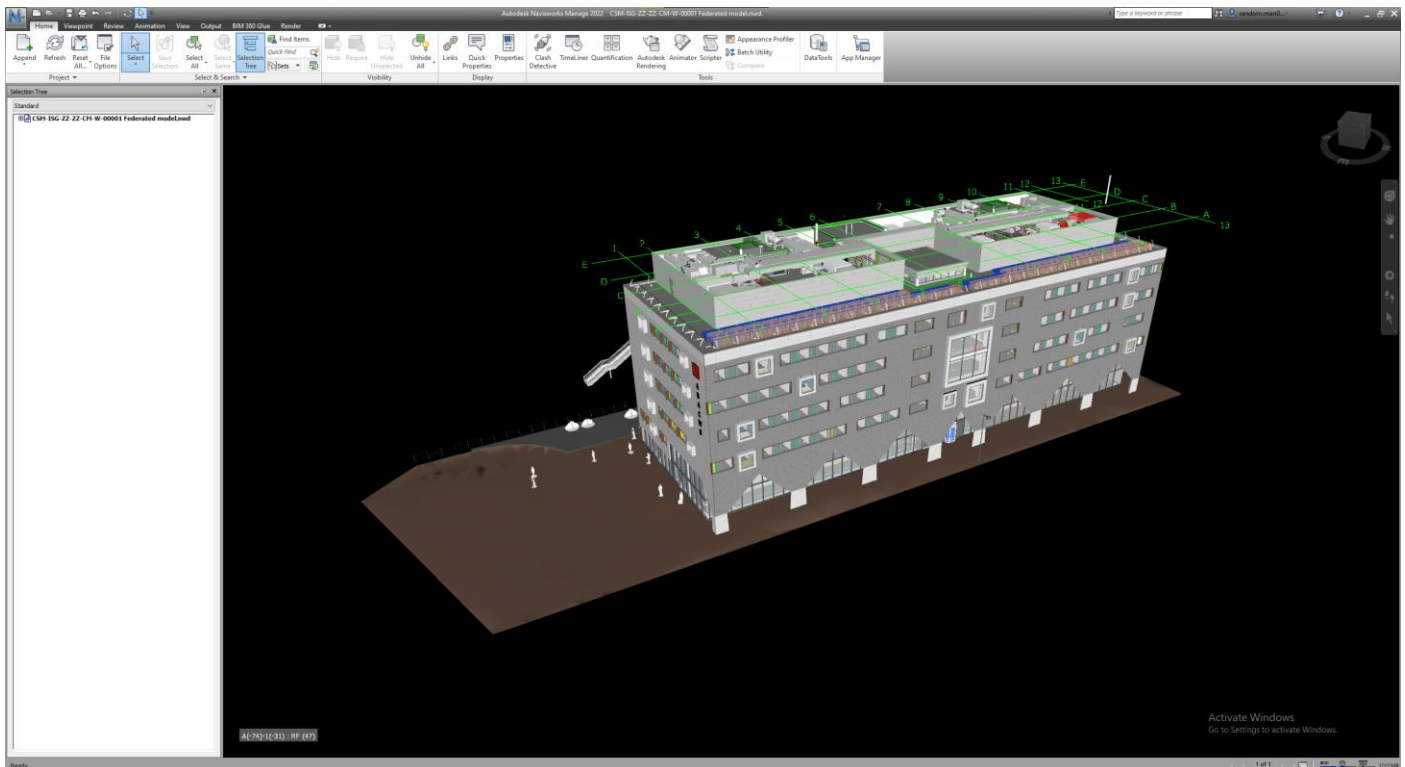


Figure 12 Original building model with details which we do not need as part of this project (displayed in Autodesk Navisworks)

7.1.1 Reducing Detail

The first course of action available to me was to remove elements of the model which were not required. This involved looking over the building and removing extraneous components. Pipes, electrical outlets, and furniture were removed from the model. This improved performance enough to make the model tolerable to work with. But it was still evident that the model needed further improvement before achieving good performance on lower end devices.

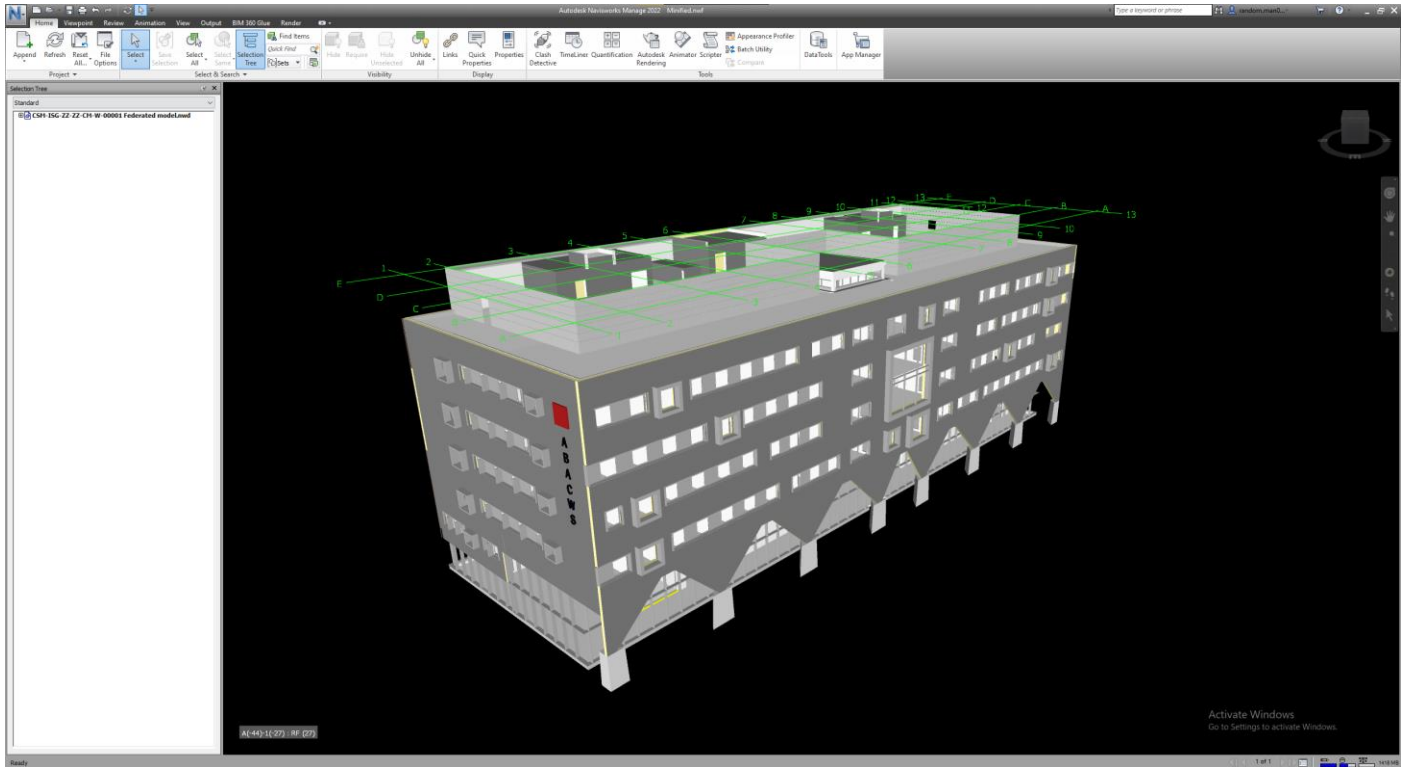


Figure 13 Building model with unrequired detail removed in Autodesk Navisworks

7.1.2 Problems With Exported Model

The 3D model of the building was given provided in a format used by Autodesk Navisworks. To load the model into the visualiser application:

- I exported it from Navisworks as an FBX file;
- imported it into Blender; and
- re-exported as a glTF file for use in the visualiser application.

The glTF format is “a royalty-free specification for the efficient transmission and loading of 3D scenes and models by engines and applications” (Khronos Group, n.d.). I have chosen this format as it is designed for transferring model information over an internet connection and results in smaller file sizes than FBX files. There were a few more issues with the model after it was exported from Navisworks that needed further correction:

7.1.2.1 Model Centring

When exported from Navisworks, the model was not centred around the origin (0, 0, 0) as I would have expected – it was about 1000 km away from the origin instead. This problem was trivial to solve in the end, but it took me a while to realise why the model was not appearing. To solve this issue, after importing the model into Blender, I moved the result to the origin before re-exporting the model as glTF.

7.1.2.2 Duplicate Materials

When exported from Navisworks, each mesh that made up the model was given its own material, this was an issue because mesh’s using the same material can be drawn together in a single render call, and as everything had a different material, the number of render calls was very high, leading to terrible performance. This was difficult to fix. The obvious solution would be to manually delete all duplicate materials, but with a model as big as this one that would take days of monotonous work. However, Blender has a scripting engine which uses Python, and after a quick search online, I found a script (MURATAGAWA, 2021) that would remove the duplicate materials.

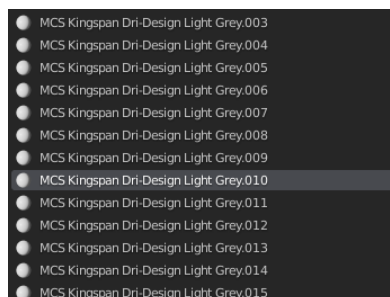


Figure 14 Blender duplicate materials

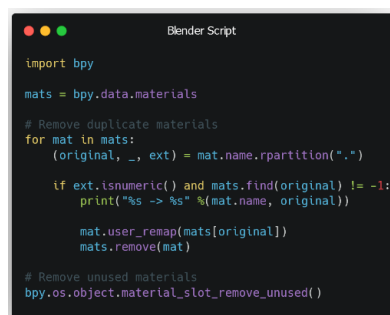


Figure 15 Blender script to remove duplicate materials

7.1.3 Results

As a result of my work on the model, the performance was now massively improved. I exported each component of the building separately so I could alter the material for each to provide better visual clarity (for example distinct colours for walls and floors). These enhancements also reduced the size of the model file, drastically improving loading times of the visualiser service.

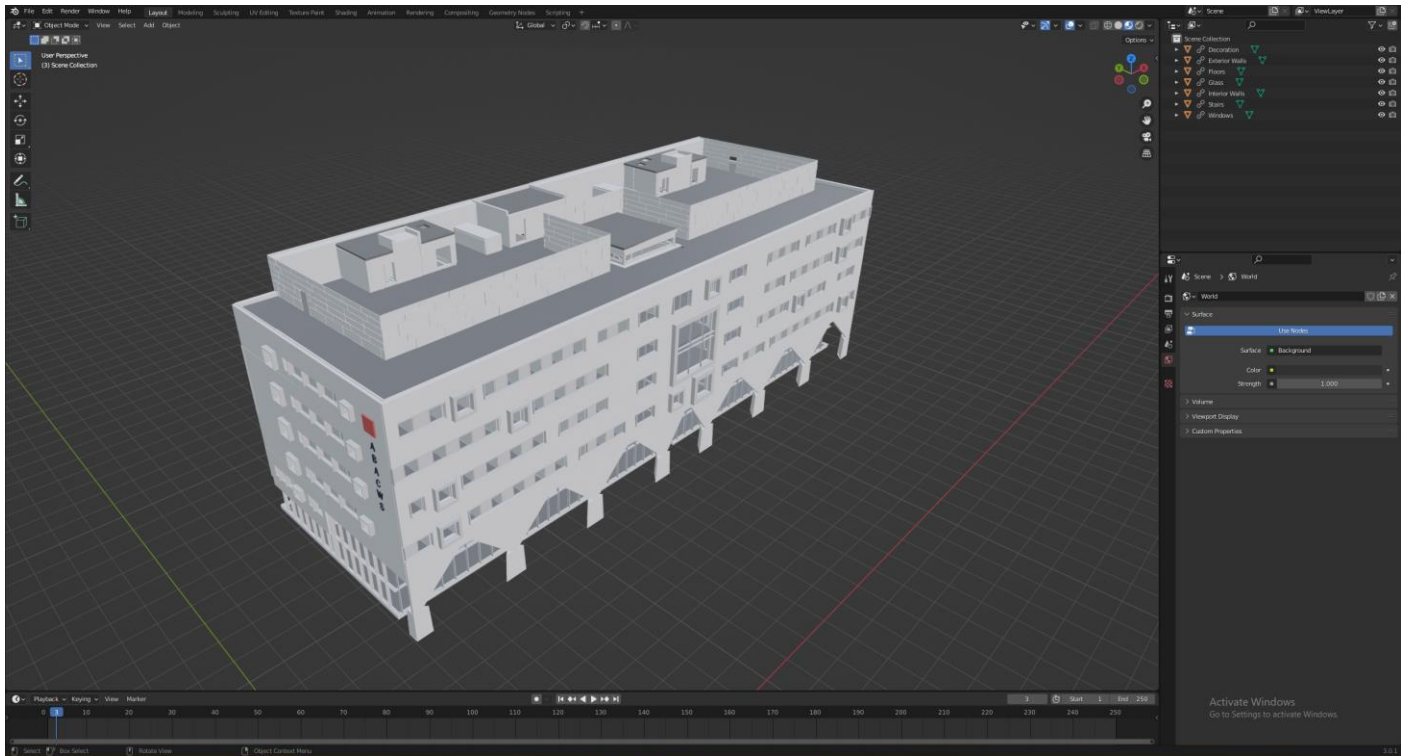


Figure 16 Enhanced building model in Blender

7.2 MEDIUM-FIDELITY PROTOTYPE

The next step was to create the initial application as a medium-fidelity prototype. I started this after planning the basic architecture of the application, creating the low-fidelity design, and deciding which technologies and libraries I wanted to use. The initial medium-fidelity prototype created from the design is shown below, it is my aim to improve this design in this section before proceeding to high-fidelity prototyping.

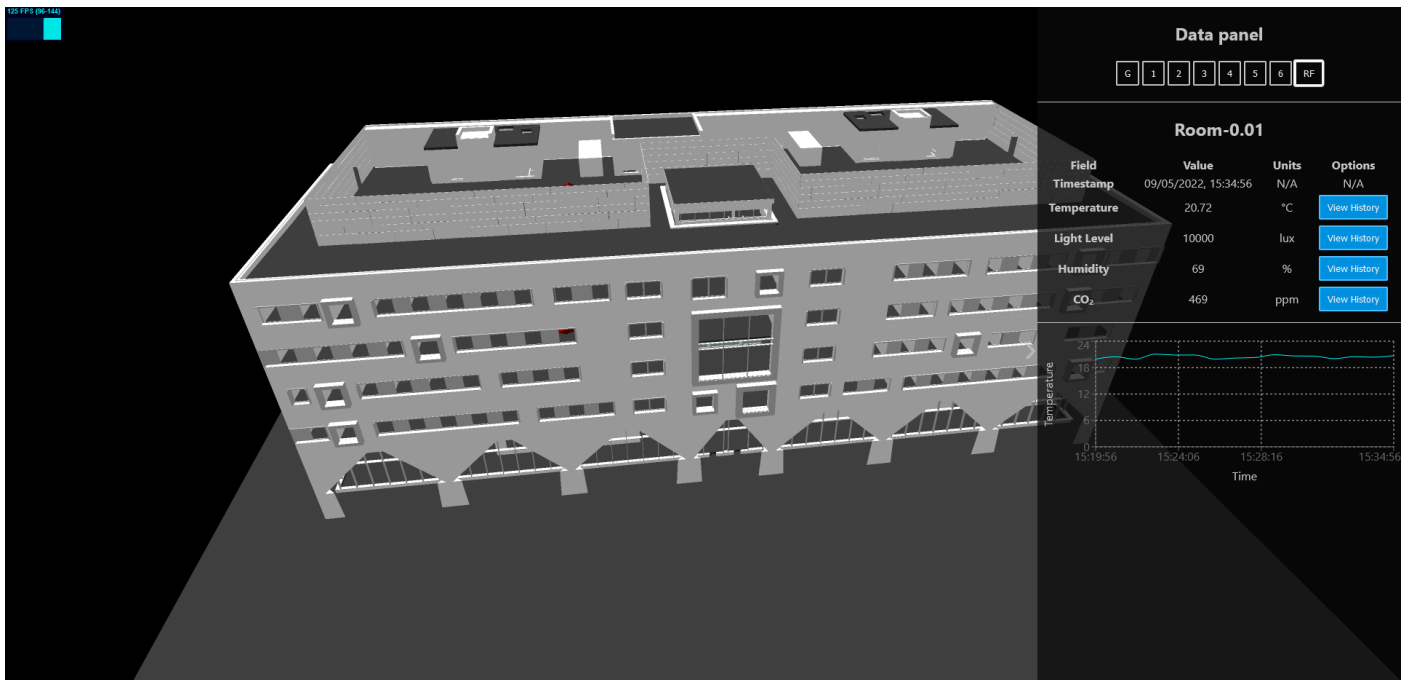


Figure 17 Initial medium-fidelity prototype built from low-fidelity design

7.2.1 Loading the 3D model

Direct integration between Three.js and React is not supported by either library. So instead, I created a class to encapsulate all the logic related to the 3D model, rendering of devices, and selection of the rendered devices. I then used DOM events to bind the two together using React hooks (Facebook, n.d.). This works nicely to keep the graphical code separated from the normal React code base, maintaining SoC (Separation of Concerns).

For the development of the model display I am using Three.js. It supports loading of models and provides several types of loaders to deal with different file types. We are storing our 3D model as GLTF in binary format (GLB) and Three.js provides a GLTF loader which vastly simplifies our graphical code.

```
// Load building model
const loader = new GLTFLoader();
const loadLayer = async(fileName: string) => {
  const layer = await loader.loadAsync(`/assets/${fileName}`);
  this.scene.add(layer.scene);
}

// Load all layers
for (const layerName of LAYERS) {
  await loadLayer(layerName);
}
```

Figure 18 Model loading code

7.2.2 Initial API Development

For development of the API, I had a choice between building it a SOAP (Simple Object Access Protocol) or REST (Representational State Transfer) API. I have decided to use REST as it is far simpler than SOAP and allows us to use simpler data formats such as JSON. Using JSON instead of XML reduces the overhead for data transfer, and as such will make our API more performant and flexible. As we are going to only be transferring data, we will not use any more advanced features of SOAP, further supporting our use of REST.

I started with setting up the API and its connection to the MongoDB database it depends on. Using MongoDB with Node was easy, and just required that I installed the MongoDB connector using NPM. A connection to the database is then achieved using the following code snippet where MONGODB_URI is the URI to the MongoDB instance:

A screenshot of a code editor window titled "Node MongoDB connection". The code is written in TypeScript and shows the import of MongoClient from 'mongodb' and MONGODB_URI from './constants'. It then connects to the database using the MongoClient constructor with MONGODB_URI.

```
import { MongoClient } from 'mongodb';
import { MONGODB_URI } from './constants';

// Connect to database
const client = new MongoClient(MONGODB_URI);
```

Figure 19 Node API MongoDB setup

This imports the MONGODB_URI variable from a constants file, which sets the variables from environment variables. Using environment variables allows for easy configuration of the application when in a Docker container.

A screenshot of a code editor window titled "constants.ts". The code defines several constants: PORT (defaulting to 5000), PRODUCTION (defaulting to false), MONGODB_URI (defaulting to 'mongodb://localhost:27017/abacws'), API_KEY (defaulting to 'V3rySecur3Pas3word'), and DEVICE_COLLECTION_PREFIX (defaulting to 'd').

```
export const PORT = Number(process.env.API_PORT) || 5000
export const PRODUCTION = Boolean(process.env.PRODUCTION)
export const MONGODB_URI = process.env.MONGODB_URI || "mongodb://localhost:27017/abacws"
export const API_KEY = process.env.API_KEY || "V3rySecur3Pas3word"
export const DEVICE_COLLECTION_PREFIX = "d"
```

Figure 20 API constants.ts file. Defines variables for use throughout the API retrieved from environment variables.

As you can see above, the file also provides default values if the environment variables are not set, or for when the API is running in a development environment where the environment variables may not be present.

A single file (api.ts) imports routers for the API's functionality. This allows us to split the routes across multiple files to make the code more readable and maintainable than it would be if contained within a single monolithic file. We separate routers by purpose to maintain SoC principles. For example, all routes for device information should be managed within one router, another route is used for the self-hosted documentation.

The separate routers are imported into api.ts and registered as express routes there.



```
import cors from 'cors';
import express from 'express';
import { devices, docs, healthcheck, query } from './routers';

/** Express app */
const api = express();
// Api will only respond to JSON
api.use(cors());
api.use(express.json());

// Register routes
api.use("/healthcheck", healthcheck);
api.use("/query", query);
api.use("/devices", devices);

// Register documentation router
api.use("/", docs);
```

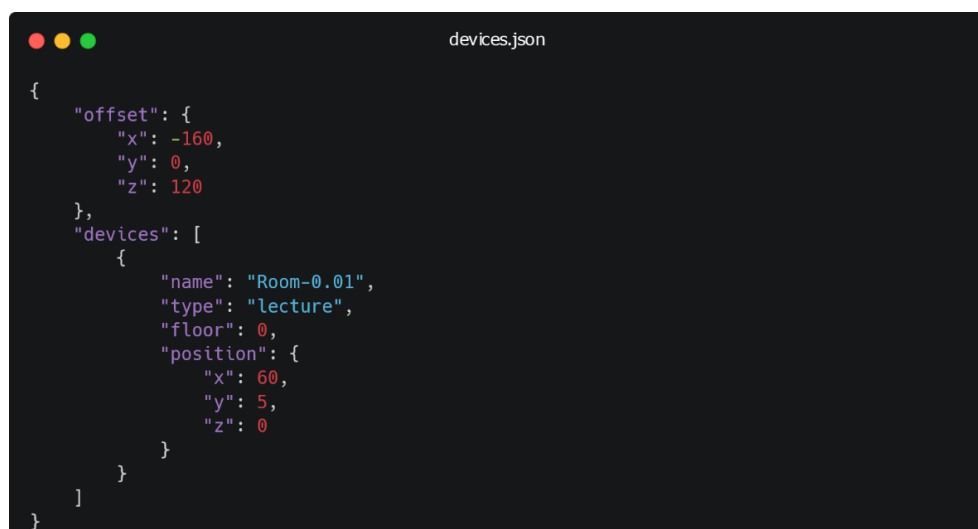
Figure 21 API api.ts file. Registers the routers for each segment of the API

7.2.3 Basic API Endpoints

After setting up OAS with my project, I proceeded to add the basic endpoints for requesting information about devices registered within the system. Whilst doing this I first needed to decide how devices would be registered within the application, to achieve this there were two methods I could use:

1. Add a devices.json file which would then contain all the information on the devices registered within the system. When the API started up, these devices would then need to be stored in the database, where the API could fetch them for future reference.
2. Add API endpoints to allow the addition and removal of devices remotely. This would require an API key of some sort to prevent non-authenticated users from adding or removing devices and would consequently be the more complex solution.

I chose the first method. This is because it is far simpler and does not require me to build in authentication at this early stage of development. This also works well with our Docker deployment, as we can simply add a Docker volume to pass through a different devices.json file if required by any deployment. We then store the devices in the MongoDB database when the API starts up, allowing us to access them as we would for any other data stored in the database.



```
{
  "offset": {
    "x": -160,
    "y": 0,
    "z": 120
  },
  "devices": [
    {
      "name": "Room-0.01",
      "type": "Lecture",
      "floor": 0,
      "position": {
        "x": 60,
        "y": 5,
        "z": 0
      }
    }
  ]
}
```

Figure 22 Sample devices.json file for API. The offset variable is applied to the device position for positioning in the 3D building model.

During this development stage I added five endpoints related to device info, data, and history. Device “info” refers to basic information about a device, such as its name, type, floor, and position. Device “data” refers to the last entry of data submitted to the API by the device. Device “history” refers to all data entries submitted to the API by a device.

Response generators for the following requests were implemented at this stage:

- GET info for all devices
- GET info for a single device
- GET data for a single device
- GET history for a single device
- PUT a single data entry for a single device

7.2.4 Initial Visualiser Development – React Hooks

The initial functionality of the Visualiser service was completed in the medium-fidelity prototype. Now I needed to integrate the functionality of the visualiser by fetching data from the API when required. Before continuing I created two React hooks to query the API, so that I would not have to repeat the logic in other components multiple times.

```
/**
 * React hook to send an API request to the given URL
 * Returns the response sent by the API once the request is completed
 * @param url The URL to send a request to
 */
export function useAPI<Body=any>(url: string) {
  const [response, setResponse] = useState<APIResponse<Body>>();

  useEffect(() => {
    apiFetch<Body>(url).then((res) => {
      setResponse(res);
    });
  }, [url]);

  return response;
}
```

Figure 23 Visualiser service useAPI hook

The apiFetch function is a wrapper function around the standard web browser fetch function, it validates that the request returned JSON and returns an object containing the body of our API request. Additionally, I developed a useAPISubscription hook which periodically refreshes the result of the query and returns the new value.

```
/**
 * React hook to subscribe to an API request to the given URL.
 * Refreshes the result every 10 seconds by default
 * @param url The URL to query
 * @param interval The interval (in ms) to refresh the results after (default 10000)
 * @returns The results
 */
export function useAPISubscription<Body=any>(url: string, interval: number=10000) {
  const [response, setResponse] = useState<APIResponse<Body>>();

  // Set interval to regularly refresh the response value
  useEffect(() => {
    // Function to be used to query API
    const fetchFunc = () => {
      apiFetch<Body>(url).then((res) => {
        setResponse(res);
      });
    };

    const intervalID = setInterval(() => {
      fetchFunc();
    }, interval);
    fetchFunc();

    // Cleanup function to remove interval
    return () => { clearInterval(intervalID) };
  }, [url, interval]);

  return response;
}
```

Figure 24 Visualiser service useAPISubscription hook, periodically updates the response from the API

These hooks use the useEffect and useState hooks which are included in React. The useState hook allows us to store a value between function calls, and the useEffect hook allows us to re-run a function whenever one of its dependency's changes. With these hooks created, we are ready to continue developing the visualiser service.

7.2.5 Further Three.js Work

7.2.5.1 Loading Devices

Devices are loaded into the model using an API request using the `apiFetch` wrapper function mentioned in the previous section. We are unable to use the React hooks for this task as the Three.js code is not directly bound to React as mentioned earlier in this section. The devices are loaded at the same time as the 3D model is loaded and the code below is used.

```
const devices = (await apiFetch<Device[]>("/api/devices")).body;  
this.setDevices(devices);
```

Figure 25 API request to load devices

The `setDevices` method then creates the mesh for each device and puts them in a scene separate from the main 3D model so that we can run ray casts to determine when a user hovers over and selects one.

```
/**  
 * Add the given devices to the current scene for rendering  
 * @param devices The devices to add to the scene  
 */  
setDevices(devices: Device[]) {  
  // Stop if devices is empty or undefined  
  if (!devices) return;  
  
  // Clear existing devices  
  this.deviceScene.clear();  
  
  // Create geometry for each device  
  const geom = DEVICE_GEOM;  
  for (const device of devices) {  
    const mat = new THREE.MeshPhongMaterial({color: DEVICE_COLOR});  
    const cube = new THREE.Mesh(geom, mat);  
    cube.position.set(device.position.x, device.position.y, device.position.z);  
    cube.userData = device;  
    this.deviceScene.add(cube);  
  }  
  
  // Add lighting so the devices look nicer  
  const ambientLight = new THREE.AmbientLight(0x404040);  
  const light = new THREE.DirectionalLight(0xf4f4f4);  
  light.position.set(-100, 100, -100);  
  this.deviceScene.add(light, ambientLight);  
}
```

Figure 26 Method for creating device geometry

7.2.5.2 Slicing the Model

Three.js makes this task easy as we can provide a clipping plane to the scene renderer to easily slice through the rendered content where we want. This clipping plane then has its height adjusted to the correct value when we call the setFloor method, hiding the unrequested parts of the model automatically with no need to split the building model and devices by floor.

```
/**
 * Method moves the clipping plane to hide floors above the given floor number.
 * @param floor Integer representing the floor number
 */
setFloor(floor: number) {
    // Constants controlling floor height
    const perFloor = 12.5;
    const base = 12.5;

    // Clamp floor value
    floor = Math.min(Math.max(floor, 0), 7);

    // Calculate clipping height
    let val = base + perFloor*floor;
    this.clippingPlane.set(this.clippingPlane.normal, val);
}
```

Figure 27 Scene slicing method, takes a floor number as argument and slices the model using a clipping plane set to the height of that floor

7.2.6 Generating Test Data

To proceed with development, I needed some test data to work with and display within the application. Instead of using real-world data, which could be time-consuming to source, I decided to generate some randomly. To further this goal, I added a script to my API implementation. This script generates random data using noisy sine waves and square waves.

```
noisySineWave TypeScript function

/**
 * Function to generate a sine wave with some random noise.
 * Used as part of the random data generation script.
 *  $f(x) = A \sin(\omega t + p)$ 
 * @param length The length of array to generate the sine wave over
 * @param amp Amplitude of the sine wave
 * @param freq Frequency of the sine wave
 * @param noise Factor to multiply random noise by
 * @returns The sine wave extended over an array of the given length
 */
function noisySineWave(length: number, amp=1, freq=1, noise=0.2) {
    const result = new Array<number>(length);

    for (let i = 0; i < result.length; i++) {
        result[i] = (amp * Math.sin(freq*(i/180)));
        result[i] = result[i] + (Math.random() * noise) - noise/2;
    }
    return result;
}
```

Figure 28 noisySineWave function in TypeScript, used to generate random data for testing

I then utilised this function to insert data into the database for all devices. This gives each device a 'temperature', 'light level', 'humidity' and 'CO2' field which we can use the test the API and visualiser for the remainder of development.

7.2.7 Graphing Data

One of the most important specification points for this application is the ability to graph historical data within the application. For this purpose, I have decided to utilise the library Recharts. Recharts is “*a composable charting library built on React components*” (recharts, 2022). It is highly customisable, integrates very well into React and allows us to easily build highly customised charts with hover tooltips.

I eventually settled on the style seen below after experimenting with some different colours and line styles but changing them at a future point in time would be a trivial task.

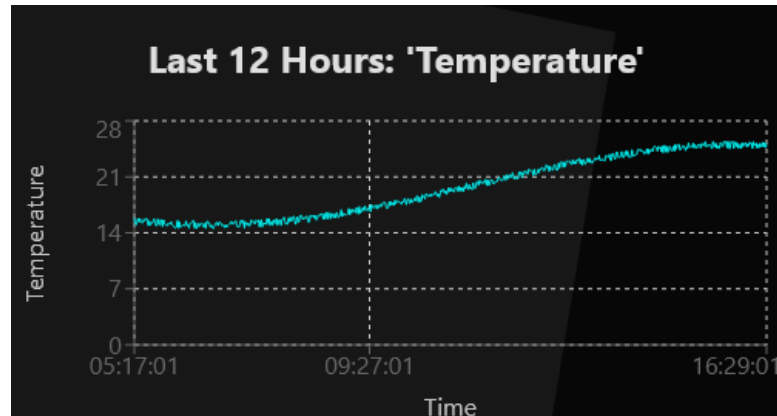


Figure 29 Visualiser basic chart style

7.2.8 Survey

After developing the medium fidelity design, I decided it would be useful to run a survey to ascertain the usability of the application so far, and as to what improvements I can make while proceeding to the high-fidelity prototype.

7.2.8.1 Survey Design

I decided to use a series of Likert scale questions to ascertain how usable the final application is. The questions I chose to be answered are based on standard SUS (System Usability Survey) questionnaires, but with a few changes. The statements answered in my questionnaire are as follows:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
The interface is intuitive and easy to navigate	☒	☒	☒	☒	☒
The system is over-complicated and difficult to use	☒	☒	☒	☒	☒
I would like to use this system frequently	☒	☒	☒	☒	☒
The system is cumbersome to use	☒	☒	☒	☒	☒
It is easy to distinguish between the devices and model of the building while navigating	☒	☒	☒	☒	☒
The building model is easy to navigate to find a particular device	☒	☒	☒	☒	☒
Viewing and visualising the data is intuitive	☒	☒	☒	☒	☒
I felt confident using the system	☒	☒	☒	☒	☒

After the Likert scale questions, I then asked the responders if they had any additional comments or suggestions. This will allow them to provide reasoning for their rating of the Likert scale questions if they wish - helping me act on any feedback they may provide and introduce any new features or improvements to enhance the final high-fidelity design. The survey was targeted at Computer Science students in all years. I chose this group as they represent a more technically inclined group of people who might use an application such as this.

7.2.8.2 Survey Results

Likert Scale

To help quantify the results of the Likert Scale questions, I have applied a score for each question. The most positive response is given a value of 5, whereas the least positive is given a score of 1. This means that for each respondent we can generate an overall score of what they thought about the application. Ideally, I would use the SUS formula here, but due to my changes to the SUS questionnaire the produced result would not be valid.

Participant ID	Score (max 40)	Percentage score
3	38	95%
4	29	72.5%
5	27	67.5%
6	18	45%
7	31	77.5%
8	37	92.5%
9	26	65%
10	21	52.5%
11	33	82.5%
Average (mean)	28.888	72.2%

Additional Comments

One issue with the application is that it provides no context for what the data from the devices really means. This is highlighted by one of the comments left by a responder: *“There needs to be some indicators to help with orientation such as what a room is used for, i.e., Lecture theatre, computer room, meeting room, office, study, etc., for the data to have any meaning”* (participant 6). This tells us that additional context and metadata related to the devices would be useful to display.

Another respondent pointed out that requesting data from individual devices is all well and good, but to provide better support to users requesting data from the entire building some more advanced query features would be ideal. These features should include being able to filter the devices by the data fields that they have, as well as other metadata fields such as the floor the devices are on and the type of room the device occupies.

7.2.8.3 Survey Issues

The SUS would have been a good choice to evaluate this program. But as the questions I have asked here are non-standard, this reduces the ability of this survey to be compared to any other similar survey. The scores generated from this questionnaire cannot be compared to other standard SUS questionnaires due to the changes made to the questions that make up the questionnaire. Usually, the questions on a SUS questionnaire are as follows:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

I did not ask questions 4, 5, 6 and 10. I instead asked about the ease of use for viewing the model, which allowed me to evaluate those components in isolation from the rest of the application.

However, as this survey is only intended to help improve the medium-fidelity prototype into a final high-fidelity prototype, comparison is of little utility here which is why these questions were omitted. It could be useful to run a SUS survey to ascertain whether this application makes improvements compared to the Newcastle Urban Observatory project that inspired it.

7.2.9 Design Changes Based on Survey Responses

The results of the survey make it apparent that the existing hiding and showing of the side bar is inconvenient for mobile devices and on smaller screens. Thus, I endeavoured to improve the experience on these platforms by introducing changes to the styles to make the side panel fill the screen when it was shown on small devices instead of only occupying a small portion of it.

To achieve this, we utilised CSS media queries to assign the correct styles with smaller screen sizes.



```
@media screen and (max-width: 500px) {  
  .data-container {  
    width: 100%;  
  }  
}
```

Figure 30 CSS media query for small screen devices

As I was implementing better support for mobile phones and small screens, it became apparent that the current method of showing and hiding the data panel was going to be insufficient. The arrow was too small to use on mobile devices and did not follow modern design patterns.

I decided instead to switch to a “hamburger menu” which is a commonly used design pattern for sidebars on mobile devices. This is a better choice than the previous design as it is a widely used and recognised icon, and so is likely to be immediately recognised and understood by users.

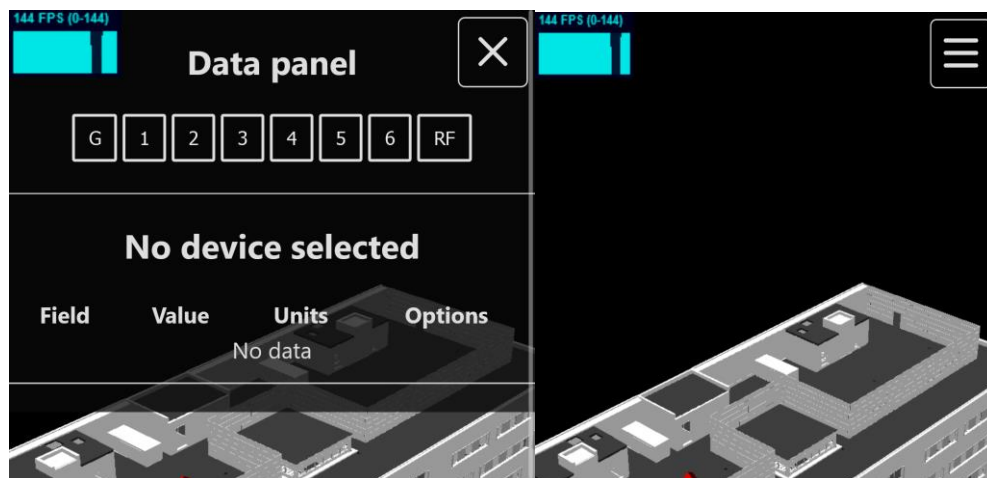


Figure 31 Data panel design with hamburger menu

This feedback received during the survey lead me to update the application to display the type and floor of the selected device in the visualiser. As the data was already present on the backend but just was not being used, adding this was simple and the results can be seen below. I also took this opportunity of adding better-defined table borders to make the table easier to read and more visually appealing.

Device: 'Room-2.47'			
Type: Office Floor: 2			
Field	Value	Units	Options
Timestamp	20/04/2022, 16:48:05	N/A	N/A
Temperature	24.74	°C	History
Light Level	10000	lux	History
Humidity	74	%	History
CO2	465	ppm	History

Figure 32 Type and floor displayed in visualiser after survey feedback

7.2.10 Improved Medium Fidelity Design

The final state of the medium-fidelity prototype before moving on to the high-fidelity prototype can be seen below.

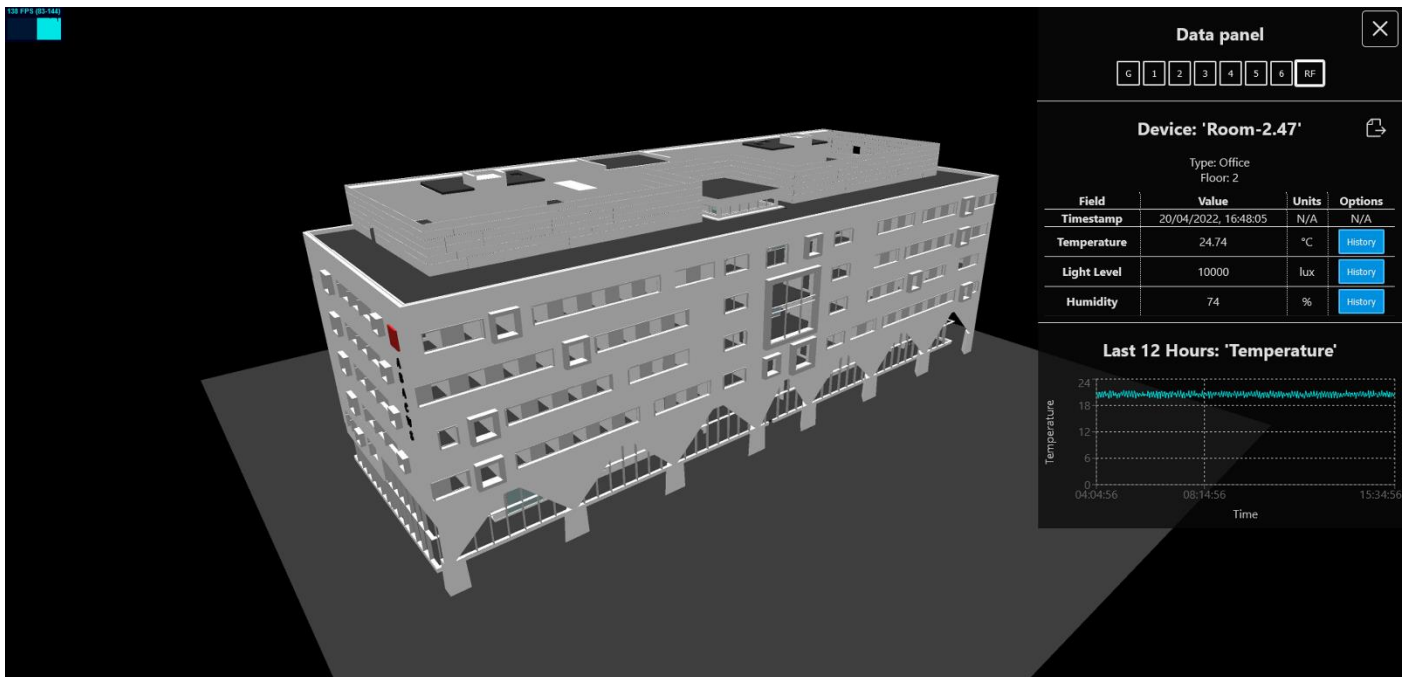


Figure 33 Initial state of the high-fidelity prototype after implementing the improvements to the medium-fidelity prototype suggested by the answers to the survey

Another feature requested during the survey was the inclusion of dataset wide querying. That is to say, the ability to query the API for data from all devices. This would include the addition of filters such as the names, types, and floors of devices. This would provide a way to easily access the entire dataset without querying individual devices. Therefore, I propose this addition shown below to the initial functional requirements. This addition will be referred to as “advanced query” in the rest of this report.

7.2.10.1 Users should be able to query the application for information on the entire dataset

Acceptance Criteria

- The application should provide endpoints that allow a user to query the entire dataset with all devices at once.
- Users should be able to filter by floor, device type, device name, and device data fields.

Assumptions

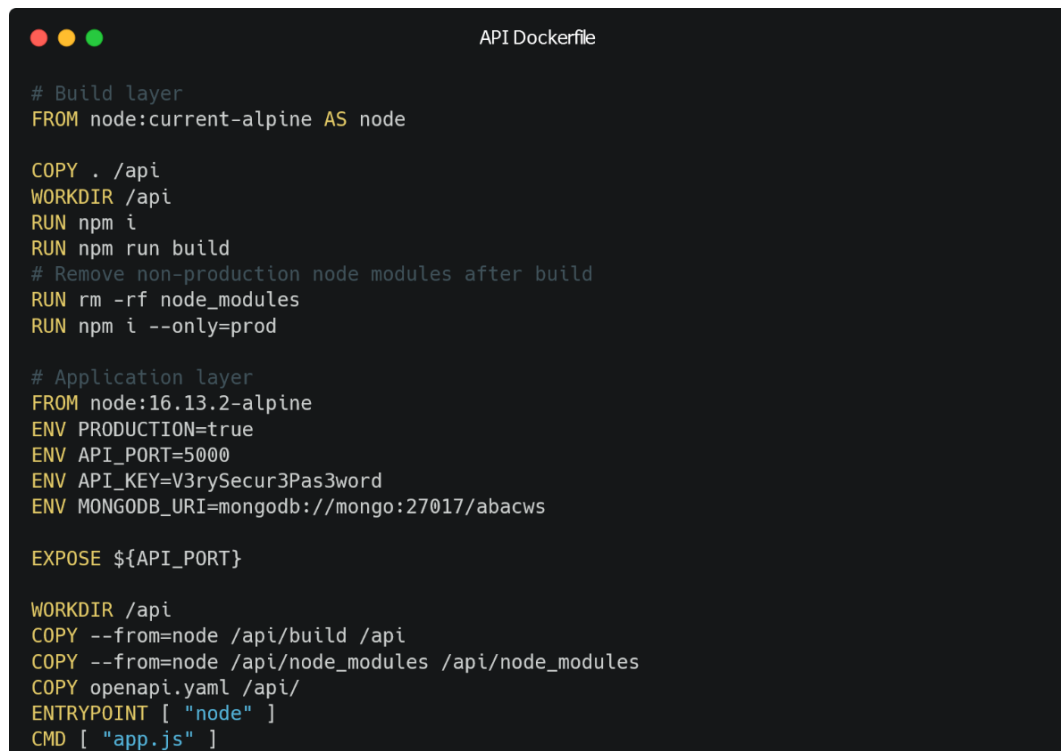
- There are devices registered with the application.
- There is data to query from the dataset.

7.3 HIGH FIDELITY PROTOTYPE

7.3.1 Docker

After setting up the initial functionality for the API and visualiser services, I decided to next work on building Docker images for both. Using GitHub actions, I would then be able to automatically build the images when I make changes (Continuous Integration (CI)) and deploy them on a web server (Continuous Deployment (CD)).

7.3.1.1 API Service

A screenshot of a code editor window titled "API Dockerfile". The code is written in a dark theme with syntax highlighting. It defines a multi-stage Docker build. The first stage, "# Build layer", starts with "FROM node:current-alpine AS node", copies the "/api" directory to "/api" in the container, runs "npm i", "npm run build", and then removes non-production node modules. The second stage, "# Application layer", starts with "FROM node:16.13.2-alpine", sets environment variables for "PRODUCTION=true", "API_PORT=5000", "API_KEY", and "MONGODB_URI", and exposes the API port. It then copies the built files from the first stage and sets the entrypoint to "node" and the command to "app.js".

```
# Build layer
FROM node:current-alpine AS node

COPY . /api
WORKDIR /api
RUN npm i
RUN npm run build
# Remove non-production node modules after build
RUN rm -rf node_modules
RUN npm i --only=prod

# Application layer
FROM node:16.13.2-alpine
ENV PRODUCTION=true
ENV API_PORT=5000
ENV API_KEY=V3rySecur3Pas3word
ENV MONGODB_URI=mongodb://mongo:27017/abacws

EXPOSE ${API_PORT}

WORKDIR /api
COPY --from=node /api/build /api
COPY --from=node /api/node_modules /api/node_modules
COPY openapi.yaml /api/
ENTRYPOINT [ "node" ]
CMD [ "app.js" ]
```

Figure 34 API service Dockerfile

Creating a Docker image from the API was simple as Node.js provides Docker images we can extend with our own files to run the API. Here we use the Alpine versions of node for the base image. Alpine Linux is “*a security-oriented, lightweight Linux distribution*” (Alpine, n.d.). It is commonly used with Docker as it produces much smaller images compared to Debian based images. For this reason, we use the Alpine version of the base image wherever we can.

7.3.1.2 Visualiser Service

Deployment of the visualiser service is slightly more complicated, as we must build the application using node (as we did with the API) which generates files which can then be served using a normal web server. For this purpose, we first use a node layer to build the service, then use NGINX to serve the files. Helpfully the NGINX image automatically substitutes environment variables into config files when it starts up, so we can still use these environment variables to configure it.

```
Visualiser Dockerfile

# Build layer
FROM node:current-alpine AS node

COPY . /app
WORKDIR /app
RUN npm i
RUN npm run build
# Remove non-production node modules after build
RUN rm -rf node_modules
RUN npm i --only=prod

# Application layer
FROM nginx:alpine
ENV WEB_PORT=80
ENV API_HOST=api:5000

EXPOSE ${WEB_PORT}

# Copy frontend build
WORKDIR /app
COPY --from=node /app/build /app

# Copy NGINX config template
COPY nginx.conf.template /etc/nginx/templates/default.conf.template
```

Figure 35 Visualiser service Dockerfile

```
NGINX template

server {
    listen ${WEB_PORT};
    server_name _;
    root /app;

    location / {
        try_files $uri /index.html =404;
    }

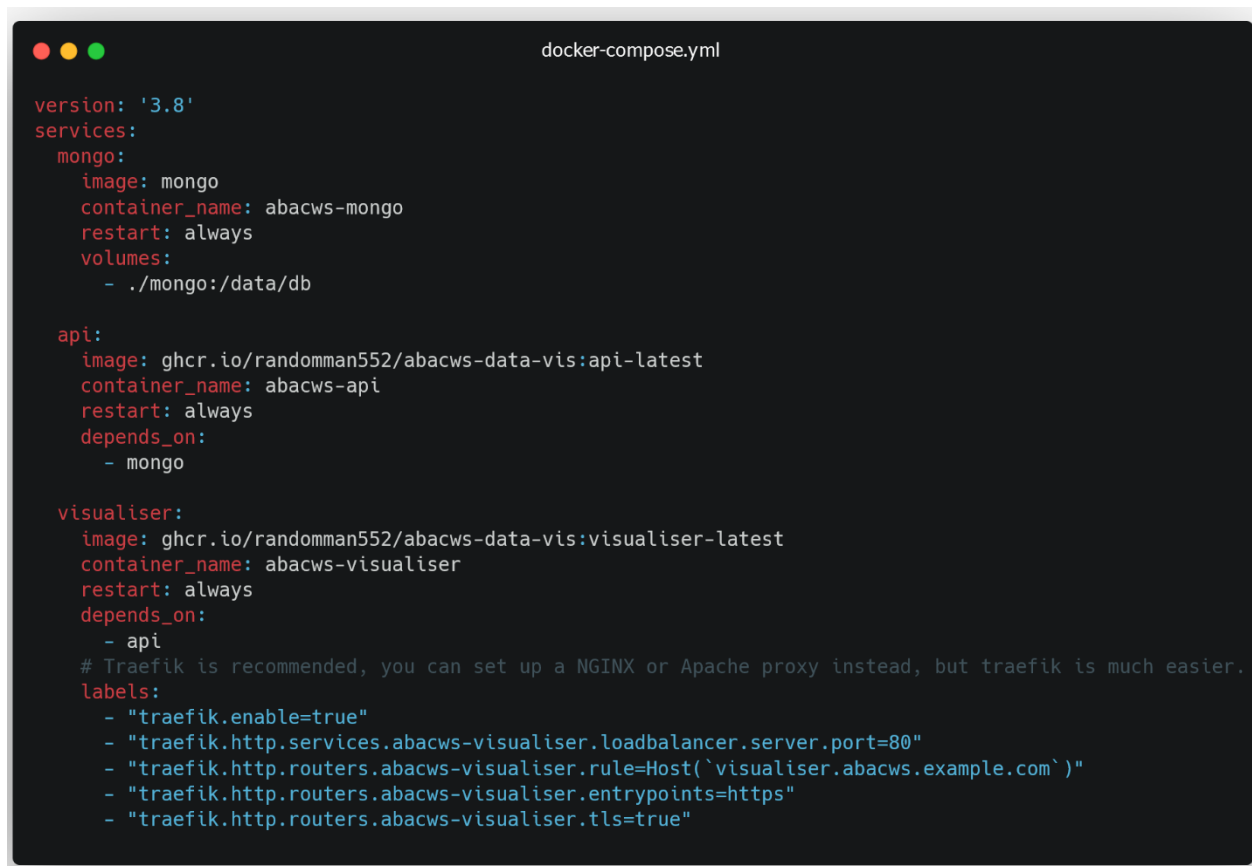
    location /api {
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_pass http://${API_HOST}/;
    }
}
```

Figure 36 Visualiser service NGINX template

This NGINX template proxies requests to /api to the location stored in the API_HOST environment variable. This means that in the visualiser service we need only direct our requests to /api and the NGINX proxy deals with the rest.

7.3.1.3 Docker-Compose

The entire application can be deployed using the following docker-compose file.



```
version: '3.8'
services:
  mongo:
    image: mongo
    container_name: abacws-mongo
    restart: always
    volumes:
      - ./mongo:/data/db

  api:
    image: ghcr.io/randomman552/abacws-data-vis:api-latest
    container_name: abacws-api
    restart: always
    depends_on:
      - mongo

  visualiser:
    image: ghcr.io/randomman552/abacws-data-vis:visualiser-latest
    container_name: abacws-visualiser
    restart: always
    depends_on:
      - api
    # Traefik is recommended, you can set up a NGINX or Apache proxy instead, but traefik is much easier.
    labels:
      - "traefik.enable=true"
      - "traefik.http.services.abacws-visualiser.loadbalancer.server.port=80"
      - "traefik.http.routers.abacws-visualiser.rule=Host(`visualiser.abacws.example.com`)"
      - "traefik.http.routers.abacws-visualiser.entrypoints=https"
      - "traefik.http.routers.abacws-visualiser.tls=true"
```

Figure 37 Example docker-compose.yml file

The labels assigned to the visualiser service handle routing for the Traefik proxy. This proxy could be replaced with NGINX or Apache instead, if this application were not hosted alongside Traefik.

7.3.1.4 Continuous Integration (CI)

Continuous Integration (CI) refers to the process of running builds or tests whenever a developer contributes to the project. As we are using GitHub for the hosting of our project, we can utilise GitHub actions to automatically build the Docker images defined above when we commit changes. I utilised this throughout the creation of the project to make it easier to deploy for testing.

GitHub actions supports some community provided actions for building and pushing Docker images, which simplifies much of our work.

A screenshot of a GitHub Actions workflow script. The script is displayed in a dark-themed editor window titled 'GitHub Actions'. The script defines a workflow named 'Push API Docker' that triggers on pushes to the 'main' branch or tags starting with 'v'. It includes a 'build' job that runs on 'ubuntu-latest' and consists of five steps: 1. Checkout repository using 'actions/checkout@v2'. 2. Login to GHCR using 'docker/login-action@v1' with 'ghcr.io' as the registry and GitHub actor/token for authentication. 3. Extract metadata (tags, labels) for Docker using 'docker/metadata-action@v3' to generate image names like 'ghcr.io/{actor}/abacws-data-vis-api-latest'. 4. Build and push the Docker image using 'docker/build-push-action@v2' with the context set to './api' and pushing to the registry. 5. The final output is a Docker image pushed to the GHCR registry.

```
name: Push API Docker
on:
  push:
    branches: [ main ]
    tags:
      - 'v*'
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v2

      - name: Login to GHCR
        uses: docker/login-action@v1
        with:
          registry: ghcr.io
          username: ${github.actor}
          password: ${secrets.GITHUB_TOKEN}

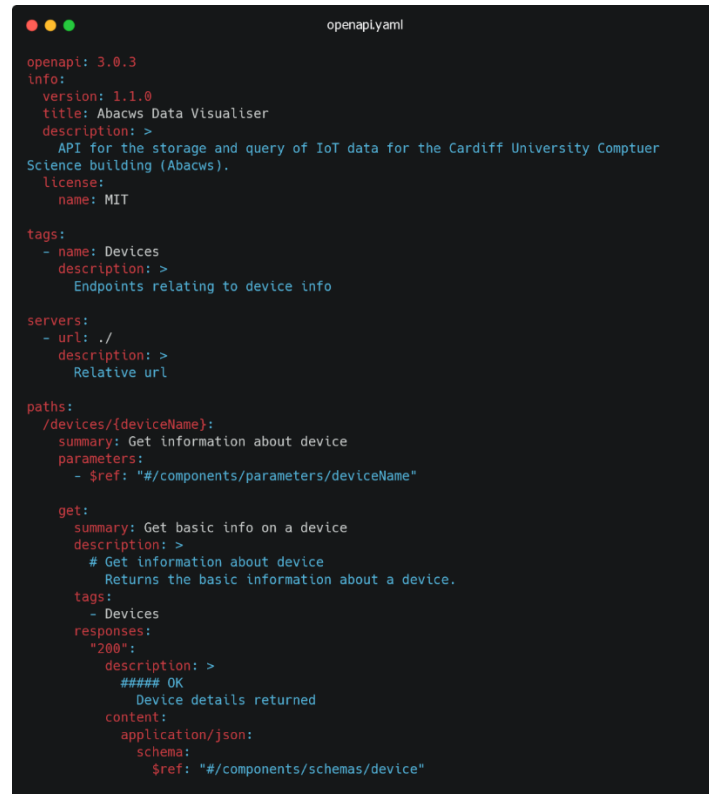
      - name: Extract metadata (tags, labels) for Docker
        id: meta
        uses: docker/metadata-action@v3
        with:
          images: ghcr.io/${github.actor}/abacws-data-vis
          flavor: |
            prefix=api-,onlatest=true

      - name: Build and push
        uses: docker/build-push-action@v2
        with:
          context: "./api"
          push: true
          tags: ${steps.meta.outputs.tags}
          labels: ${steps.meta.outputs.labels}
```

Figure 38 GitHub actions script to build and push the API Docker image

7.3.2 Self-Documenting API

The OpenAPI 3.0 Specification (OAS) was discussed in the background section of this report. This specification allows for the creation of programming-language agnostic human- and machine-readable documentation for API's. It is supported by a wide variety of development and testing tools and as such I decided to adopt it to make my API "self-documenting".

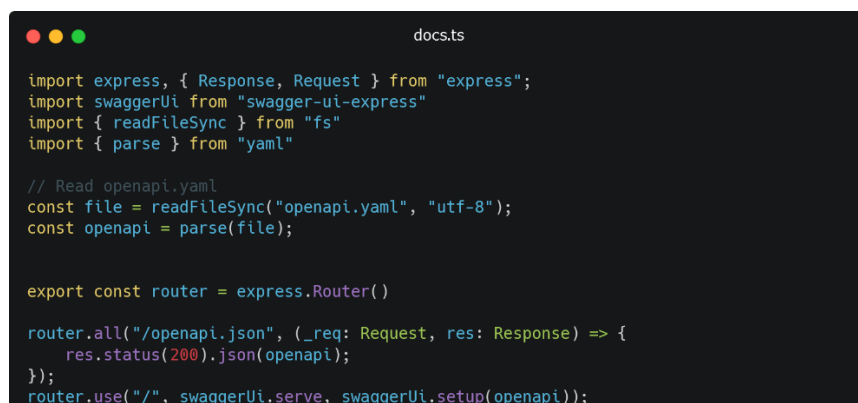
A screenshot of a code editor window titled 'openapi.yaml'. The file contains an OpenAPI 3.0.3 specification for 'Abacws Data Visualiser'. It includes fields for version (1.1.0), title, description, license (MIT), tags (Devices), servers (relative url), and a path '/devices/{deviceName}' with a GET method. The response is a 200 status code with a JSON body containing device details.

```
openapi: 3.0.3
info:
  version: 1.1.0
  title: Abacws Data Visualiser
  description: >
    API for the storage and query of IoT data for the Cardiff University Computer
    Science building (Abacws).
  license:
    name: MIT
tags:
  - name: Devices
    description: >
      Endpoints relating to device info
servers:
  - url: ./
    description: >
      Relative url
paths:
  /devices/{deviceName}:
    summary: Get information about device
    parameters:
      - $ref: "#/components/parameters/deviceName"
    get:
      summary: Get basic info on a device
      description: >
        # Get information about device
        Returns the basic information about a device.
      tags:
        - Devices
      responses:
        "200":
          description: >
            ##### OK
            Device details returned
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/device"
```

Figure 39 Sample of the openapi.yaml file defining the API.

By "self-documenting" I mean that the API should host its own documentation as part of its function. Making it easy for users to find and use the API when they need to, without having to refer to external documentation services.

For this purpose, we can turn to the library swagger-ui-express (scottie1984, 2022) to automatically generate human-readable documentation for our express based API, based on the OAS file we use to define it. This module also provides the ability to query the API endpoints from within it, making it a useful testing tool.

A screenshot of a code editor window titled 'docs.ts'. The file contains TypeScript code that imports express, swaggerUi, and fs. It reads the 'openapi.yaml' file and parses it. Then, it creates an express router and serves the swaggerUi documentation at the root path, using the parsed openapi specification.

```
docs.ts

import express, { Response, Request } from "express";
import swaggerUi from "swagger-ui-express";
import { readFileSync } from "fs";
import { parse } from "yaml";

// Read openapi.yaml
const file = readFileSync("openapi.yaml", "utf-8");
const openapi = parse(file);

export const router = express.Router()

router.all("/openapi.json", (_req: Request, res: Response) => {
  res.status(200).json(openapi);
});
router.use("/", swaggerUi.serve, swaggerUi.setup(openapi));
```

Figure 40 swagger-ui-express used to serve OAS documentation

As I was developing the rest of the API service for this project, I used this file (openapi.yaml) to plan what each new endpoint would need to do, what sort of data it would require, and what it would return. I could then use tools such as Postman which support the building of testing suites from an OAS file to verify that the code I had written for the endpoint returned what I was expecting. Doing this improved the quality of the documentation and the API itself.

7.3.3 Advanced Query Features

One of the requirements laid out in the specification for this application was that anyone should be able to query the entire dataset using an “advanced” query endpoint. In this section I will discuss the implementation of this endpoint and the features it presents.

Initially I developed only the API endpoints for this, but I realised that this made it hard to access the functionality for less API-inclined users. So, I created a form within the visualiser service that enables users to query these advanced endpoints without having to understand how the API functions, or how to query it.

7.3.3.1 API Implementation

Firstly, I developed the API implementation for the advanced queries. The API needed to take the following parameters from the user (as defined in the acceptance criteria for this requirement):

- Query type: The type of data we want to retrieve (info, data, or history).
- Device name (optional): The name(s) of the devices we want to query.
- Device type (optional): The type(s) of devices we want to query.
- Device floor (optional): The floor(s) devices we want to query are on.
- Data fields (optional, only for data and history query types): The fields we want the data from the devices to contain.
- Time range (optional, only for history query type): The time range of data we want. Any data outside of this range should be discarded, this parameter includes two parameters, ‘from’ and ‘to’.

The fields above (with exception of query type) can all have one or more values. The method used to achieve this was to allow the user to provide a comma-delimited list, which we then split in the API when received. E.g., “name, name1” would be split into [“name”, “name1”] by the API which would then perform queries on the dataset.

As the query type is the only non-optional parameter provided, it is included in the path of the URL when querying the API. This creates three sub-paths, each of which corresponds to a different type of query. Before proceeding I codified these additional endpoints into the OpenAPI specification for the application. All the parameters are passed as query parameters in the URL as they are the most easily accessible.

GET	/query	Advanced query endpoint for device info	▼
GET	/query/data	Advanced query endpoint for device data	▼
GET	/query/history	Advanced query endpoint for device history	▼

Figure 41 Swagger OpenAPI docs for the advanced query endpoints

There is duplicate functionality between these query endpoints. Namely all the query endpoints take a 'name', 'type', and 'floor' parameter and the data and history endpoints both take the 'fields' parameter. Here we can utilise a useful feature included in express called middleware. This feature allows us to define a function to handle the common functionality between the query endpoints before proceeding to their individual functionality.

Name	Description
name string (query)	The names of the devices we wish to query. • Can be a single value, or a list of comma-delimited values name
type string (query)	Types of devices we wish to query. • Can be a single value, or a list of comma-delimited values type
floor string (query)	Floors on which the devices we wish to query are • Can be a single value, or a list of comma-delimited values floor
has string (query)	The fields we want the returned data to contain. • Devices and data not containing these fields will not be returned. • Can be a single value, or a list of comma-delimited values has

Figure 42 OpenAPI documentation for parameters passed to advanced query endpoints (except for the 'history' endpoint)

```

/api/src/api/middleware/query.ts

/**
 * Middleware for advanced query endpoints.
 * Forms the basis of all requests to '/query' endpoint.
 * Obtains the info of the documents that were requested and places them in res.locals.devices.
 * This middleware does not return any response by itself, but instead calls next to allow further processing.
 */
export async function queryMiddleware(req: Request, res: Response, next: NextFunction) {
  // Get the query parameters
  const name = req.query.name?.toString().split(",");
  const type = req.query.type?.toString().split(",");
  // Floor has to be converted to numbers here
  const floor = req.query.floor?.toString().split(",").map((value) => { return Number(value) });

  // Create filter from passed parameters
  const filter: any = {};

  if (name)
    filter.name = {
      "$in": name
    }
  if (type)
    filter.type = {
      "$in": type
    }
  if (floor)
    filter.floor = {
      "$in": floor
    }

  const devices = await client.db().collection("devices")
    .find(filter)
    .project({ _id: 0 })
    .toArray();

  // Store result for future use
  res.locals.devices = devices;
  // Call next endpoint handler
  return next();
}
```

Figure 43 Advanced query middleware handling common functionality of all advanced query endpoints

We to split each of the parameters into an array and then build a MongoDB query with a filter to send to the database to get the relevant devices. After getting the devices, we store them for later use in the response object, which is passed to the next function which returns the final response.

```

/api/src/api/routers/query.ts

/**
 * Advanced query endpoint that returns device info and each device's most recent data entry
 * Methods: GET
 */
async function queryData(req: Request, res: Response) {
  const has = req.query.has?.toString().split(",");

  // Build data filter and projection for mongodb request
  const [filter, projection] = buildDataQueryParams(has);

  // Get devices from middleware
  const devices = res.locals.devices as Document[];

  // Query the database for the last data entry for each device
  for (const device of devices) {
    const collection = client.db().collection(`${DEVICE_COLLECTION_PREFIX}_${device.name}`);
    device.data = await collection.findOne(filter, { projection });
  }

  // Filter out any devices without data
  const filtered = devices.filter(device => device.data)

  res.status(200).json(filtered);
}
```

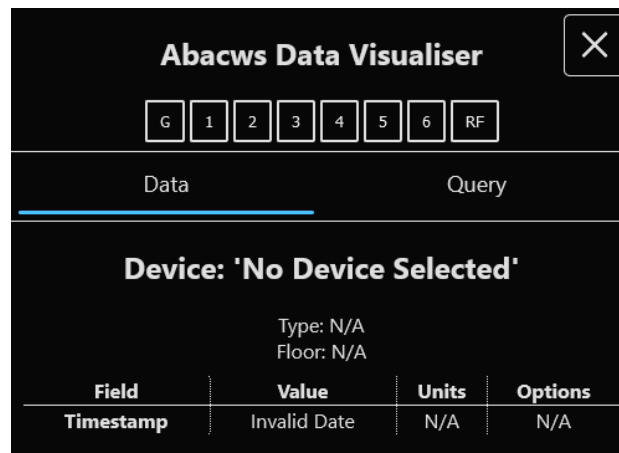
Figure 44 Advanced query data endpoint, uses the query middleware and then adds the most recent data entry for each device onto the device info

The function shown above handles the advanced query endpoint for data. This is very similar to the one handling history, which instead queries the database for all the previous data entries for each device. The history endpoint also takes a 'to' and 'from' parameter specifying the time range of data that is allowed.

7.3.3.2 Visualiser Query Form

With the functionality of the API out of the way, it is now time to add a form to the visualiser so that end users do not need to manually perform requests. To start with, I needed a place to put this form that would not interfere with the existing interface and functionality.

I decided to make the side menu, currently containing only the device details, into a tabbed interface. This ensures that additions made in this secondary tab will not impact existing functionality but will provide a place to put the new form.



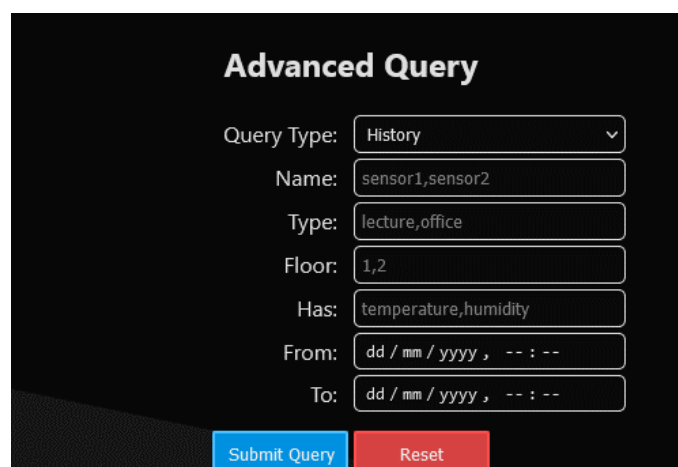
The screenshot shows the 'Abacws Data Visualiser' interface. At the top, there's a title bar with a close button. Below it, a row of buttons labeled 'G', '1', '2', '3', '4', '5', '6', and 'RF'. A tabbed interface follows, with 'Data' and 'Query' tabs; 'Data' is active, indicated by a blue underline. The main content area for the 'Data' tab displays 'Device: No Device Selected' and 'Type: N/A', 'Floor: N/A'. Below this is a table with the following structure:

Field	Value	Units	Options
Timestamp	Invalid Date	N/A	N/A

Figure 45 New tabbed interface separating data panel from query panel. The active tab is highlighted with a blue underline to let the user know which they have selected

The query form allows the user to choose between 'info', 'data' and 'history' query modes using a dropdown selection menu. The fields appropriate for the query time they have selected are they displayed below that. Irrelevant input fields are not shown to the user, as they might cause confusion. Each input field has a placeholder entry to show the user what sort of data it accepts.

The form is organised using the left/right alignment design pattern to make it clear which labels are relevant to each form input. Upon submitting the form, the query is downloaded in JSON format directly from the API.

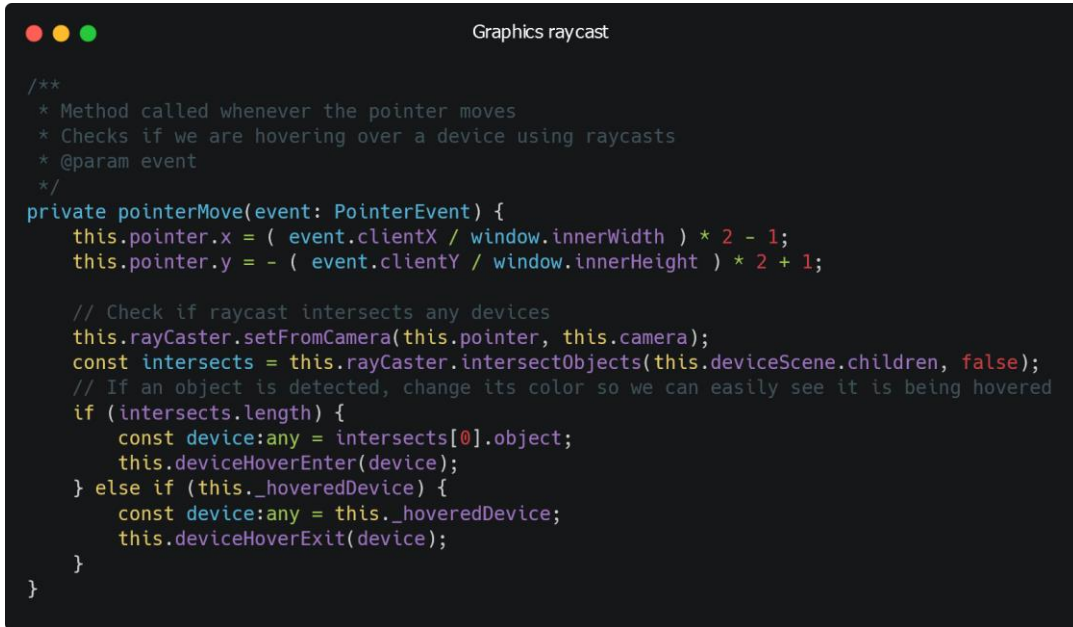


The screenshot shows the 'Advanced Query' form. It features a title bar and several input fields arranged in a left-aligned pattern. The fields are: 'Query Type' (a dropdown menu set to 'History'), 'Name' (text input with 'sensor1,sensor2'), 'Type' (text input with 'lecture,office'), 'Floor' (text input with '1,2'), 'Has' (text input with 'temperature,humidity'), 'From' (text input with 'dd / mm / yyyy , -- : --'), and 'To' (text input with 'dd / mm / yyyy , -- : --'). At the bottom, there are two buttons: 'Submit Query' (blue) and 'Reset' (red).

Figure 46 Query form layout

7.3.4 Device Selection

To allow the user to select a device from the 3D model, we use a ray cast to project into the scene whenever the user moves the pointer around on the canvas. The method shown below is attached to the canvas and listens for pointer move events. It then takes the new position of the pointer and finds out if a device is being hovered over by using ray casting.



```
Graphics raycast

/**
 * Method called whenever the pointer moves
 * Checks if we are hovering over a device using raycasts
 * @param event
 */
private pointerMove(event: PointerEvent) {
  this.pointer.x = ( event.clientX / window.innerWidth ) * 2 - 1;
  this.pointer.y = - ( event.clientY / window.innerHeight ) * 2 + 1;

  // Check if raycast intersects any devices
  this.rayCaster.setFromCamera(this.pointer, this.camera);
  const intersects = this.rayCaster.intersectObjects(this.deviceScene.children, false);
  // If an object is detected, change its color so we can easily see it is being hovered
  if (intersects.length) {
    const device:any = intersects[0].object;
    this.deviceHoverEnter(device);
  } else if (this._hoveredDevice) {
    const device:any = this._hoveredDevice;
    this.deviceHoverExit(device);
  }
}
```

Figure 47 Three.js ray cast to find the device the user is currently hovering over when the move the mouse

Once we know the device the user is hovering over, we can alter its appearance to make it obvious to them that they are hovering over the device. When a pointer down event is next fired on the window, we make the hovered device the selected one. We then alter the appearance of the selected device so that it is obviously selected.



```
/**
 * Method called whenever a pointer down event occurs on the window.
 * Used to select a device when we click on it
 * @param event
 */
private pointerDown(event: PointerEvent) {
  this.pointerMove(event);
  if (this._hoveredDevice && event.button === 0)
    this.devicePointerDown(this._hoveredDevice);
}
```

Figure 48 Code to set the selected device when a pointer down event occurs

Below you can see the different appearances of the device nodes when inactive, hovered, and selected respectively. The differences in appearance are important as it lets the user know visually what device they have selected instead of them just seeing the devices name in the data panel.

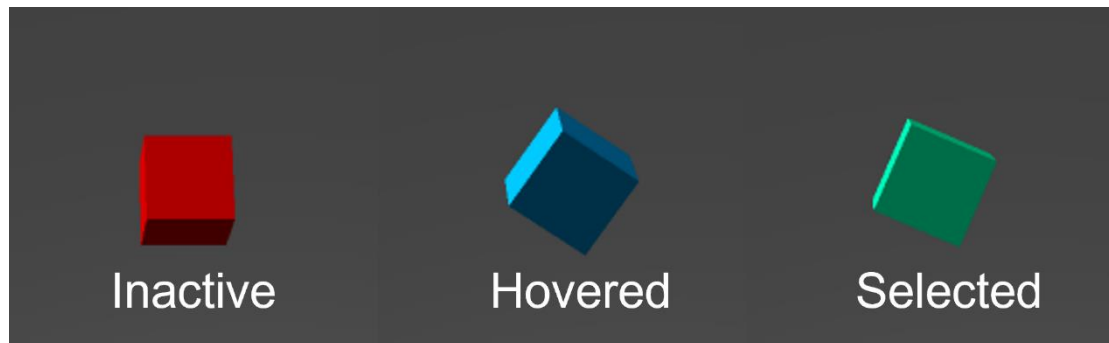


Figure 49 Difference between unselected, hovered, and selected device nodes in the visualiser

8 RESULTS AND EVALUATION

8.1 RESULTS

Some screenshots of the final produced web application are shown below, there is a section for the visualiser service and API service. Both services are hosted together using Docker on a Linux based web server. The deployment is hosted publicly at <https://abacws.ggrainger.uk/> if you want to experience the application for yourself.

A demo of the application can be viewed [here](#), and a discussion of the code can be viewed [here](#).

8.1.1 Visualiser



Figure 50 Screenshot of visualiser after application loads

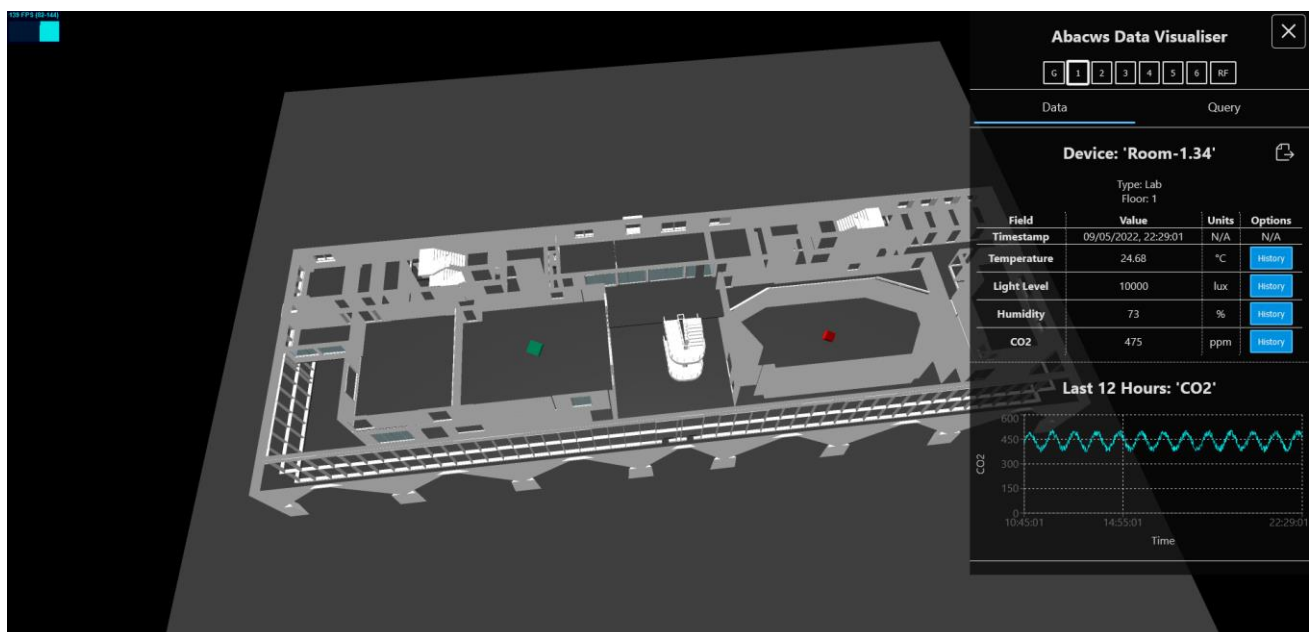


Figure 51 Screenshot of visualiser after selecting a device and plotting historical data for the 'CO2' field

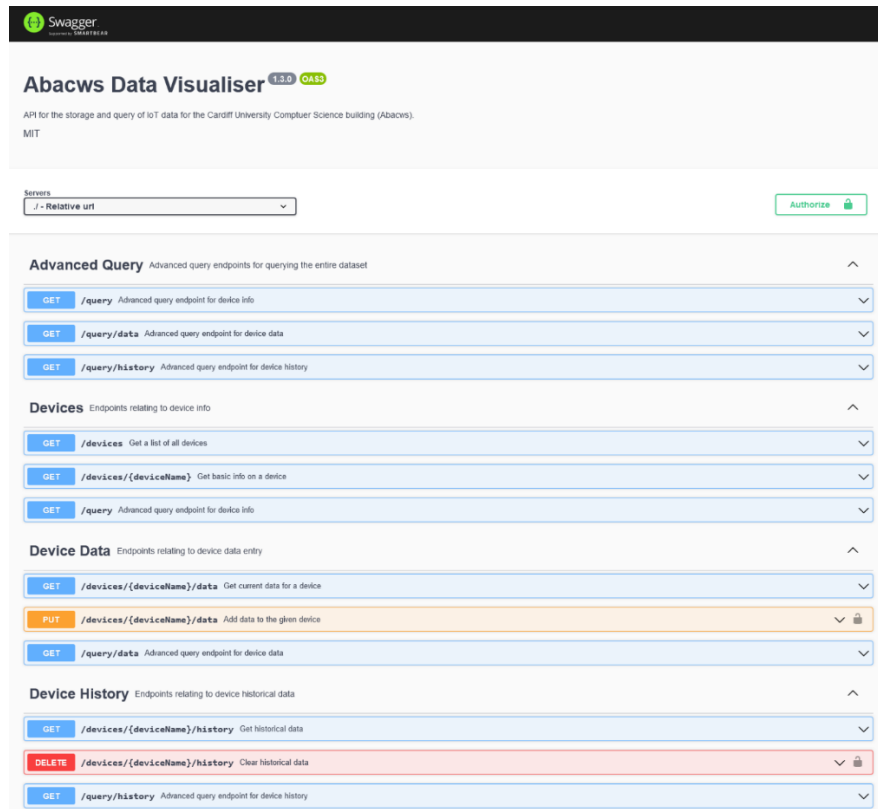


Figure 52 API self-documentation

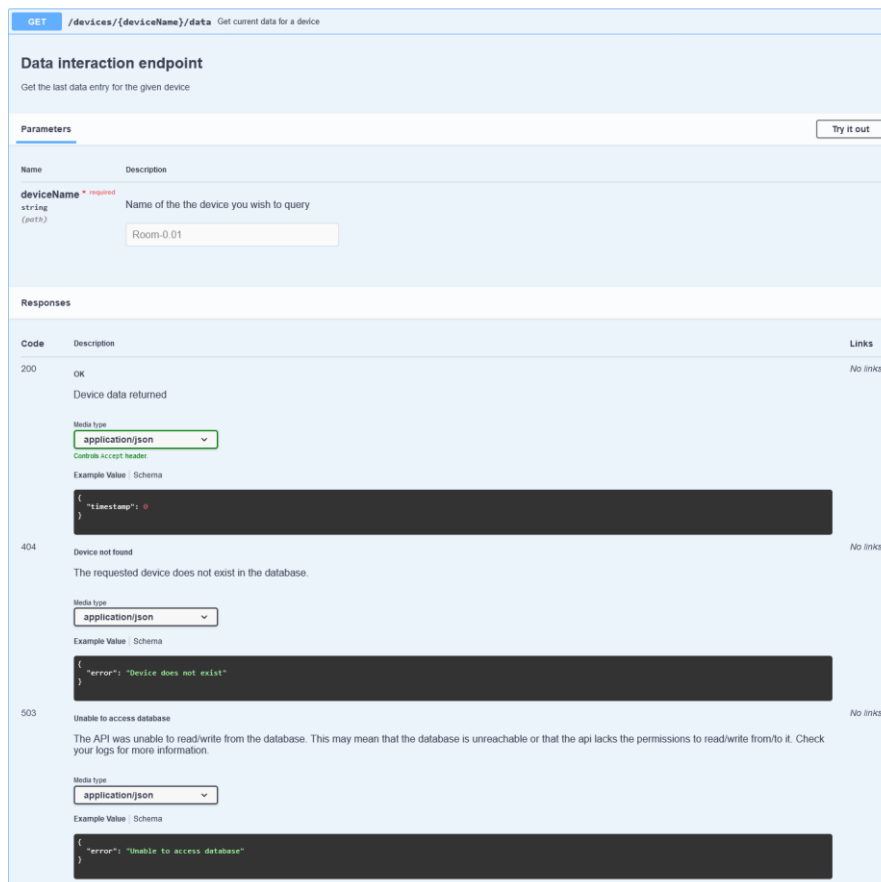


Figure 53 Example of the documentation provided by the API. This is for the device data endpoint; all endpoints are documented in the same way.

8.2 EVALUATION

Overall, I have successfully met most of the specification points I set out in the specification and design section of this report. There are some minor bugs to fix and enhancements that could be made to the application before it would be fully deployed in a real-world scenario.

In the section below I have conducted tests against the specification created at the beginning of this project. Where certain acceptance criteria have failed, I have discussed why that point was failed, and how important it is to the functioning of the software.

8.2.1 Evaluation against Requirements

In this section I will go through the requirements I laid out at the start of this report, and will discuss which requirements have been fulfilled, and which (if any) have been failed or partially met. Any fulfilment or failure will be evidence with screenshots from the application and will be given a **PASS**, **PARTIAL**, or **FAIL**.

8.2.1.1 Devices must be able to add data to the application ✓(PASS)

Device must be able to send a HTTP request to a web endpoint to add data to its representation.
Sending the following data in a request.



```
{
  "value": true
}
```

Figure 54 Data sent using PUT request to /api/{deviceName}/data

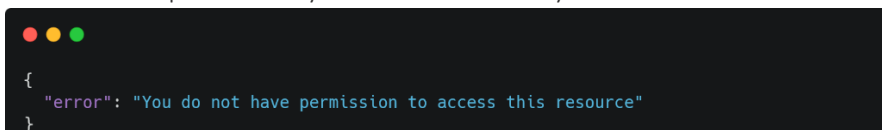
Results in a response with success code 202. Database now contains the data below. As you can see a timestamp entry has been added as there was not one present before.



```
{
  "_id": {
    "$oid": "6277f12483d99a93f6cccf66"
  },
  "value": true,
  "timestamp": 1652027683506
}
```

Figure 55 Database entry after sending data to API

An authorisation token must be provided by the device to carry out this task.



```
{
  "error": "You do not have permission to access this resource"
}
```

Figure 56 Response from the API when trying to add data without an authorisation token

8.2.1.2 User must be able to see a model of the building ✓(PASS)

User viewing the web application can view a 3D model of the Abacws building.



Figure 57 View of the model of the building

Building should be navigable using orbit controls.

Three.js comes with orbit controls which have been imported and used here. Note that this is hard to show in this document as it requires animation, but it does work.

8.2.1.3 User must be able to see devices represented within the building model ✓(PASS)

User viewing the application can see IoT devices represented within the model

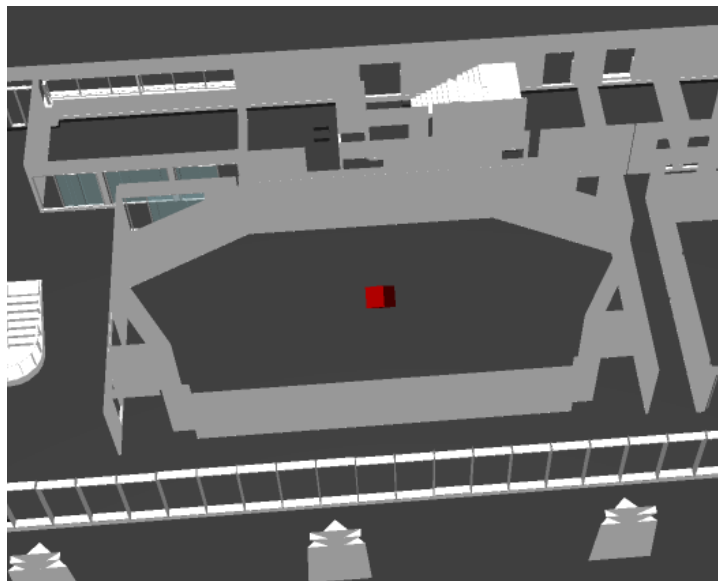


Figure 58 Devices within the application represented as spinning red cubes

8.2.1.4 User must be able to slice through the 3D model to view different floors ✓ (PASS)

Upon selecting a floor, the 3D model must be sliced to show no floors above the selected floor.



Figure 59 Floor selector before and after selecting a floor

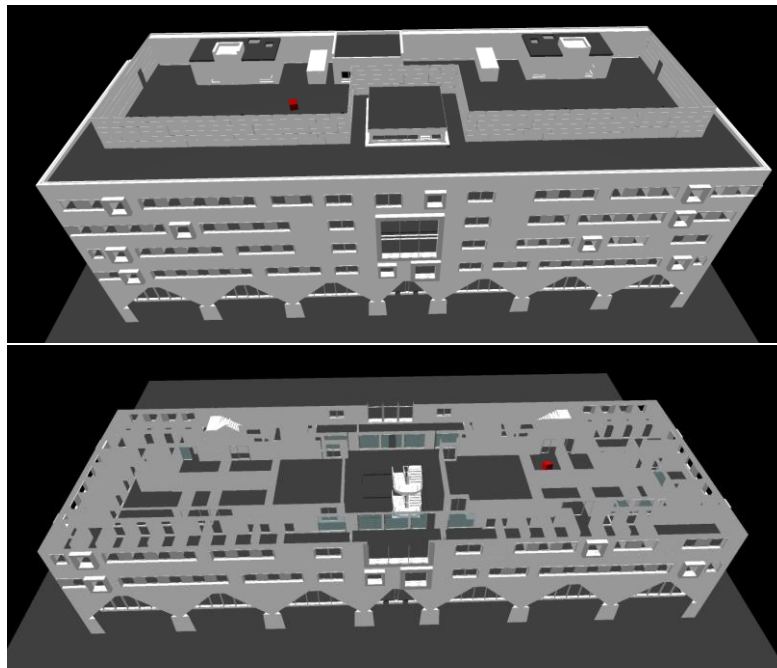


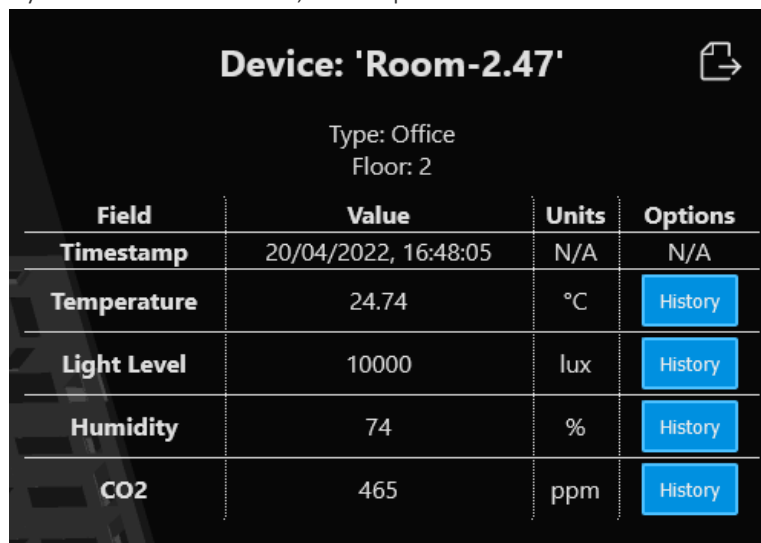
Figure 60 Model before and after selecting the third floor

As you can see upon selecting a floor the model is clearly cut down to that floor for easy access to the devices there. This makes the model much easier to navigate as you do not have to move around inside the model with the camera to find what you are looking for.

8.2.1.5 User must be able to view the data provided by a device ✓(PASS)

Upon selecting a device, the user will be able to view the most recent data entry for that device.

This data should be displayed in tabular format, with options to view historical data for fields.



The screenshot shows a user interface for a specific device. At the top, it says 'Device: 'Room-2.47'' with a share icon to the right. Below this, it indicates 'Type: Office' and 'Floor: 2'. A table displays the most recent data for various fields. Each field has a corresponding 'History' button in the 'Options' column.

Field	Value	Units	Options
Timestamp	20/04/2022, 16:48:05	N/A	N/A
Temperature	24.74	°C	History
Light Level	10000	lux	History
Humidity	74	%	History
CO2	465	ppm	History

Figure 61 Data displayed after selecting a device, includes option to view device history

8.2.1.6 User must be able to visualise historical data for a field on a chart/graph ✓(PARTIAL)

While viewing the most recent data entry for a device, the user should have the option to view historical data of that field, for that device.

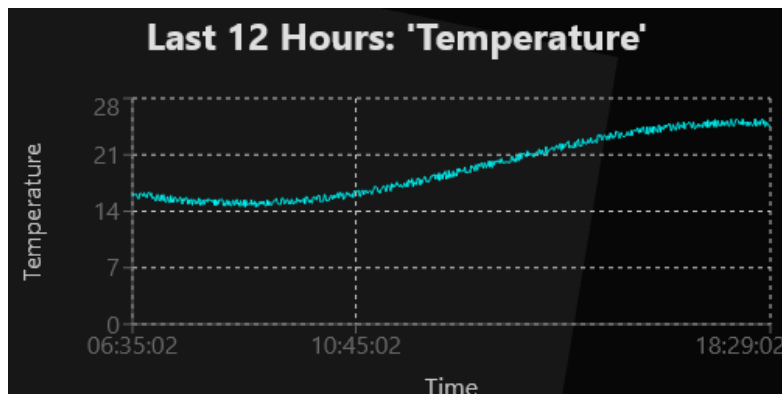


Figure 62 Graph of 'Temperature' in Visualiser

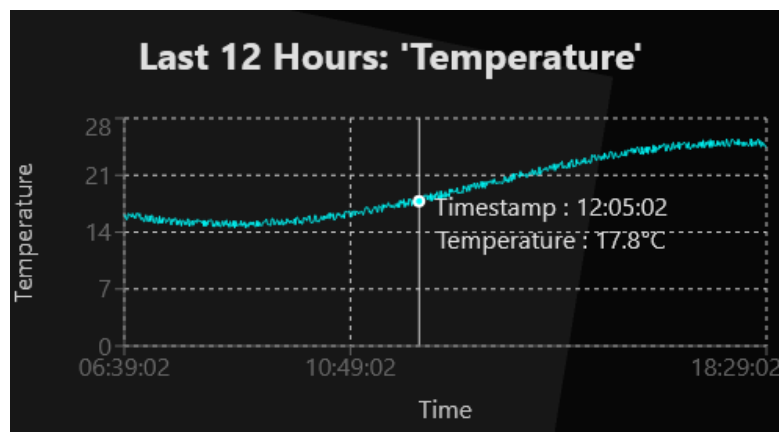


Figure 63 Graph of 'Temperature' in Visualiser when hovering with the mouse

User could also have the option to plot multiple data fields onto many charts ✗ (FAIL).

Only a single chart is currently supported. This is a potential future improvement, but I am not sure how useful it really is to users as they can download the data and compare between devices themselves.

8.2.1.7 User should be able to export data in JSON format ✓ (PARTIAL)

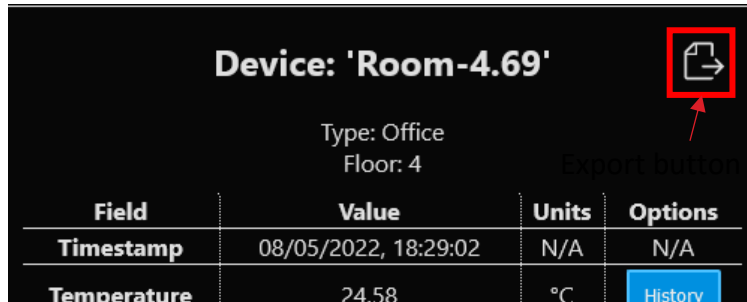


Figure 64 Export button location with device details

With a device selected, the user can select an “Export” option.

They will then choose a location to save a file and be able to download the data in JSON format ✓ (PARTIAL)

The user is unable to choose where to store the downloaded data, but this is not strictly required for this requirement to function making it a ‘nice to have’ requirement rather than a deal-breaker.

8.2.1.8 Users should be able to query the application for information on the entire dataset ✓(PASS)

The application should provide endpoints that allow a user to query the entire dataset at once. These endpoints are provided under the /query path within the API.

Users should be able to filter by floor, device type, device name, and device data fields.

Name	Description
name string (query)	The names of the devices we wish to query. • Can be a single value, or a list of comma-delimited values name
type string (query)	Types of devices we wish to query. • Can be a single value, or a list of comma-delimited values type
floor string (query)	Floors on which the devices we wish to query are • Can be a single value, or a list of comma-delimited values floor
has string (query)	The fields we want the returned data to contain. • Devices and data not containing these fields will not be returned. • Can be a single value, or a list of comma-delimited values has

Figure 65 Documentation for advanced query endpoint parameters

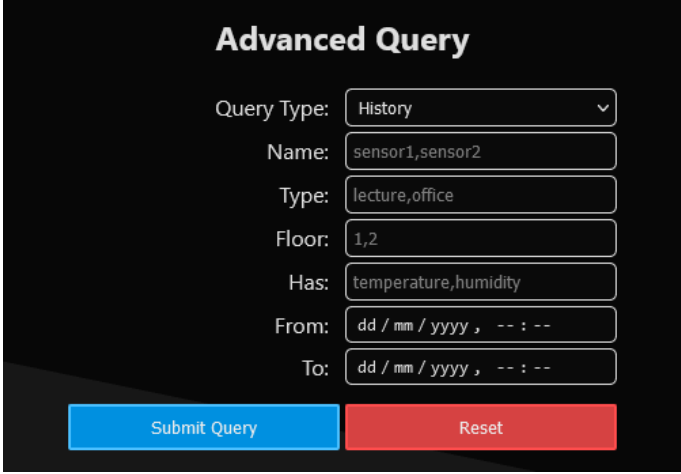
The documentation for the advanced query endpoints can be seen above. These endpoints allow anyone to query the entire dataset. Each of the parameters listed above can be a single value, or a list of values separated by commas. The parameters can be combined to form even more advanced queries. You can make an advanced query on device info, device data, or device history. Under the hood these requests use MongoDB directly, making the query fast and efficient.

For example, querying this URL <https://api.abacws.ggrainger.uk/query/data?name=Room-0.01%2CRoom-2.47&type=office%2Clecture&floor=2&has=temperature> nets the following result:

```
[
  {
    "name": "Room-2.47",
    "type": "office",
    "floor": 2,
    "position": {
      "x": 72,
      "y": 31,
      "z": -154
    },
    "data": {
      "timestamp": 1651948202242,
      "temperature": {
        "value": 20.19,
        "units": "°C"
      }
    }
  }
]
```

Figure 66 Advanced query example result in JSON format

The visualiser service presents a form to allow less technically inclined users to query the entire dataset without interacting with the API. While this form is a step up from forcing users to use the API directly, it could use some tooltips suggesting to the user what each field does without having to consult the API documentation.

The image shows a web form titled "Advanced Query" on a dark background. The form contains several input fields: "Query Type:" with a dropdown menu showing "History"; "Name:" with a text box containing "sensor1,sensor2"; "Type:" with a text box containing "lecture,office"; "Floor:" with a text box containing "1,2"; "Has:" with a text box containing "temperature,humidity"; "From:" with a text box containing "dd / mm / yyyy , -- : --"; and "To:" with a text box containing "dd / mm / yyyy , -- : --". At the bottom of the form are two buttons: a blue "Submit Query" button and a red "Reset" button.

Advanced Query

Query Type: History

Name: sensor1,sensor2

Type: lecture,office

Floor: 1,2

Has: temperature,humidity

From: dd / mm / yyyy , -- : --

To: dd / mm / yyyy , -- : --

Submit Query Reset

Figure 67 Advanced query form as part of visualiser service. Allows anyone to query the entire dataset without technical knowledge needed to use the API directly.

8.3 REPRODUCIBILITY

It has been important throughout this project that I write good quality, maintainable code with high quality documentation. This has been the deciding factor behind the utilisation of technologies such as the OpenAPI specification, TypeScript, and Docker. I have done this so that any future developers or researchers looking at the code for this project do not have difficulty understanding how the application is split up, and how it functions.

The use of TypeScript improves the quality of tooltips and type hints while programming within IDE programs such as Visual Studio Code. This makes the code far easier to read as the IDE can provide live feedback about what variables and fields are available throughout the application. While previously developing applications using JavaScript, the type hints worked half the time, then broke at some points. This made extending an existing application far harder than it should have been. The introduction of TypeScript in this project has massively improved efficiency for me, and hopefully for any future readers of the code.

During development I have used modern web development technologies and techniques. Technologies such as React (Facebook, n.d.) make understanding the application much easier than it would be if only using raw HTML and JavaScript. The application makes use of several open-source libraries for functionality that would otherwise be very complex to build, for example the utilisation of express (expressjs, n.d.) and recharts (recharts, 2022) to build the API and graph rendering respectively.

The use of the service architecture serves to make the application more extensible, as a future developer can add additional services to fulfil tasks if it does not fall under the responsibilities of the existing services. For example, a future developer could add a new service to act as a bridge between the existing API service and a Building Automation System such as BACnet.

Service architecture is a great way to develop extensible programs without requiring developers to understand the system in its entirety or programming language. Adding a new service could be done with a good understanding of the API service which binds the other services together, without an understanding of TypeScript or React.

9 FUTURE WORK

We have come a long way. But there are several potential additions that I think would be beneficial to this application if it were to be deployed and used in a production environment.

This application would benefit from including data provided by existing Building Automation Systems (BAS) by using a protocol such as BACnet. Due to the service architecture used in the final system, the best way to do this would be to create another service responsible for querying the BACnet system and adding the results to the API using existing endpoints. If I had more time at the end of this project, this is the option I would explore first as the project stands to gain the most from the inclusion of more data.

Using this system of services means that integrating this system with any number of BAS would only require that a new service be added to bridge the gap between this system and the BAS in question. This makes the system highly extensible as future developers can simply add new containers to extend the application.

Currently the devices listed in the system can only be changed or added to by altering the `devices.json` file found in the API container. This reduces the customisability of the software. Currently a user that wishes to add more devices to the system could only do so if they had access to the server or environment in which the API service container was deployed. Not all users who wish to add devices, and have legitimate authority to do so, are likely to have access to this. As such it would be beneficial to provide several API endpoints related to the addition and removal of devices.

With the addition of this feature, it would also be necessary to expand the current authentication system. Having one API key shared between all the users and devices is a security risk, the main problem with this is that if the key gets leaked all the data in the system can be destroyed by a malicious user. One way to improve the security of this system would be to have an API key for each person who can add devices, this API key would then be shared with the devices they add to the system. This means that if the key becomes compromised, the amount of data at risk is vastly reduced.

It would also be beneficial to run a user survey (such as an SUS) to ascertain ways the UI of the visualiser service can be further improved. The results from these surveys could they also be used to compare this product from other similar systems such as the one in use by Newcastle Urban Observatory.

10 CONCLUSIONS

Throughout this report, and project, I have successfully designed and developed a software solution to increase the ease of access of IoT data for the Abacws building. The software developed here achieves its objectives. It improves the ease of access and ease of visualisation for IoT data in the Abacws building.

I met 6 out of 8 requirements fully, with 2 others being partially fulfilled. The 2 requirements that were partially fulfilled were of lower priority and do not prevent the application from serving its purpose. Overall, I would consider this application successful.

The application is extensible through its use of service architecture and Docker. Any future developments can be added to the application as additional services deployed alongside the existing ones.

I have made several improvements on the Newcastle Urban Observatory project, which was one of the inspirations for this work:

- The UI of the final prototype is simpler and more in line with modern UI design patterns, such as the side menu, which I have improved using the “hamburger” system.
- The final prototype has made huge improvements to the usability of the application.
- I have introduced an improved interface for mobile devices and atypical sized screens.

In terms of potential future development, we have learned about BACnet and the basics of Building Automation Systems and how they can be used to increase the wealth of data already available via the Internet of Things. Additionally, we have learned about several technologies that are useful for the development of modern applications, e.g., React, Sass, TypeScript, and Docker.

In summary, this application improves on the existing Newcastle Urban Observatory from a UI perspective. The API needs some additional work to integrate with BAS properly but forms a sturdy base for future development. The IoT is an important development in the world of commercial buildings. This project has the potential to improve the smart use of data, to save energy and reduce emissions, help improve usability and provide improved accessibility to the building.

11 REFLECTION

I have enjoyed developing the software product of this project immensely. It gave me the opportunity to learn how to use technologies and techniques I have not used before. The only technologies used in this project that I had prior experience with were React and MongoDB. With the use of TypeScript, express, node, OpenAPI, and Docker I vastly increased the scope of my technical skills over the course of this project.

Dealing with the 3D modelling was also new to me. While I have previously used Autodesk Inventor to model small things, I have never used Navisworks or Blender before. Initially, dealing with the model of the building I was given was painful as it was far too detailed and Navisworks would not export it correctly. However, over the course of the project I have learned a great deal about Blender that I otherwise wouldn't have experienced.

At the beginning of the project, I set out a specification based on the existing Newcastle Urban Observatory application analysed in this report. The application I have developed achieves these specification objectives. It works well on different screen sizes on mobile devices. I have also highlighted some potential future extensions of this application via the use of BAS.

As part of this project, I have incorporated user feedback in the form of a survey. This informed several decisions I made about the design. In future I would expand this across my iterative development process and get more feedback earlier in the design process to highlight potential issues. I was surprised at how fast and how much the quality of my prototype developed with a little user input. This is something I would increase my focus on in future projects.

During this project, I also utilised a Kanban board to keep track of my progress through the functional requirements I set out at the start of development. This helped me keep on track, but I did not use the board enough to realise its full potential. Later in the project, when an idea came to mind, I utilised the GitHub issues feature whenever I encountered a bug or a potential design enhancement. This assisted in my development of the project and is something I will utilise more in the future.

Overall, I have learned a great deal over the course of this project and my skills have improved significantly. I found this project intellectually stimulating and enjoyable, and I hope to work on similar projects in the future.

12 REFERENCES

- Alpine. (n.d.). *about / Alpine Linux*. Retrieved May 8, 2022, from <https://www.alpinelinux.org/about/>
- Docker. (n.d.). *Docker*. Retrieved May 1, 2022, from [docker.com](https://www.docker.com/): <https://www.docker.com/>
- Docker. (n.d.). *Docker Compose*. Retrieved May 1, 2022, from [docs.docker.com](https://docs.docker.com/compose/): <https://docs.docker.com/compose/>
- Domingues, P., Carreira, P., Vieira, R., & Kastner, W. (2016). Building automation systems: Concepts and technology review. *Computer Standards & Interfaces*, 45, 1-12. Retrieved from https://www.researchgate.net/profile/Paulo-Carreira-3/publication/284131458_Building_Automation_Systems_Concepts_and_Technology_Review/links/5a2ff9670f7e9b0d50f8ba14/Building-Automation-Systems-Concepts-and-Technology-Review.pdf
- expressjs. (n.d.). *Express*. Retrieved May 1, 2022, from expressjs.com: <https://expressjs.com/>
- Facebook. (n.d.). *React*. Retrieved May 1, 2022, from reactjs.org: <https://reactjs.org/>
- Facebook. (n.d.). *React Hooks*. Retrieved May 2, 2022, from [reactjs.org](https://reactjs.org/docs/hooks-overview.html): <https://reactjs.org/docs/hooks-overview.html>
- Huynh, D., & Nguyen-Ky, S. (2020). Engaging Building Automation Data Visualisation Using Building Modelling and Progressive Web Application. *Open Engineering*, 10(1), 434-442.
- Khronos Group. (n.d.). *Khronos glTF Overview*. Retrieved May 04, 2022, from Khronos: <https://www.khronos.org/glTF/>
- Makabee, H. (2012, 02 05). *Separation of Concerns*. Retrieved May 03, 2022, from Effective Software Design: <https://effectivesoftwaredesign.com/2012/02/05/separation-of-concerns/>
- Microsoft. (n.d.). *TypeScript*. Retrieved May 1, 2022, from [typescriptlang.org](https://www.typescriptlang.org/): <https://www.typescriptlang.org/>
- MongoDB. (n.d.). *MongoDB*. Retrieved May 1, 2022, from [mongodb.com](https://www.mongodb.com/): <https://www.mongodb.com/>
- mrdoob. (n.d.). *Three.js*. Retrieved May 1, 2022, from threejs.org: <https://threejs.org/>
- MSEdgeTeam, Mikehoffms, vchapel, captainbrosset, jum-trd-ms, Reezaali, . . . realenginerd. (2022). *Overview of Progressive Web Apps (PWAs)*. Retrieved May 13, 2022, from <https://docs.microsoft.com/en-us/microsoft-edge/progressive-web-apps-chromium/>
- MURATAGAWA, K. (2021). *Blender Stack Exchange*. Retrieved March 27, 2022, from <https://blender.stackexchange.com/a/229803>
- Newman, M. (2022). *BACnet: Answers to Frequently Asked Questions*. Retrieved May 6, 2022, from <http://www.bacnet.org/>
- nodejs. (n.d.). *NodeJS*. Retrieved May 1, 2022, from [nodejs.org](https://nodejs.org/en/): <https://nodejs.org/en/>
- recharts. (2022). *Recharts*. Retrieved May 9, 2022, from <https://recharts.org/en-US/>
- Sass. (n.d.). *Sass*. Retrieved May 2, 2022, from sass-lang.com: <https://sass-lang.com/>

scottie1984. (2022). *swagger-ui-express* - *npm*. Retrieved May 05, 2022, from
<https://www.npmjs.com/package/swagger-ui-express>

swagger. (n.d.). *Openapi 3.0 Specification*. Retrieved May 1, 2022, from [swagger.io](https://swagger.io/specification/):
<https://swagger.io/specification/>

Traefik. (n.d.). *Traefik Proxy*. Retrieved May 11, 2022, from <https://traefik.io/traefik/>