Sheffield Hallam University

College of Business, Technology and Engineering

DEPARTMENT OF COMPUTING

55-604708 Project (Technical Computing)

2019/20

Author: Manolis Vrondakis

Student ID: 26028420

Year Submitted: 2020

Supervisor: Dr. Carlos Eduardo da Silva

Second Marker: Alessandro Di Nuovo

Degree Course: BEng Software Engineering

Title of Project: Controlling your IoT project from anywhere effortlessly with VIoT

Confidentiality Required?: No

Acknowledgements

I would like to thank my project supervisor Dr. Carlos Eduardo da Silva for his valuable guidance and advice throughout this project.

I would also like to thank my friends and family who have helped support me throughout university.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem and Motivation

If you want to build your own remotely accessible internet of things (IOT) device, you need the infrastructure to go along with it. This is known as an IoT platform. Existing Internet of Things platforms provide both supporting infrastructure, such as servers and the support of multiple device protocols, and also the user interface side of IoT such as displaying data, for example sensor readings. However, these platforms are limited as they are not friendly to the final user in allowing control, and mainly focus on collecting and displaying data from the device. Furthermore, most platforms are geared towards industry and lots of low-level configuration is required for their setup. If you wanted to control your custom Internet-enabled lamp using an existing platform, you would have to navigate a barrage of menus and write unnecessary boilerplate code - which is far too intricate and has too steep of a learning curve to quickly do in the home.

Creating your own easy to use control panel specific to your device requires a significant amount of time and effort which involves hosting your own server to handle communication, creating a custom user interface using whatever platform technology is appropriate (if you want to control the device with a mobile app and browser then two separate interfaces are required), handling authentication and much more. This pulls the focus away from the device and puts it onto the infrastructure - which is something that a device maker does not want to worry about. The aim of this project is to abstract this complexity away from the maker and develop a solution where a device can connect to the service, describe itself and have a quick to access and easy to use control panel generated for it which allows the device to be controlled

remotely from anywhere.

1.2 Objectives

The main objective of VIoT is to make the experience as easy as possible for an IoT hobbyist creating an IoT project, which is controllable from an online interface. VIoT should allow new users to easily integrate the service into their IoT projects by providing client libraries and handling infrastructure, authentication, user-interface generation and more for them. An internet of things platform will be created which incorporates user-interface generation based on templating, along with libraries that allow users to connect their device to the service. To evaluate the solution, devices which utilise the platform to be controlled remotely will be created, incorporating rich and dynamic control panels.

1.3 Document Structure

This document is organised as follows:

Chapter 2 presents background knowledge about Internet of Things, web development technologies, different Internet of Things protocols and platforms, software engineering methodologies and user interface generation based on templating. This chapter also considers which particular approaches and tools will be used in the creation of VIoT.

Chapter 3 presents the design of the solution and broadly describes the architecture of the service.

Chapter 4 describes how the project was implemented, including which languages and frameworks were utilised, along with the software engineering approaches that were used.

Cheaper 5 demonstrates the devices that were created to evaluate the system in full and also evaluates the implementation of the project.

Finally, chapter 6 discusses the contributions and limitations of this project as well as future work that could contribute to the improvement of the project.

Chapter 2

Background Research

2.1 Web development technologies

2.1.1 Single-page applications & Multi-page applications

Traditionally, every time data is exchanged back and forth between a user and website, a new page is requested from the server. This approach can be useful due to the high level of compatibility between browsers, as all the client must do is display the server-generated HTML. Also, it typically requires a smaller technology stack than single page applications, making development easier. However, this approach can heavily affect user experience as it requires the entire page to be generated on the server, sent to the client, then rendered in the browser which can take a significant amount of time, leaving the user waiting. A study by Google DoubleClick reported that 53% of mobile site visits are abandoned if the page takes more than 3 seconds to load, suggesting that load times are very important in retaining visitors on a site. Furthermore, it is likely that if an additional client such as a mobile phone application needed to interact with service, a new backend would have to be created.

With the advent of AJAX and other modern JavaScript features, it has become possible to render pages on the client completely separately from the service, and only request data from the server. This enables a fluid user experience. Web applications built using this approach are known as single page web applications because they only make an initial request for the page when it is loaded. These feel a lot faster than traditional web applications to the user, and with advent of advanced single page application libraries it has become trivial to quickly develop a web application following this principle.

However, single page applications are not without their disadvantages; often they are not supported by search engines and therefore are harder to perform search-engine optimization on. Also they rely on JavaScript, meaning that if the user has JavaScript disabled in their browser, the application will not work.

React

React is an open-source JavaScript library created and maintained by Facebook and other contributors. It is aimed at building complex and interactive user interfaces on the web. It is the most popular JavaScript framework of its type (single page application framework) followed closely by Angular and VueJS in 2019.

React implements a component based architecture, which embraces the inherent coupling of UI and rendering logic. Instead of splitting markup and logic into separate files like other single page application frameworks, React separates concerns with loosely coupled units, which it refers to as "components" that include both markup and logic. A component in React is a function or ES6 class which accepts properties (known as props) and returns JSX which describes how UI should appear. There are two types of React components: functional and class-based. As shown in *Figure 2.1*, functional components are the simplest type, which are declared with a function that returns JSX.

```
const Button = ({ label, onClick, disabled }) => {
  return <a onClick={!disabled && onClick}>{label}</a>
}
```

Figure 2.1: Example of React functional component

Class-based components are created by declaring an ES6 class with lifecycle methods included. Another name for class based components is "stateful" components, as they can have their own state which can be passed to children. An example of a class-based component is shown in Figure 2.2.

A feature that React benefits from is utilisation of a "Virtual Document Object Model (DOM)", which is an in-memory representation of the browsers DOM. When a component is updated, either by its state or props changing, instead of updating the entire DOM, React computes exactly what has changed by comparing its virtual DOM to the browsers, then efficiently up-

```
class CountingButton extends Component {
    constructor(props){
        super(props);
        this.state = {
          pressedTimes : 0
        }

        this.onClick = this.onClick.bind(this);
    }

    onClick(){
        const {pressedTimes} = this.state;
        this.setState({
            pressedTimes: pressedTimes + 1
        });
    }

    render(){
        const {pressedTimes} = this.state
        const buttonLabel = `The button was pressed ${pressedTimes} times`;

        return (
            <Button label={buttonLabel} onClick={this.onClick} />
        )
    }
}
```

Figure 2.2: Example of React class-based component

dating only the necessary parts which the browser needs to re-render, leading to significant speed increases. This is also mostly done under the hood with little input from the developer. React also utilises life-cycle methods which hook onto events of a component, allowing users to alter behaviour. Examples of these include componentDidMount which is called when a component has been created in the user interface and associated with a node in the DOM. Another example is render() which is the only required life-cycle method and is called every time a component is updated, which is when its state or props change and is used to render actual DOM elements. This is shown in Figure 2.3.

In earlier versions of React, only updating the DOM was handled by React itself, meaning that without additional libraries any data would have to be passed top-down (parent to child) via props. This can add unnecessary complexity to applications that require certain types of props to be passed to a large amount of components, for example any theming data. The addition of Context in React provides a way to share "global" values like these without having to explicitly pass props down every level of the tree.

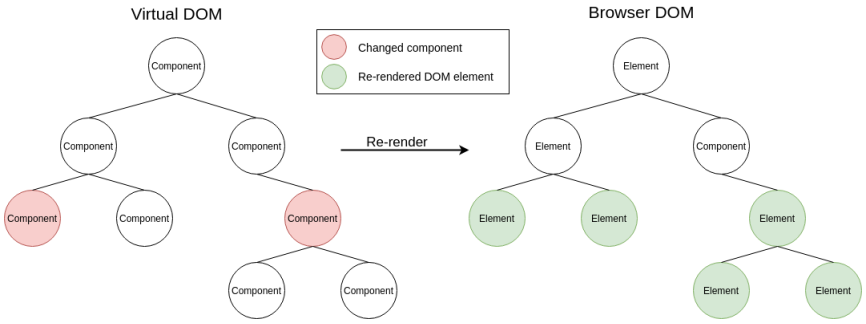

Figure 2.3: React virtual DOM updating the browser DOM

2.1.2 Angular

Angular is another open-source single-page-application web framework created by Google. It is a complete rewrite of the popular AngularJS framework, created by the same team. Unlike React, Angular was created to be used with TypeScript, which is a language created for the development of large applications which transpiles to JavaScript.

Angular prides itself on extensive documentation and modular architecture, which is specifically designed to be as scalable as possible - making it ideal for large teams. This is evidenced by its use at large companies such as Microsoft and Apple. One downside to Angular is that it has a lot of concepts, meaning its learning curve is steep in comparison with other frameworks such as React.

Each Angular application is a set of modules, referred to as NgModules. An application always has one root module that enables bootstrapping between other modules, and usually has more feature modules depending on the size of the project. Angular has the concept of components which are different from React components. In Angular, they are a set of templatable views which Angular can choose and modify according to program logic. Views can have different services injected into them, which provide business-logic functionality unrelated to specific views, making applications highly modular and efficient. Both components and services are simply classes, with decorators that mark their type and provide metadata that tells Angular how to use them (Angular, 2020). Components include a template in their metadata, which is a HTML file with Angular markup inside, allowing specific parts to be modified before use.

Bulma

Bulma is an open-source, free CSS framework based on Flexbox, a recent innovation in web browser technology allowing elements to be arranged dynamically, depending on the screensize. While a huge number of CSS frameworks exist, Bulma was chosen for this project. While a relatively new framework, Bulma is has a modern look, is modular, doesn't utilise any Javascript libraries for its base functionality, is responsive, well documented and compatible with all major browsers. It is also written using Sass which is a popular & powerful CSS pre-processor which allows developers to easily modify variables and have the changes compiled into the CSS code. For example, the default colours and sizes can be easily changed.

2.1.3 NodeJS

Browsers have been competing to offer the user the fastest experience on the web since their inception. Companies that create browsers (Google - Chrome, Mozilla - Firefox, Microsoft - Edge) invest a lot of time and money into finding ways to make JavaScriot run faster, and as a result of this Google created the V8 JavaScript engine, which was a huge milestone in increasing the performance of JavaScript on the web. V8 parses JavaScript code at runtime into an abstract syntax tree, and then generates byte-code from this using the V8 interpreter "Ignition". This bytecode is converted to optimized machine code using the V8 optimizing compiler "Turbofan" on the fly, leading to huge performance increases.

Because of these speed increases, it has become viable to incorporate V8 into server side technologies. NodeJS was built on the V8 JavaScript engine and has since been adopted by major companies, as it offers a modern development experience with a huge ecosystem of free user-made libraries and frameworks available for use by developers. As well as this, it keeps web technologies more tightly coupled, as both the server and client code can be written in JavaScript.

The design of NodeJS, offering an event-driven architecture capable of asynchronous I/O, allow it to offer a high level of throughput and scalability, which is ideal for web applications that have many I/O operations, as well as real-time web applications.

2.2 Software engineering methodologies & processes

There is an ever-growing list of methodologies aimed at optimizing the process of creating software by providing a set of guidelines for teams and individuals to follow. The two most common discussed in this section are Waterfall and Agile

Waterfall

The waterfall model is a much older methodology for developing software. It empathises splitting the project into distinct phases which move forward step by step until the project is completed. You make a project plan upfront, and then execute it linearly without any deviations. Each waterfall project follows the same phases:

- **Requirements** - Where needs of the project are analysed and what the software needs to do
- **Design** - Where the system architecture is planned and the technology is chosen
- **Programming** - Where the code is written
- **Testing** - Where the code is tested and ensured it does what is supposed to
- **Operations** - Where the code and final product is deployed to a production environment and support is provided.

The different phases of a waterfall project must happen in the exact order defined and one set of tasks cannot happen before the previous ones have been completed. This presents problems when the time is estimated incorrectly, as if one stage goes over its allocated time, the rest of the phases are delayed.

However, there are some benefits of Waterfall. One advantage is the extensive documentation required by the methodology. Because you can't go back to a previous phase after starting a new one, you are forced to create extensive documentation. Another advantage is that it is the easiest methodology to comprehend - as the projects are split into easy to understand and distinct phases. Another benefit of waterfall is that if a project is being created for a client, there is no need for their input after the initial requirements phase, which can maximise time efficiency.

The waterfall methodology comes from the construction & manufacturing industry - where iterations to projects are very costly and therefore too expensive to change. The problem with developing a software product is that planning for exact timeframes is very tricky as you can never be sure of exactly how much time certain tasks will take.

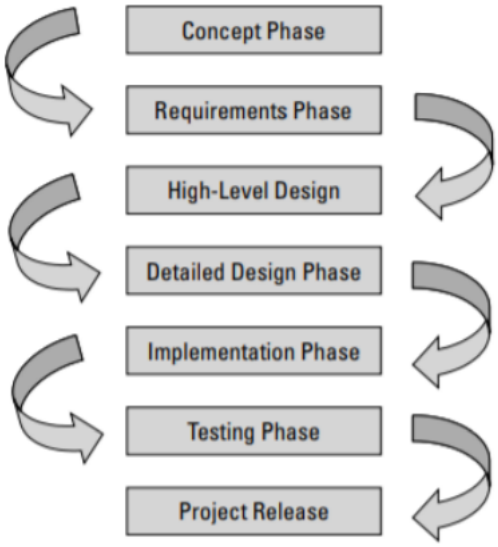

Figure 2.4: Phases of Waterfall Methodology (Luckey & Phillips, 2006)

Agile

Agile is a modern methodology and process for developing projects. While it can be used for any type of project, it was created to aid in the development of software.

The main principal of Agile is to break large projects into small and manageable chunks of work referred to as "sprints", which usually take place in a fixed time period. After each task is completed, something of value is produced which is presentable to the relevant parties.

To begin a sprint, developers will hold a sprint planning meeting, which is where developers assign "tasks" to the sprint. The time of each task is estimated by the team which gives a time budget, and tasks can be added and removed in order to stick to the initial time budget of the sprint.

After a sprint is finished, a retrospective takes place. During a retrospective, developers will assess the successes and failures of the sprint in order to

improve the next sprint.

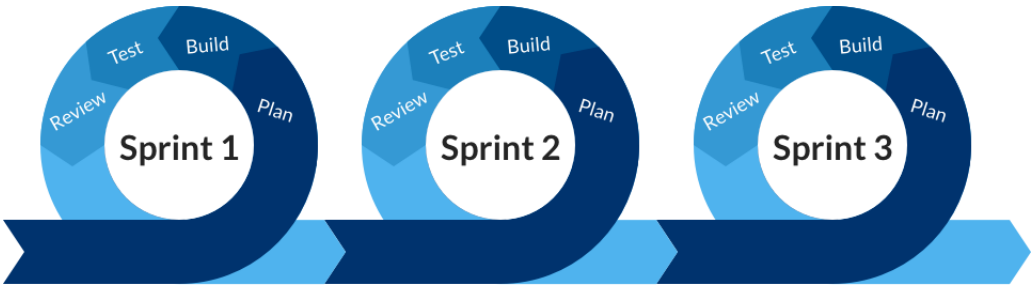

Figure 2.5: Agile Methodology (Littlefield, 2019)

Unlike the Waterfall project methodology which has strict sequencing of events in place, Agile has all stages of waterfall such as researching, design, development and testing happening at once. Agile is an effective methodology because it allows flexibility in project management.

Furthermore, in a report published by the Standish Group, 2014, it was concluded that projects created using the Agile software development methodology are up to 3 times more likely to succeed when compared to projects using a more traditional software development approach. A comparison between Waterfall and Agile project successes is shown in Figure 2.6

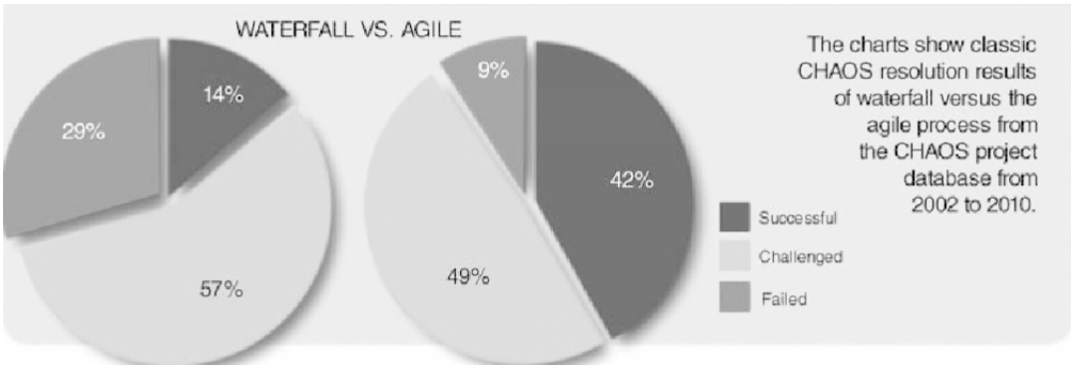

Figure 2.6: Comparison between Waterfall and Agile project successes (Schwaber & Sutherland, 2012)

2.3 User interface based on templating

The idea of generating user-interface from a template has been explored before. There are a number of projects are built around this idea, including Alpaca forms - which is a project aimed at rendering HTML forms with a

JSON template input. An example of a form rendered using Alpaca forms is shown in Figure 2.7, with the JSON used to generate this form shown in Figure 2.8.

There are already some libraries that focus on HTML UI generation from JSON, Examples of these include: Alpaca Forms and uniforms. The reason a custom schema was chosen is because the idea of dynamic template generation with state considerations & IoT functionality has not been implemented in any other platforms. Since the platform technology with WebSockets is not a standard, using another templating engine would almost certainly require heavy modification to the core code. Furthermore, creating a templating engine from scratch allowed us to have maximum stylistic control, to make the elements uniform with the rest of the client.

A major advantage of taking this templating approach is that it is platform agnostic. Meaning that if another device needed to be supported, for example a phone app, all that would need to change is the code that interprets the template, rather than updating every interface. Different clients can interpret the template in different ways, meaning that one template can support multiple devices. Another advantage is the ease of creation for the user, the JSON template files have a relatively low learning curve and are easy to understand and pickup for first time users. Whereas using a domain-specific language (DSL) or HTML & CSS would have a steep learning curve as the user would have to learn something new and significantly more complicated than a JSON template.

2.4 Internet of Things

The term Internet of Things was coined by Kevin Ashton in 1999. Ashton was a 30-year-old computer scientist working on supply chain optimization using radio frequency identification tags (RFID) at P&G. He was trying to persuade P&G to put RFID tags and other sensors on products in their supply chain in order to generate data about where the products were, whether they'd been scanned in a warehouse, placed on a shelf, or sold.

At the time, the internet was a new exciting technology and Ashton needed something to get the attention of the executives at the company. In his presentation about the technology, he was referring to the supply chain being a "Network of Things", and the internet being a "Network of Bits" - so he decided to merge the two together, coming up with "Internet of Things". While Ashton got the attention of some P&G executives, the term "Internet of Things" did not get widespread traction until summer of 2010,

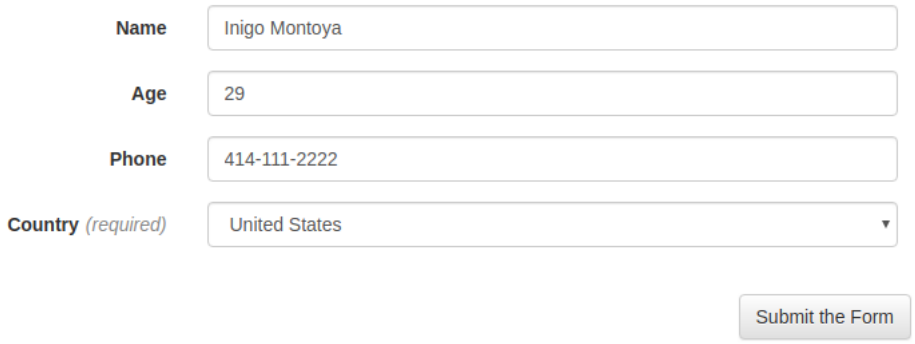

Figure 2.7: Example of a form rendered using Alpaca forms

```
{
  "name": {
    "type": "string"
  },
  "age": {
    "type": "number",
    "minimum": 0,
    "maximum": 50
  },
  "phone": {
    "type": "string"
  },
  "country": {
    "type": "string",
    "required": true
  }
}
```

Figure 2.8: JSON used to render the form in Figure 2.7

when information leaked that Google's StreetView service had not only took 360-degree pictures, but had also stored information about Wi-Fi networks. People questioned whether this was the start of Google not only indexing the internet, but also the physical world (Knud Lasse Lueth, 2020).

20 years after the initial usage of the term, "Internet of Things" is now an ambiguous term which refers to controlling and viewing data from internet enabled smart devices.

While the idea of "Internet of Things" was initially only being used by commercial entities, the technology has evolved and become cheap enough to be used by the everyday consumer. With the advent of low-cost Wi-Fi enabled microcontrollers it has become possible for there to be thousands of cheap commercial IoT devices in the market. Examples of these include smart light bulbs, smart plugs and smart thermostats.

2.5 Internet of Things protocols

2.5.1 MQTT

Message Queuing Telemetry Transport (MQTT) is one of the most commonly used protocols for IoT devices. Its small code footprint, low power usage, minimized data packet size and ease of implementation make it suitable for Internet of Things devices, which are often constrained devices with low-bandwidth and power usage limitations.

As shown in Figure 2.9, MQTT is a publish/subscribe protocol that allows devices to connect to a broker, which mediates communication between its connected devices. Each device can subscribe to a topic, and other devices can publish messages to this topic. Once a message has been published, the broker looks at which clients have subscribed to the topic and forwards it to them.

MQTT has scalability at its core, as it incorporates a central server which monitors clients, subscriptions and messages. If another device is added to the network, code doesn't have to be changed on subsequent devices to make them communicate, as this is facilitated by the broker. Devices only subscribe to the data they are interested in, allowing devices to communicate as efficiently as possible.

In the case of network downtime or an unreliable connection, MQTT implements a quality of service (QOS) feature which reliably ensures that mes-

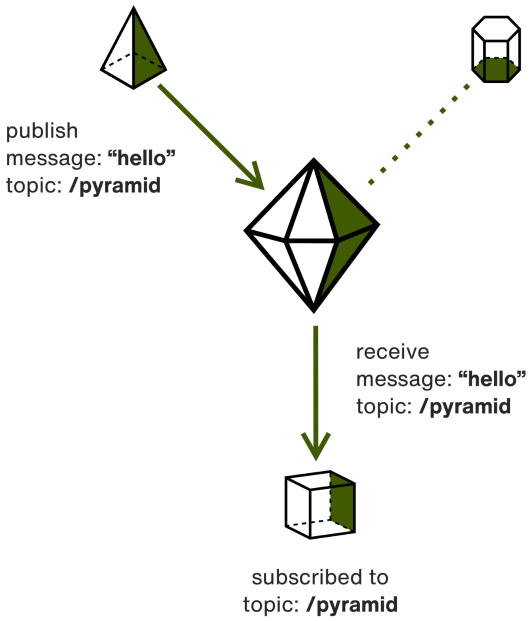

Figure 2.9: MQTT Publish/Subscribe mechanism (Pulver, 2019)

sages are delivered to clients, by repeatedly sending messages until they are received. This feature is also configurable for devices which do not require it. Since network connections are often volatile, MQTT has an option called last will/testament which makes it possible to publish a message when a device goes offline, which is very useful for monitoring devices. This is done by using a keep-alive connection, requiring the device to send a message "ping" to the server every few hundred milliseconds, as shown in Figure 2.10. If the server stops receiving these commands from one of its clients, it can safely assume it has gone offline and take actions accordingly.

2.5.2 Zigbee

Zigbee was created to work effectively with devices with low-latency and low-power requirements. Similarly to Bluetooth and Wi-Fi, it offers short-range connectivity of up to 100 meters, making it ideal for IoT devices. However unlike Bluetooth and Wi-Fi which are high data-rate standards supporting the transfer of media, software and other types of large files, ZigBee is designed to transfer at a maximum of just 250Kbps to use the minimum amount of power on the device, increasing the battery life of battery powered devices.

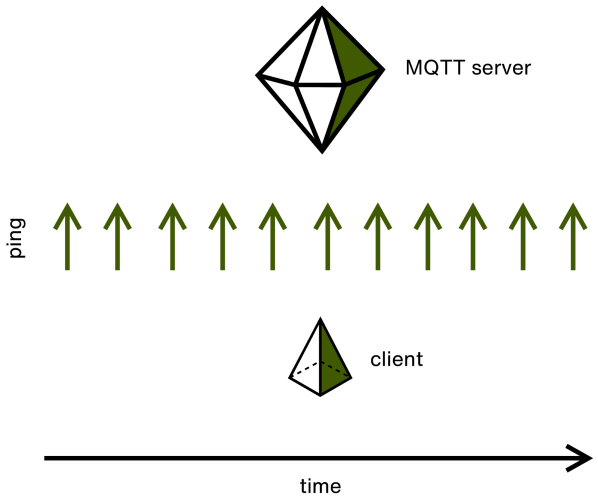

Figure 2.10: How MQTT keep-alive works (Pulver, 2019)

It was initially built in 1998 as a low-power alternative to Bluetooth due to its inefficiency to work in certain applications, making it a great choice for IoT device including industrial automation, embedded systems and home automation.

“Zigbee is one of the most widely adopted smart home technologies in the IoT. With over half a billion Zigbee chips worldwide it has made its way into many of the largest smart home ecosystems.” (Zigbee Alliance, 2020)

Zigbee is based on the Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 personal network standard consisting of 2 layers: physical and MAC, and implements its own application layer and network layer. Zigbee devices create a ”mesh” network, where each device communicates with as many other devices as possible, and route data from device to device. Zigbee supports over 65,000 devices at any one time in an interconnected network, making it ideal for IoT devices.

2.6 Internet of Things platforms

In order to have a network of connected devices, something is needed to facilitate the connection between these devices. This is known as an Internet of Things platform. The most advanced Internet of Things platforms allows scalability at any endpoint, offering the maximum performance of device connectivity regardless of changing circumstances. Internet of Things

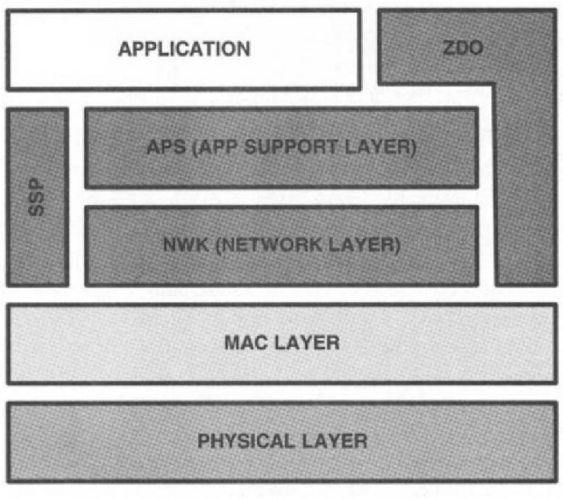

Figure 2.11: Zigbee Architecture (Eady, 2007)

platforms also have varying levels of user-friendliness. Since the Internet of Things platforms is almost as fragmented as it is boundless, IoT platforms have evolved many different types of solutions to address the heterogeneity of specific IoT-powered devices.

At their core, Internet of Things platforms facilitate the data flow, provide security & authentication for devices, and collect, visualize and analyse data sent from devices. Using an IoT platform allows greater focus to be put on the value-adding parts the device, instead of requiring device makers to focus on lower level technologies. This means that maximum attention can be paid to the actual development of the device, providing huge benefits such as being able to send the device to market faster - or in the hobbyist case, having a working device faster.

"Connectivity and data management, which historically have required huge investments in time and development costs should be "givens" on the IoT platform, as reliable as electricity generation, and just as liberating to users." (Perry, 2016)

Since Internet of Things is a new & emerging technology and therefore is drastically changing every few years, it is important that the platform and system protocols it incorporates can be easily understood and used by new users. Platforms must be intuitive and employ the inherit design patterns and features that a user would expect from an Internet of Things platform, to make them easy to use and pickup for first time users.

Internet of Things platform features

One key feature that is incorporated by a large majority of Internet of Things platforms is device management. When a device is first developed and installed, it requires constant checks and management to keep downtime to a minimum. Sometimes devices will fail and need to be repaired or require updates to add functionality & bugfixes. Because of this, most Internet of Things platforms implement device management capabilities. As stated in Weber, 2016, four key fundamentals of device management exist:

- Provisioning & authentication
- Configuration & control
- Monitoring & diagnostics
- Updates & maintenance

Provisioning refers to adding a device into the platform, including authenticating them and making sure that only legitimate devices can be added.

Usually when an internet of things device is shipped its configuration is not standard, as the end user may want to update settings such as its name, location, and other application specific settings.

Especially considering systems which have hundreds or thousands of interconnected devices, it is important for Internet of Things platforms to incorporate tools to monitor these devices so if there are failures, they can easily be identified and rectified.

Bugs are an inevitable by-product of creating software. Therefore, most IoT platforms provide a mechanism for updating devices post-deployment. This is known as over the air updates.

Figure 2.12 shows how device management works in the popular IoT platform AWS IoT core. Including provisioning, configuration, updating, and control.

2.6.1 Amazon AWS IoT Core

AWS IoT Core is a managed cloud service for IoT devices. It provides support for devices to connect using MQTT and HTTP. A distinguishable feature of this platform is the fact that it can integrate with other AWS

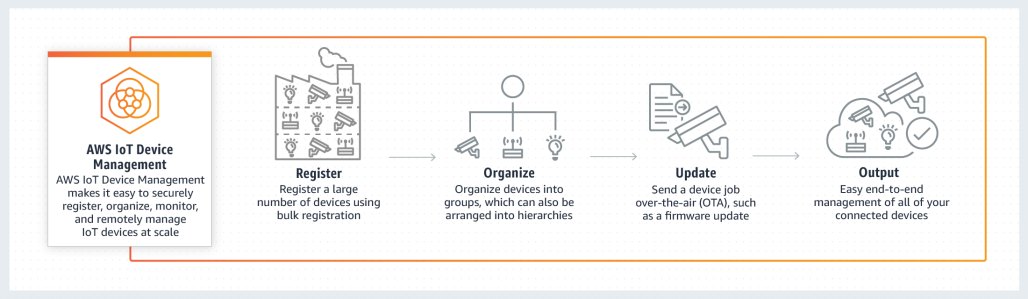

Figure 2.12: Amazon Web Services, 2020b Life-cycle of device management in Amazon Web Services IoT core, including provisioning, monitoring, updating and controlling

cloud services such as Amazon Lambda and Amazon Kinesis. It provides secure data ingestion as well as compute capabilities utilising cloud services Amazon Lambda and Amazon Kinesis, as well as storage. It supports encryption, authentication, monitoring, as well as other services which makes it ideal for a secure, reliable and scalable IoT infrastructure solution for enterprise applications.

AWS IoT Core is built on AWS Cloud which is used in over 190 countries (Amazon Web Services, 2020a). The platform employs a drag and drop interface to connect web services and devices, make developing IoT applications faster.

2.6.2 Thingsboard

ThingsBoard is an open-source IoT platform that is focussed on allowing users to control and monitor IoT devices.

One benefit of ThingsBoard for some users is its open-sourced nature, allowing it to be hosted by anyone. It is also available hosted in a Software as a service (SaaS) model, meaning that it isn't necessary for users to deploy the platform themselves. ThingsBoard provides device management, data collection, processing and visualisation for IoT projects. It supports industry standard protocols MQTT, CoAP and HTTP.

The standout feature of ThingsBoard is its IoT dashboards feature. It allows the user to create rich dashboards for data visualization and remote device control in real time. An example of this is visible in Figure 2.13.

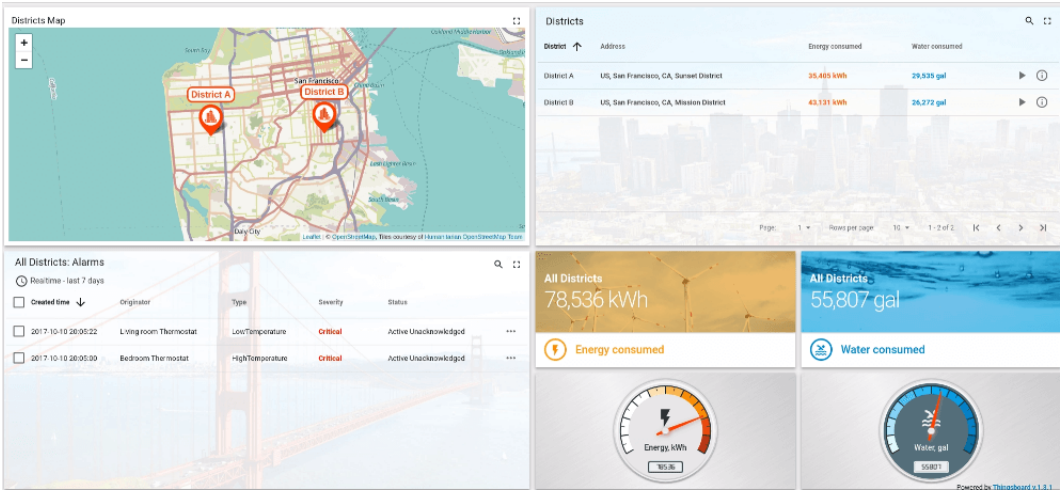

Figure 2.13: Exmaple of an IoT dashboard created using ThingsBoard (ThingsBoard, 2020)

2.7 Final Considerations

The client for VIoT will be written for browsers. The most obvious way to allow users to have access to their own device-specific control panels would be to allow them to upload HTML, JavaScript and CSS. Doing this would have some major disadvantages. Firstly, these files wouldn't be trivial to create, as it would require knowledge of web technologies and wouldn't take the complexity away from the user - a key aim of VIoT. Secondly, there are some major security concerns in allowing other people's code to run in the same domain context of the VIoT client. Lastly, the code would then become platform specific and only intrinsically support clients which can run JavaScript and render HTML and CSS.

In order to abstract the complexity of making a user interface away from the user and to counter the concerns above, it was decided that the user interface would be generated using a JSON template. JSON was selected as the templating language as it's supported in basically every programming language and technology, has a small code footprint and can easily be read by computers and written by humans.

For the templating part of the site, I decided to implement my own system which would offer maximum control and flexibility, this decision was taken because no existing templating engines would support the IoT functionality required by the site, meaning they would have to be heavily modified.

We decided to use the Agile methodology for developing the VIoT platform.

This is because there's a high likelihood that the requirements will change during the development of the project and Agile allows this to happen easily without restarting entire processes, allowing new features and functionality to be added. Since the project will only be demonstrated once it is complete, the benefit of having a deliverable at the end of each waterfall stage will not be beneficial.

Lastly, VIoT will be a hosted site, taking the complexity of hosting a project of this size away from the end user.

Chapter 3

Design

3.1 Introduction

The VIoT platform is a solution for IoT control. It empowers makers to control their Internet of Things devices without investing significant amounts of time in creating the underlying infrastructure which is traditionally required for remote device control. It does this without sacrificing on customisability enabling the maker to focus on the device itself. VIoT is composed of a device management platform and a template based user-interface engine. These two elements combined together create a powerful and user-centred experience for controlling IoT devices.

The architecture of the device management platform is shown in Figure 3.1. It consists of a client component which communicates with a service component through HTTP and WebSockets. The service communicates with a MySQL database and a central MQTT broker. The central MQTT broker has a "Forwarder service" listening to all messages published and forwarding them to a device MQTT broker. Devices on the network subscribe and publish to the relevant topics on the device MQTT broker, to communicate back-and-forth with the service. The architecture is designed this way to allow for multiple protocols to be implemented if required. It is possible for other services to connect to the central MQTT broker and provide an interface for devices to utilise the platform using other protocols.

The template based GUI engine takes a user-created or generated JSON template and device state as an input and outputs a dynamic, real-time graphical user interface which is reactive to the state of the device.

The remainder of this chapter describes the VIoT platform (Section 3.2) and

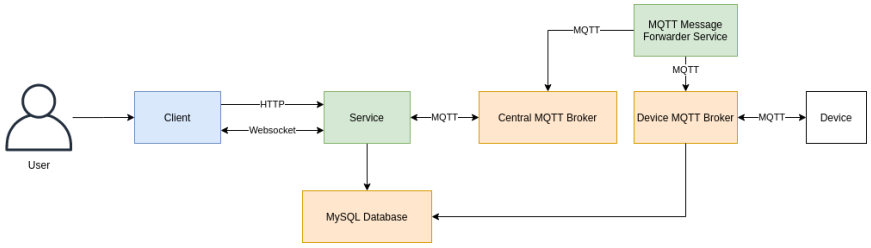

Figure 3.1: Simplified Service architecture

the template engine (Section 3.3) in detail.

3.2 The VIoT Platform

VIoT provides organisation to a users devices, which is useful in cases where they have many. The platform allows users to create "environments" such as a building or outdoor space. Each environment can have a number of zones associated with it, for example "Kitchen" or "Bedroom". These are referred to as "Spaces". Spaces have a name and image to help distinguish them from one another. All of a users environments and spaces are displayed on the "Spaces" page for easy access.

The platform allows users to add, remove and edit devices from the Devices page. To create a device, the user inputs its name, selects an image and optionally assigns a template; the mechanics of which is described in Section 3.3.3. After this is complete the device is added to the system and appears on the Devices page. A unique API key is generated for each device, which the user can use to connect their physical device to the platform. Each device can be assigned to a single space, making it appear when the space is accessed. If the device is online, the user can access the templated control panel as described in Section 3.3.

Devices can initialize a connection with VIoT by connecting to the device MQTT broker, with the device's API key as their username and password. This API key is checked by the broker and the connection is accepted if it is valid. As soon as a device connects, it subscribes to a "receive" topic and publishes a connection message containing the "connect" event. All topic names are prefixed with the device's API key, along with a "/" as per MQTT convention.

3.2.1 Device MQTT Message Structure

Each MQTT message sent by the service or connected devices to communicate follow the same structure. Firstly, the topic name begins with "device/" to signify the recipient, followed by the unique API key of the device, followed by the command. There are 4 commands reserved by the service which are described in Tables 3.1, 3.2, 3.3 and 3.4.

Users can use any other command name in their template using the "action" property on interactive elements as shown in Table 3.5. The value property is taken from the element itself, for example a dropdown would have its selected value in this property, and a toggle button would have ""true" or ""false" depending on if it's toggled or not. The data property is an optional value that comes directly from the template, which is meant to be a place to store additional contextual information about the element required by the device, such as which tab the element is on.

Command	connect
Description	Sent to the platform by devices when they first connect. Used to notify the user when a device has connected
Example	<No content >

Table 3.1: Description of "`connect`" command.

Command	status
Description	**RPC** - Sent to a device when a user requests the device's online status on the platform. Device responds with its current status
Example	{status: "<online/offline >"}

Table 3.2: Description of "`status`" command.

Command	template
Description	**RPC** - Sent to a device when a user requests the device's template. Device responds with its current full state and template if is stored on the device
Example	{template: "{...template}", state: {TEXT: "On", LABEL: "Hello"}}

Table 3.3: Description of "`template`" command.

Command	state
Description	Sent by a device when its state has updated. This can be the full state or partial state (only including some properties) which is merged by the client
Example	state: TEXT: “Off”

Table 3.4: Description of “`state`” command.

Command	<any >
Description	Sent by the platform when a user performs an action on the device control panel.
Example	{value: 23, data: {tab: “Bedroom”}}

Table 3.5: Description of device actions

3.3 VIoT Template Engine

VIoT implements a templating engine to generate state-reactive control panels for user devices.

Figure 3.2 presents a general architecture of the template engine. The engine has 2 inputs: a JSON template which follows a specific schema that includes the elements and options of the control panel, and the device state - a set of key-value pairs sent by the device in real-time to show its current state. For example, a light device may include information about whether it’s on or off, its current colour and its brightness in its state.

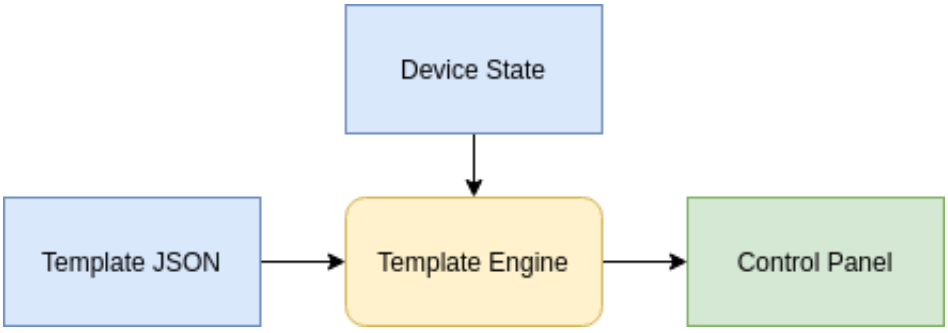

Figure 3.2: Template flow diagram

There are several different elements that the user can utilise in their templates to create rich, highly interactive and useable control panels. The engine includes both elements which only affect the control panel stylisti-

cally such as tabs and sections, and elements which can control the device and be altered by the device state, such as buttons, sliders, and more. These elements have actions attached to them, describing what should be sent to the device when they are changed - for example when a button is pressed or when a dropdown item is selected. Some elements, such as sliders and dropdowns also send an additional value property to the device when they are changed reporting their state. For example the specific dropdown item selected or the value of the slider. Finally, each elements with an action attached can also provide a data property in its template, which can have any data inside of it, but is especially useful for providing specific context information such as what tab the button is on, which helps if a template is dynamically generated. There is an explanation of each element in Section 3.3.1.

When a user presses a checkbox or does another action on their control panel, the checkbox will not change locally on their control panel. Instead, as shown in figure 3.3, when the checkbox is pressed the action is sent through VIoT to the device, which then updates its state and sends the result back to the user. This means that if there are two users using the control panel at once then they will both be in sync with each other's actions, with no extra effort being required from the maker to implement this functionality.

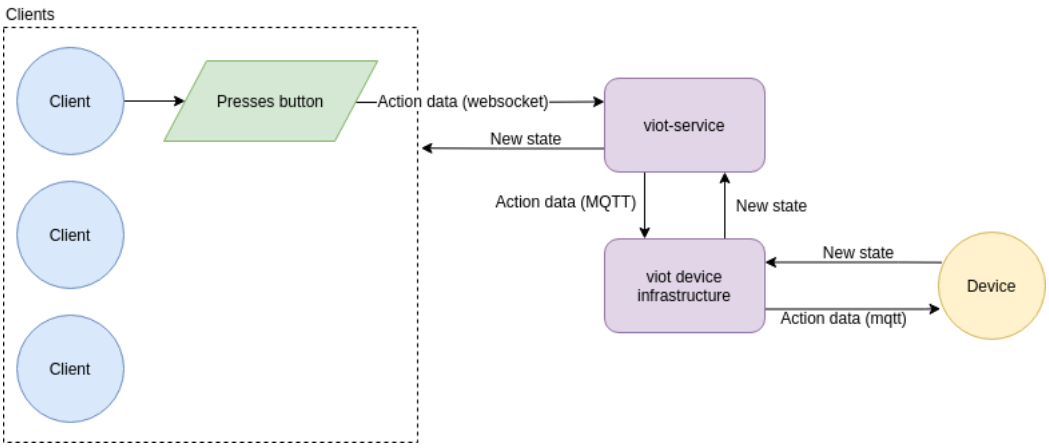

Figure 3.3: How all clients are updated when a user performs an action

3.3.1 Template Engine Elements

VIoT offers a variety of controls available to users allowing them to implement rich control panels for their devices.

- **Section** - A simple large label specifying a section of the interface. The element takes a "label" property which specifies what the label should say, and a "content" property, which is an array of elements that are part of the section. This element is useful for organisational reasons.

- **Tabs** - Horizontal tabbed navigational element which horizontally renders buttons that allow switching between different interfaces. The element takes a property called "content", which is an array of objects containing each tabs label and content. Each tab will render a button with a label, when clicked the interface will switch to the one provided in the clicked tabs content property. Only one tab can be active at once, and by default the first tab is active.

- **Button** - A highly-customisable stateless button element which can send an action specified in its "action" property to the device when pressed. It has stylistic properties "label", "active", "size", "color", "rounded" and "toggleable" which describe how the button should look.

- **Select** - A stateful drop-down list which allows the user to select one item from a list. It takes an "options" property which is an array of objects containing the items label and value. Provided an "action" property is provided, when an item is selected its value is sent to the device. The select element also takes a state property which is used to see which item is currently active on the device. A possible usage of this element is selecting the colour of an element.

- **Slider** - A control with a handle that allows users to select a specific value in a range. The slider element takes properties "min" and "max" which show the minimum and maximum ranges of the slider respectively. There is also a "step" property which reduces the amount of allowed values by limiting the size of each movement. The state property is also used in this element to show the current value of the slider. When the slider value is changed, the new value is sent to the device. One possible usage for this element would be controlling the brightness of a light.

- **Checkbox** - A checkbox control and label which can either be on or off. It only has a "label" property which allows the user to change the value of the label, a "state" property which links to a key in the device state to get the active status of the checkbox, and an "action" property which is sent to the device along with the value of the checkbox when the user presses it. This is a useful element for optional options, such as a low-energy mode for a light.

- **Toggle Button Bar** - An element which shows a list of inline buttons, where only one can be active at a time. It's similar in functionality to a dropdown list but provides a different style and more stylistic control to the user. It takes a "buttons" property which is an array of objects that define the buttons. Each button can have a "color" and "rounded" property which define the colour of the button and whether it's rounded or not, and also properties "toggleOnColor" and "toggleOffColor" which override the colour of the button if it's active or inactive. Each button is also required to have a "value" property, so that when it's pressed the value of that button is sent to the device. The "state" property of the device shows which button is currently active. An example of how this element could be used is to select a mode of operation on a fan, such as "off", "slow" or "fast".

The templating engine also provides layout elements which allow the users to modify the layout and size of their control panel. These elements are fully responsive and modify their widths based on screen size.

- **Container** - Dynamically sized container which is full-width for small screen sizes and has a limited width for larger screen sizes.
- **Columns** - A horizontal container for columns.
- **Column** - A vertical column that can have a specific size proportionate to the columns. Columns take the property "size" which can be a number out of 12, showing how horizontally large they are.

3.3.2 Template Schema

All VIoT templates follow the same schema. At the top level there are properties that describe the control panel, including its horizontal size (with the "size" property) and whether it's rounded or not (with the "rounded" property). Finally, there is a property called "content" which is an array of elements in the control panel. Each element contains a "type" property which indicates which control the templating engine should render. Other than this type property, all other properties are specific to the element. However there are conventions used, such as if the element is a control it will require an "action" property to send the action to the device, and if an element has children it will require the "content", property which is an array of elements. This is used in elements such as tabs which require content inside of them.

An example of a template for a toggle button is presented in Figure 3.4. The first line shows a "type" property which describes what element to render - in this case a toggle button. The "action" property describes what to send to the device when the button is toggled, as described in Table 3.5. The "activeLabel" and "inactiveLabel" define what the button says when it's active and inactive respectively - these two properties are unique to the toggle-button element. Finally, the state property tells the button which state key to use, described in Section 3.3.4.

```
{
    "type": "toggle-button",
    "action": "toggle-power",
    "activeLabel": "On",
    "inactiveLabel": "Off",
    "state": "POWER_ACTIVE"
}
```

Figure 3.4: Template example

3.3.3 Template delivery

A popular WiFi enabled microcontroller is the Espressif Systems ESP-8266. This microcontroller has just 4MB of flash storage, which includes the size of the users code, and it's maximum MQTT packet size is just 256 bytes, making it inconvenient to store the template on the device itself as it is very constrained. Therefore, there must be a way for the user to abstract the template away from the device if it is constrained.

There is a 'Templates' page in VIoT where users can add and preview templates, and then link them to devices. When a client requests a device's template after accessing it on the site, VIoT will check if there is a template linked to it. If there is, VIoT will be sent it to the client instead of requesting it from the device itself. In other scenarios where the device is not constrained and the user has not linked a template, VIoT will request the template directly from the device by sending the "template" message described in Section 3.2.1 - this enables templates to be dynamically generated by devices if required, as demonstrated by the lights device in Section 5.2.2.

3.3.4 Device State & Live updates

In order to generate interactive and intuitive user interface using templates, there needs to be a way for the template to display the state of the device, rather than just being a series of UI elements which aren't reactive. When the client initializes the WebSocket connection with VIoT, it also makes a HTTP request to get the current state of the device. This is a JSON serialized object of key-value pairs with appropriate values for the UI element it will be used on. Template elements can incorporate a state property which indicates which key from the state is used to modify them. A simple example of this is presented in Figure 3.5 considering a toggle button which needs to know whether it's active or not. A device state which includes the "**POWER_ACTIVE**" property is shown in Figure 3.4. In this case, the button would be toggled as the value of the property is true.

```
{
    "POWER_ACTIVE": true,
}
```

Figure 3.5: Example of device state

Since all instantiated template elements have access to the current state of the device, the element can look at its "state" property and see if it's active in the device state. If it is it can alter its behaviour accordingly, such as displaying the value of "activeLabel" when the state indicates that it's active, and vice-versa.

When the user changes the value of a dropdown element, the new dropdown value is not changed locally on the client. Instead, the action is sent to the device, which then updates its state which is picked up by VIoT and broadcast to all connected clients currently viewing the devices control panel. A benefit to this approach is that all connected clients will receive the state update in real-time, so if two or more clients were connected at once their control panels would not be out of sync if one makes a change.

Chapter 4

Implementation

4.1 Introduction

This chapter describes how VIoT was implemented, including what technologies, methodologies, libraries and tools were used. React was used to create the client for VIoT as shown in Section 4.4, while the NodeJS was the primary technology used to create the service component which utilised the large ecosystem of packages available using the Node Package Manager (NPM), shown in Section 4.3.

4.2 Software Engineering Technology & Approaches

Since both the server and client were written in JavaScript, the JavaScript IDE Jetbrains Webstorm was a perfect fit for this project, as it provides features such as intelligent code completion and refactoring, and automatic error detection. Webstorm is cross-platform for Linux, MacOS and Windows. This is the main software that will be used for running, debugging and developing the VIoT IoT platform.

Git was chosen as the version control tool for the development of VIoT, as it provides a modern version control experience by supporting branching and remote repositories allowing the code to be backed up and collaborated on. GitHub was chosen as the remote code repository due to its high availability, popularity and rich feature set.

Agile was used as the Software development methodology for VIoT operating on 2 week sprints. Agile helped deliver the project in a given timeframe while

allowing for changes to be made along the way.

Trello was used extensively as the project management tool for the project. Trello uses a Kanban-style board to organise tasks which helped visually depict what work needed to be completed next. The project has many different components and Trello helped to keep on top of the workload. The required tasks were organised into 4 different lists: To-do, In development, QA (quality assurance) and Completed. At the beginning of each sprint, a list of tasks was created and put into the To-do list. Once a task was being worked on, it was moved to the In development list, and once the task was programmed it was moved to the QA list. Finally, once the task was validated and tested, it was moved to the completed list. Figure 4.1 shows the Trello board midway through the project.

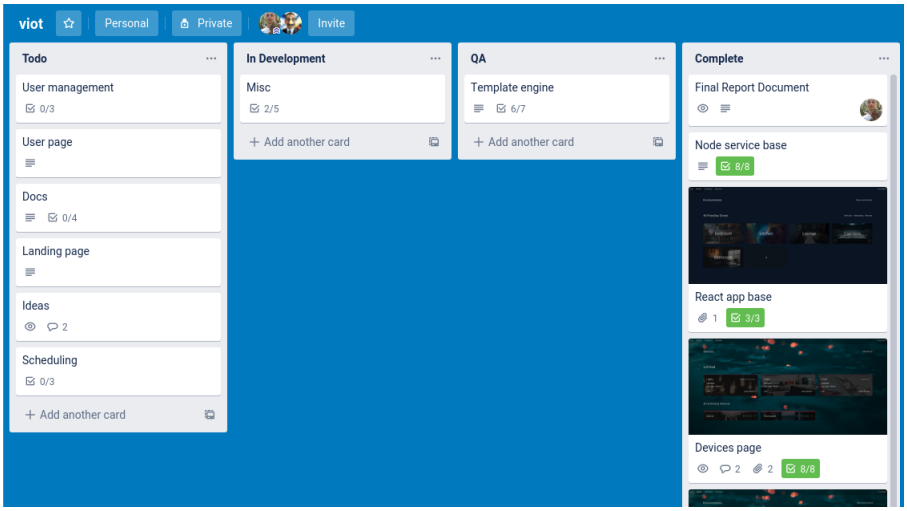

Figure 4.1: VIoT Trello Board

4.3 Service Component

The service component of VIoT implements a HTTP REST API and WebSocket interface, and utilises an MQTT client connected to a central Mosquitto message broker server, as well connecting to a MySQL database to store data. It was written in NodeJS due to its high real-time performance and large ecosystem of libraries and frameworks.

Express was used to create the REST API because it helps fast-track the development of server-based applications and is the de-facto framework of choice for many developers to create web servers in NodeJS. Alongside Ex-

press, a small library called Mel was utilised which handled routing and input validation automatically. Figure 4.2 shows code utilising Mel to create a POST endpoint with a route parameter (:deviceId), along with several required inputs (such as label and template). Mel helped create endpoints quickly without repeating code or re-implementing the same input validators in every API. As well as this, it was also possible to automatically generate OpenAPI specifications using Mel, which was very useful for debugging purposes.

```
mel.post({
    route : "/device/:deviceId",
    description : "Update a device",
    input : (input) => {
        input.user = util.loggedIn();
        input.label = mel.string("label", "Device label", {
            maxLength : 64
        });

        input.template = mel.int("template", "Template ID", {
            min: -1,
            default: -1
        });
    },
    run : async ({user, label, template, deviceId}) => {
        // Code that runs when validated
      ...
    }
});
```

Figure 4.2: POST endpoint using Mel

Figure 4.3 shows the internals of the VIoT service. It consists of a set of APIs described in Section 4.3.3 which all interact with the Mel service to create HTTP endpoints to be used by the client. There is also an MQTT client connected to Mosquitto and a WebSocket service which allows APIs to define their own WebSocket events to be used in the WebSocket server.

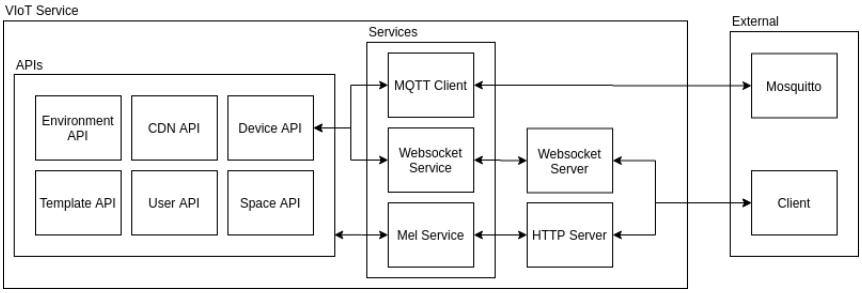

Figure 4.3: Service component structure

The service used the Passport authentication middleware for Express with its GitHub strategy, allowing easily implementation of OpenID login with GitHub. GitHub was chosen because it is a developer focussed site, as is the target audience of the product. Passport makes it easy to add other login strategies in the future. I used OpenID instead of the traditional username-password authentication strategy as to not store users passwords on the site.

4.3.1 Security

While creating VIoT, the security of the system has been taken into consideration and the architecture been designed in a way that without direct access to the servers no unauthorised user can control any connected devices.

The most obvious attack vector for the system would the theft of device API keys. If an API key was stolen, the attacker would be able to monitor which commands are sent from the user to the device, but they would not be able to see what is sent from the device to the service, or view the real devices state or template. Also, the service does not authenticate users with the traditional username/password method and instead uses OpenID authentication with GitHub, which supports multiple stage authentication, including the use of authenticator apps and hardware keys.

4.3.2 Scalability

While scalability was not a major focus point while creating the VIoT service, precautions were taken to ensure that the system can scale effectively. The majority of the service is horizontally scalable meaning multiple instances can be ran on different hardware to increase throughput, while the central MQTT server is vertically scalable, meaning more processing power can be added to increase the server output.

4.3.3 HTTP API

The majority of interfacing with the service is done with a HTTP server following the REST Pattern. The service provides APIs for user authentication, templating, environments, spaces, devices and real-device management.

The User APIs provide endpoints for user information and authentication.

When a user accesses the authentication endpoint, they are redirected to the GitHub OpenID page and are allowed to login. Once completed, they are redirected back to the API with an authentication code as a GET parameter. The service then sends this code to GitHub for verification and if it is valid the user session is set. This is all done under the hood with Passport and it's GitHub strategy.

The Environment APIs provide endpoints for fetching, creating, updating and deleting environments. Environments only have 1 property: their label, but they can have a number of spaces linked to them and also any number of users that can access and modify them. This is done with a link-table in the database which stores the environment ID, the user ID, and whether they have write privileges to that particular environment or not. By default, when a user creates an environment, they are automatically added to it and given write access. Only users which have write access can modify the environment and link spaces to it. Environments were implemented this way to allow multiple users to control and modify the same environment, but the addition of users to environments was not fully implemented to the site. The linking of spaces is done with another link table in the database, which stores the space ID and the environment ID. The space can only be linked to a single environment at any one time in the code, also through a unique key in the database.

Space APIs provide endpoints for creating, fetching, updating and deleting spaces, as well as starring/unstarring spaces. Starring spaces simply returns them first in the list when the user accesses their spaces.

The template APIs provide endpoints for viewing, creating, updating, deleting and validation of templates.

The device APIs include APIs for fetching all the available device images, creating, updating and removing devices, starring and un-starring devices, checking device states, and linking/unlinking devices from a space.

The API which adds devices first checks the number of devices that the user has and fails if they have too many. It then validates the image the user has selected and verifies that the user doesn't have another device with the same name. It then generates a unique API key using Node's Crypto module, and a binary utility is used to generate a PBKDF2 hash of the device's API key for use in the Mosquitto authorization plugin. Finally, both the API key and the hashed key along with the device label and image are inserted into the MySQL database to later authenticate connected devices.

The VIoT service utilises an MQTT client using the "mqtt" package available on the Node Package Manager (npm) to initiate a connection to the

central Mosquitto server. This connection uses a private username and password which has super-user access over the server, meaning it can subscribe to a wildcard of topics. Some device APIs utilise this connection to enable users to communicate with their connected devices. These include the device status API, which takes a device ID as input, and after checking that the device exists in the database, it publishes the status command (described in Section 3.2.1) to the central MQTT server on the device's receive topic. This then makes its way through the pipeline to the device, which responds by publishing its status to its emit topic, which makes its way back up the pipeline to the service again, allowing VIoT to respond to the API request with the devices status. This follows the remote procedure call (RPC) pattern. All device RPCs have a timeout of 2 seconds, after this period the RPC will be cancelled and the service will respond with an error.

There is also a content delivery network (CDN) API for users to retrieve device and space images. When a user requests to view their spaces or devices, a full image URL is returned in the response for the client to use to load the image. Image URLs currently point to the CDN API, but this could easily be substituted for an external CDN service in the future.

4.3.4 WebSocket Interface

As well as the REST APIs, there is also a WebSocket interface which mainly facilitates the communication between clients and devices. WebSockets enable real-time bi-directional communication on the web, and the popular Socket.IO library was used to abstract the complexities around using raw WebSockets away from the service. As well as this, Socket.IO was chosen as it provides fallback options for older browsers that do not support WebSockets. Clients can send a "subscribe" message containing a space ID through their WebSocket connection, and after verifying the user is part of the environment that the space is in, the service adds them to the device's room. This means that anything the device then sends (state update, new template, etc) is published to them and all other clients in that room, enabling all clients to get updates instantaneously and simultaneously.

4.4 Client Component

It was important to ensure the user has a fluid experience using the site. This was in part accomplished by using React and following the Single Page App (SPA) model to remove any full page refreshes. Once the client has fetched the initial JavaScript bundle from the server, the only network request that

the client must make is for data, such as fetching user information. This virtually removes any loading times. The widely-used react-router-dom library was used in the project, which uses the HTML5 history API allowing the site to still support regular routing - such as how refreshing the page will take you to the same page you were on before, even though you're not getting a different response from the server.

Several animation libraries were used to improve the user experience of using the site. Animate.css, a collection of CSS-only animations along with the react-animated-css library which provides React bindings (as shown in Figure 4.4) for Animate.css allowed animations to easily be implemented into many components in the site.

```
const PageText = (props) => {
    const { children, modal} = props;

    return (
        <Animated animationIn="fadeIn">
            <h1 className={`page-text has-text-centered ${modal ? "is-modal-text" : ""}`}>
                {children}
            </h1>
        </Animated>
    );
};
```

Figure 4.4: Example of component utilising animate.css React bindings]

The VIoT client has a "Tile" interface shown in Figure 4.5 and is meant to be as clean as possible, meaning only information which is relevant at the time is shown. This design contrasts other IoT platforms which can be complicated to view and use. The design needed to enable users to quickly access their devices with minimal effort, and this is accomplished by the large tiles used extensively throughout the site which are easy to read and click on, and also provide visual feedback such as becoming slightly enlarged when hovered over. The site is also fully responsive and works on mobile, tablet, desktop, and full HD displays. Using the Bulma CSS framework helped achieve this with it's grid based layout system based on Flexbox.

The client makes use of React's Context API to handle state management. At the root level of the site there is a provider which stores the user information, making it available to use by any other component by using the user context consumer. The templating engine also makes use of React Context to allow templating elements to access device state. As templates are dynamic it's impractical to individually pass the device state to each of the templating elements, so each element that requires the device state simply uses the device context consumer to access it, allowing for elements that

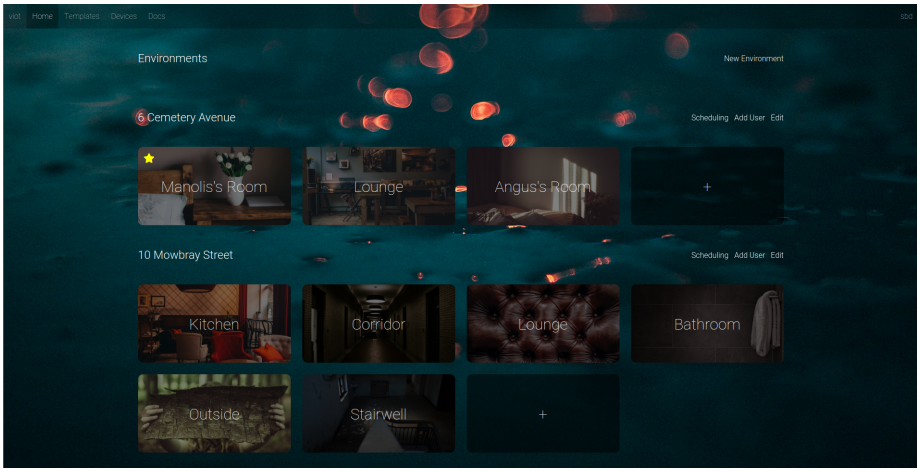

Figure 4.5: VIoT Environments Page

dynamically change based on the state of the device. Figure 4.6 shows a checkbox templating element using the device consumer to read the state of the device to see if it's checked or not.

4.5 Client Libraries

To keep in-line with the mission statement of VIoT - making IoT control as easy as possible, it was important to provide ways for clients to integrate their projects with VIoT without writing that much code themselves. Three client libraries were created, written in Node, Python and C++ which are easy to drop into projects and use, providing virtually effortless integration with the service. All three libraries have similar interfaces with language specific naming differences, such as the convention of using camel case function names in JavaScript, and lowercase underscore separated function names in Python. This means that once a user has used one of the libraries, they can easily transition to using one of the other language libraries. These libraries provide interfaces to easily connect to VIoT and send and receive messages to and from the service.

4.5.1 Node

Node has excellent JSON support with no external libraries needed to parse it. It can be parsed directly into JavaScript objects, and since the VIoT service uses JSON messages to communicate, the Node library was easy to

```
class Checkbox extends Component{
    render(){
        const {options, centered, state} = this.props;

        return (
            <Fragment>
                <deviceContext.Consumer>
                    {device => (
                        <label className="checkbox">
                          <input
                              type="checkbox"
                              className="checkbox"
                              onChange={(event) => this.checkboxChanged(device, event)}
                              checked={device.state[state]}
                          />
                          <span className="checkmark"></span>
                        </label>
                    )}
                </deviceContext.Consumer>
            </Fragment>
        )
    }
}
```

Figure 4.6: Checkbox templating element utilising device state

implement. As well as this, Node has a large ecosystem of libraries, including an excellent package for MQTT communication (mqtt on NPM). The library made full use of Node's module system, allowing users to easily import it into their projects and use. Figure 4.7 shows the use of the Node library to connect and use the service.

4.5.2 Python

The Python ecosystem and user-base has been largely fragmented between Version 2.0-3.0 as it was a major language overhaul which made a lot of syntactical changes meaning that the two versions are not cross-compatible with each other. However, Python 2 was officially deprecated in January 2020, so it was decided Python 3 would be used. A Python feature called decorators was used to allow users to bind VIoT actions to functions that run when they are called. Decorators are a software design pattern that allow the altering of a function, method, or class without having to change the source code of the declared function. An example of decorators being used to declare an action callback for VIoT is shown in Figure 4.8 on Line 3. While not trivial to implement, when implemented into a library they allow the user a pleasant development experience, and reduce code duplication. Once connected to VIoT, all the developer has to do to run code when an

```
const viot = require("viot");

viot.on("some-action", (data) => {
   // Code to run when "some-action" is sent from VIoT
});

viot.initialize({
    apiKey: "API Key",
    template: {}
});
```

Figure 4.7: Example of using the VIoT Node library to connect and use the VIoT service

action is sent is add the decorator to their function. The entire process can be completed in less than 4 lines of code.

```
viot.begin({"apikey": "125d87105914a90ab31cd8963c6d5e87bdcae8e2a05d"})

@viot.action('turn-on')
def process(data):
    viot.update_state({on: True})
    ...
```

Figure 4.8: Example of using the VIoT Python library to connect and use the VIoT service

4.5.3 C++

The C++ library is targeted at Arduinos and other Arduino compatible microcontrollers. C++ provides no support for JSON or MQTT so external libraries had to be used. Microcontrollers have significant memory constraints, so this had to be taken into consideration while creating the library. A limit of 1024 bytes per inbound message was imposed, this was necessary as a fixed-size had to be used to successfully decode JSON. Other than this, the library is as simple as the other languages to use. Figure 4.9 shows an example of the C library in use. Once the client is initialized with viotClient.initialize, the user is able to bind their VIoT template actions to functions to run when the action is called. It shows the binding of the

”some-action” action to the ”someAction” function, so when VIoT sends the action, the code inside the function will run and the developer will be able to access the data from the ’doc’ parameter.

```
#include <viot.h>

viot viotClient(espClient);
void someAction(DynamicJsonDocument doc){
    // Code to run when "some-action" is sent
}

void setup(){
    viotClient.initialize("API Key");
    viotClient.on("some-action", someAction);
}

void loop(){
    viotClient.loop();
}
```

Figure 4.9: Example of using the VIoT C++ library to connect to the VIoT service

4.6 Service and client interaction

Figure 4.10 shows the full end-to-end interaction between the user, client, service, device layer and physical device.

The device uses one of the 3 client SDKs to connect the device MQTT server using its API key. It then subscribes to its topic and publishes a connection event. The MQTT forwarding service (viot-mqtt) is subscribed to all the device topics. Once a device sends a connection event, it picks this up and forwards the message to the central MQTT broker. The main VIoT service is listening to events on the central MQTT broker. Once it detects a connection event, it will find the device on the system and if the person who created the device has an active WebSocket connection it will broadcast to them that the device has connected to the system. Anyone viewing the device on the site through a space will see it appear as online. When a user clicks to the devices control panel, they will send a HTTP call to VIoT requesting the template and current state of the device. VIoT then publishes to the central MQTT broker that it’s looking for the device. This is picked up by the MQTT forwarding service (viot-mqtt) which forwards the

message to the device MQTT broker on the right topic, which the device is subscribed to. Then, once the device reads the message, it will send another message with its state inside back up along the chain, eventually reaching the main VIoT service. If the response does not include a template, VIoT will check if there is a template assigned to the device in the database and send that to the user.

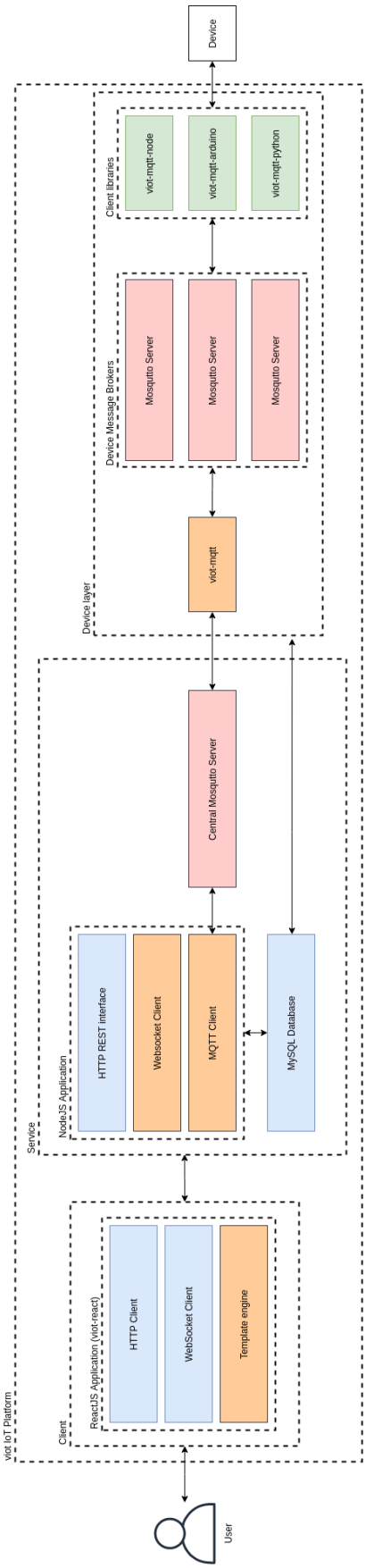

Figure 4.10: VIoT Service architecture

Chapter 5

Evaluation

5.1 Introduction

This chapter describes the test devices that were created to evaluate the functionality and usability of VIoT and also gives a critical reflection on the development of the project as a whole.

5.2 Test devices

In order to test the main platform functionality and ease of use of the client SDKs that were created, and also to add essential features that were missed, 3 test devices were created, all with different requirements.

One device is a stateless TV remote which is a stand-alone device attached to a microcontroller which used the VIoT templating delivery system. Another was a coffee machine, which involved modifying an existing 'dumb' coffee machine to add internet functionality using the IoT platform. Finally, VIoT was added to a Python LED light strip program, which involved handling device state, dynamic template generation, and more.

5.2.1 TV Remote

The first device created to test the functionality of VIoT was a TV remote - which is a simple project someone might typically work on as their first IoT device. A ESP-8266 microcontroller was used with the VIoT C++ library

to create this device.

The first step in creating this device was figuring out how to control the TV using a microcontroller. The most obvious approach was using an IR LED to emulate the TV's remote by using an IR LED pulsing binary codes. Another option was using HDMI CEC, which is a protocol as part of the HDMI specification which allows devices to control each other through HDMI. This is for instance how a TV would turn itself on when a disc is inserted into a DVD player, or how it's possible to change the TV volume using a set-top boxes remote control. Because most microcontrollers do not support HDMI and the fact that IR LEDs and IR receivers are relatively inexpensive, this approach was taken. The microcontroller and IR LED are shown in Figure 5.1.

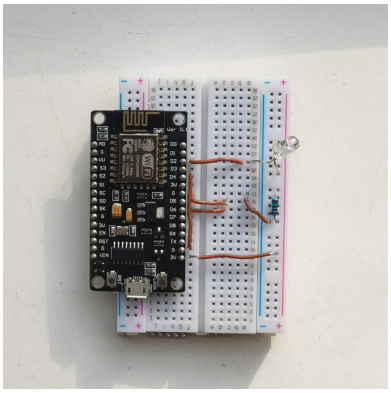

Figure 5.1: ESP-8266 with IR LED

I first used an IR receiver to read the binary codes emitted by the TV's remote control, then these codes were replayed these back using an IR LED attached to the ESP-8266 which confirmed the TV was being controlled. After doing this for each button, I now had a list of binary codes corresponding to each button on the TV remote.

Next, I created a TV remote template for the user interface. Since the microcontroller is a constrained device on storage and memory, I chose to use the template delivery method built into VIoT, which simply involved creating a new template on the 'Templates' page and saving it. Next, I added the device to VIoT through the 'Devices' page and used the C++ library to add the service to the ESP-8266 project - abridged code for this is shown in Figure 5.2. After this, the device showed as online on VIoT (shown in Figure 5.3) and I was able to add it to a space. Finally, after accessing the device through the space I was able to view the control panel as shown in Figure 5.4 and control the TV remotely.

```
void tv_power(DynamicJsonDocument doc){ irsend.sendRC5(0×C, 12); }

void setup(){
    viotClient.setApiKey(apiKey);
    viotClient.on("tv-power", tv_power);
}

void loop(){
    viotClient.loop();
}
```

Figure 5.2: Abridged C++ code for TV remote

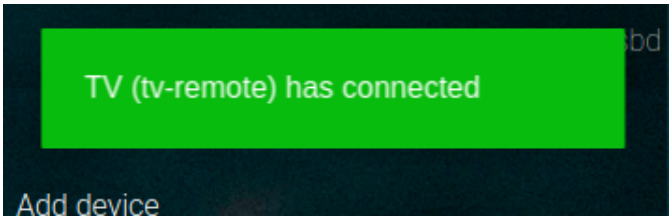

Figure 5.3: TV connected to the VIoT IoT platform

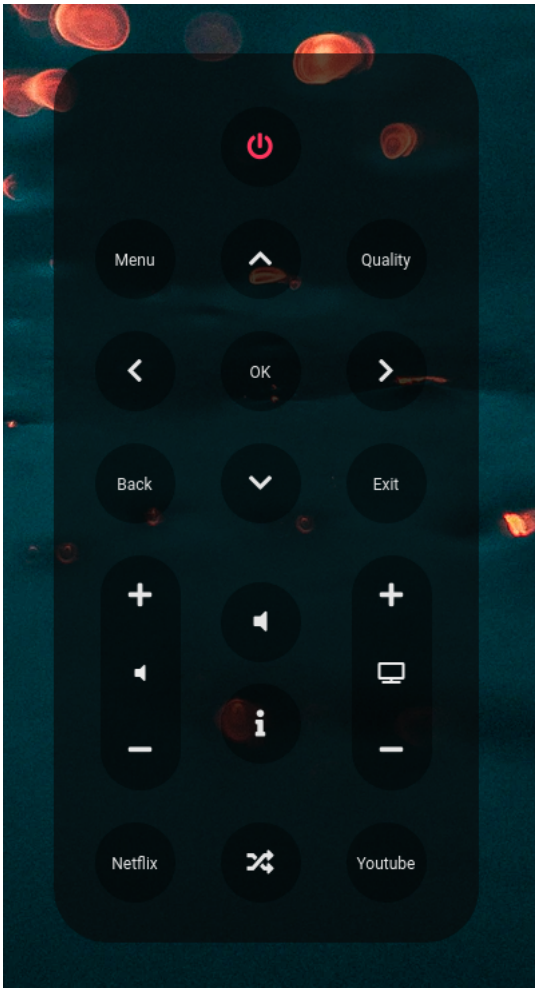

Figure 5.4: Rendered TV remote template

5.2.2 LED Light Strips

This device involved adding VIoT to an already existing Python program which independently connects to microcontrollers driving LED strips, sending them light patterns and different effects. The Python program used to run a webserver with a control panel which is only accessible from the local network, and the goal of this device was to move this control panel and its functionality to the VIoT platform, allowing the program to be controlled from anywhere.

Converting the current control panel to a template for use in VIoT would be a true test of the templating engines capabilities as it has a complex

interface including elements such as tabs, sections, sliders, drop-downs, buttons, checkboxes and more. As well as this, the programs control panel is dynamic and changes based on the program's configuration and requires the template to be synchronised with the state of the lights, such as knowing its current effect and all of the effect options. To make this process easier, a template was manually created that mirrored the HTML control panel of the program, and then this was used in the dynamic generation of the template. The Python program used AJAX on the control panel to interact with the lights, so it was simple to move the functions it used from being controlled by the web server to being controlled by VIoT callbacks.

After doing this and adding the device to VIoT, it appeared online, and it was possible to control the lights remotely after accessing the device from its linked space. Figure 5.5 shows the control panel in action. As an added benefit compared to the local control panel, the control panel syncs between connected clients allowing them to see each other's changes in real-time.

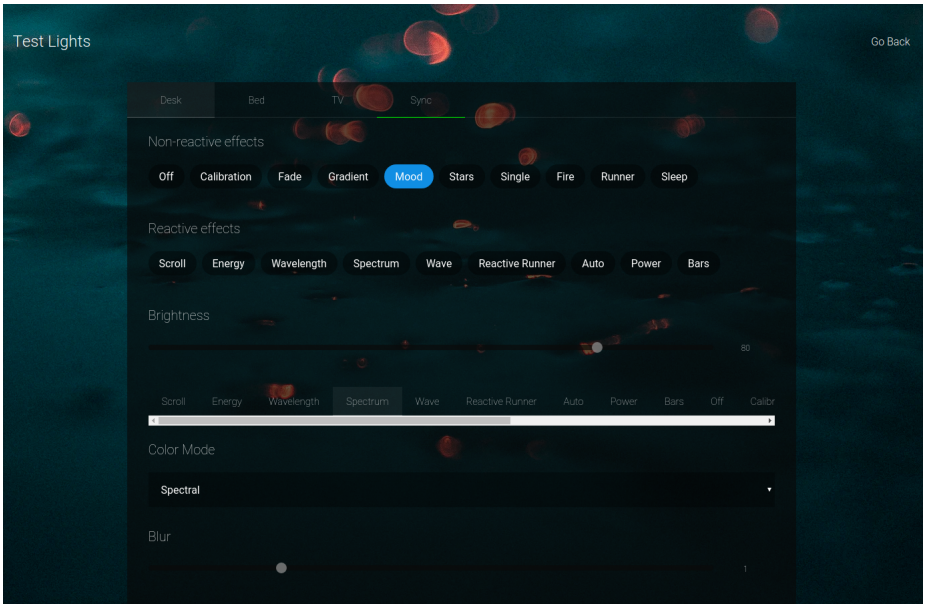

Figure 5.5: Lights device control panel on VIoT

5.2.3 Coffee Machine

Finally, I wanted to see if the functionality of an existing device could be improved by adding internet functionality, for this, a Nespresso CitiZ coffee machine was chosen which is shown in Figure 5.6. The Nespresso CitiZ has two buttons which dispense a small or large coffee when pressed. A

multimeter was used to measure the voltage, and it was found that control board used 3.3 volts, the same voltage as the ESP-8266, meaning no voltage transformation was necessary. After splicing new wires onto each of the 4 wires, I was able to connect them to the ESP-8266 through a breadboard and press the buttons with code, as well as physically. The entire breadboard and ESP-8266 were able to fit into the coffee machine internally with no modifications required to the case due to the empty space inside the machine, as shown in Figure 5.7.

Figure 5.6: Nespresso CitiZ

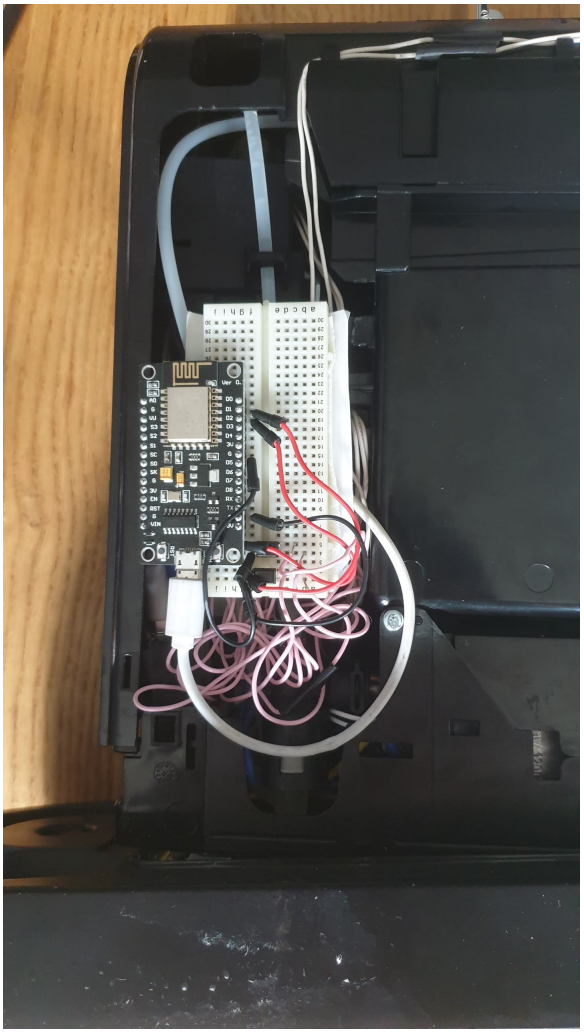

Figure 5.7: Nespresso CitiZ with an ESP-8266 inside

A simple template was created for the coffee machine and added to VIoT. Shown in Figure 5.8, the template has two icon button elements which send the ”small” and ”large” actions respectively when pressed.

```
{
    "size": 3,
    "rounded": true,
    "control": [
        {
            "type": "container",
            "content": [
                {
                    "type": "columns",
                    "centered": true,
                    "content": [
                        {
                            "type": "column",
                            "content": [
                                {
                                    "type": "icon-button",
                                    "icon": "mug-hot",
                                    "icon-color": "info",
                                    "action": "small"
                                }
                            ]
                        },
                        {
                            "type": "column",
                            "content": [
                                {
                                    "type": "icon-button",
                                    "icon": "coffee",
                                    "icon-color": "info",
                                    "action": "large"
                                }
                            ]
                        }
                    ]
                }
            ]
        }
    ]
}
```


Figure 5.8: Coffee Machine Template

I used the VIoT C++ library to interface the two buttons with VIoT through two functions, and after adding the device to the platform, it appeared as online and it became possible to access the control panel and call the two functions through the platform, pressing the buttons remotely. The rendered control panel is shown in Figure 5.9.

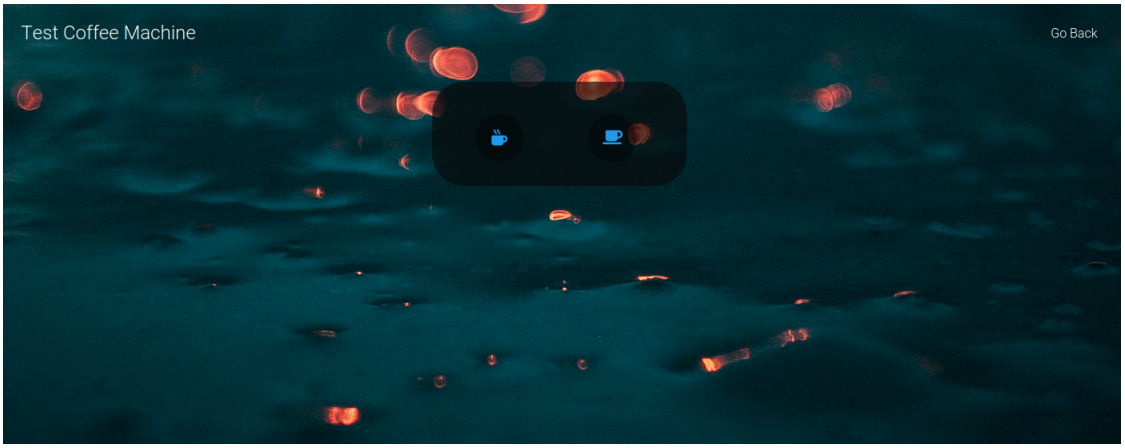

Figure 5.9: Coffee Machine rendered control panel

5.3 Conclusion & Reflection

Creating these three devices proves that the system works and is feasible for use in IoT control.

The development of VIoT went smoothly due to the large amount of planning that went into the project beforehand, which is required for a project of this size. Many different architectures were explored before settling with the current one to ensure the system provided a high level of stability and flexibility. It was useful to look at other platforms and take their best features into consideration and incorporate them into the system. The original idea for the control panels was to allow users to upload their own HTML, JavaScript and CSS, but using the templating approach yielded great benefits and greatly decreased the complexity and effort required to use the system. Templating for user control panel generation has not really been explored before, especially in the IoT world, and in hindsight using this approach seems obvious and has few drawbacks.

Creating the three test devices proved that the system works and is feasible for use in IoT control.

5.3.1 Personal development

Creating VIoT has taught me a lot about Internet of Things, including the technologies used and the different approaches that companies have taken to control and monitor internet connected smart devices. The project has also benefited me as I will use it in the future to create and control other smart devices.

Chapter 6

Final Remarks

6.1 Contributions

The main contribution of this project is the idea of using templating for user interface generation in the IoT world. VIoT explores both the idea of static and dynamic templates being sent by a device or stored away from it and ultimately shows it is possible to take the complexities of creating beautiful dynamic control panels away from the end user and allow device makers to focus on their device instead of the design and infrastructure normally required for IoT projects.

6.2 Limitations

One limitation of the project is that the templates still require some learning curve to create and their isn't extensive documentation about how to use them. Another limitation is that the complex structure of the VIoT, 3 separate projects (The client, service and MQTT forwarder) and 3 external services (Central Mosquitto server, device Mosquitto server and MySQL server) allows for a high level of scalability but this means that the system is difficult to deploy locally if a user required it to run on their local network.

6.3 Future Work

In order to combat the learning curve required to create templates and remove any real barrier to entry in using VIoT, a What You See Is What You Get (WYSIWYG) editor could be created implementing a drag and drop interface for templating elements. Each element could have code samples included with it for all 3 of the client libraries showing how to use the elements in devices.

Another way the service could be improved is by adding extensive documentation about how to use the site and templating engine, and also provide more code examples of the client libraries.

Bibliography

Amazon Web Services. (2020a). "About AWS". Retrieved May 12, 2020, from https://aws.amazon.com/about-aws/

Amazon Web Services. (2020b). AWS Internet of Things Device Management. Retrieved May 12, 2020, from https://aws.amazon.com/iot-device-management/

Angular. (2020). Introduction to angular concepts. Retrieved May 12, 2020, from https://angular.io/guide/architecture

Eady, F. (2007). *Hands-on zigbee : Implementing 802. 15. 4 with microcontrollers*. Elsevier Science & Technology.

Google DoubleClick. (2016). *The need for mobile speed: How mobile latency impacts publisher revenue*.

Knud Lasse Lueth. (2020). Why the internet of things is called internet of things: Definition, history, disambiguation. Retrieved May 12, 2020, from https://iot-analytics.com/internet-of-things-definition/

Littlefield, B. (2019). Guide to agile methodology. Retrieved May 12, 2020, from https://coolblueweb.com/blog/guide-to-agile-methodology/

Luckey, T. & Phillips, J. (2006). *Software project management for dummies*. Wiley Publishing, Inc.

Perry, M. J. (2016). *Evaluating and choosing an iot platform*. O'Reilly Media, Inc.

Pulver, T. (2019). *Hands-on internet of things with mqtt*. Packt Publishing.

Schwaber, K. & Sutherland, J. (2012). *Software in 30 days : How agile managers beat the odds, delight their customers, and leave competitors in the dust*.

Standish Group. (2014). *Chaos report*.

ThingsBoard. (2020). Dashboards. Retrieved May 12, 2020, from https://thingsboard.io/docs/user-guide/ui/dashboards/

Weber, J. (2016). Fundamentals of IoT device management. Retrieved May 12, 2020, from http://iotdesign.embedded-computing.com/articles/fundamentals-of-iot-device-management/

Zigbee Alliance. (2020). Zigbee & the smart home. Retrieved May 12, 2020, from https://zigbeealliance.org/market-uses/smart-home/