

School of Electronic Engineering  
and Computer Science

# Final Report

**Programme of study:**

Computer Science with Business  
Management

**Project Title:**

**Scalable Web  
Application for a  
Wireless Sensor  
Network**

**Supervisor:** Dr Kamyar Mehran

**Student Name:**

Sakib Chughtai

Final Year

Undergraduate Project 2018/10



Date: 24/04/2019

# Abstract

Rising energy costs across the globe has made it increasingly important to monitor how much energy we are consuming. This report presents the research and implementation of a REST API using Node.js, Express.js and MongoDB. The purpose of the implementation is to provide users with energy consumption data of the appliances in their home as well as an estimated cost. The goal is for users to make smarter monetary decisions based on their energy consumption so they can cut down costs. The report concludes that the combination of Node.js, Express, REST API for backend development and MongoDB as a database provides efficient method of creating an implementation that is scalable. Additionally, the requirements have been fulfilled and the application has been tested to ensure peak performance under load.

# Acknowledgements

I would like to thank my project supervisor for providing me with very useful advice and guiding me in the right direction throughout the project process. I would also like to thank my project partner Lenojan Jeyarajan for creating the hardware implementation that this project uses.

# Table of Contents

<b>Introduction.....</b>	<b>1</b>
Aims and Objectives.....	1
Motivation.....	2
Project Structure .....	2
<b>Background.....</b>	<b>3</b>
Wireless Sensor Network .....	3
REST API .....	4
Node.js .....	5
Express.js .....	5
MongoDB .....	6
Related Work .....	7
<b>Design and Implementation.....</b>	<b>10</b>
Requirements Analysis.....	10
Design .....	11
Implementation.....	14
<b>Evaluation .....</b>	<b>20</b>
Results .....	20
Feedback .....	23
Strengths and Weaknesses .....	23
<b>Conclusion .....</b>	<b>25</b>
Future Work .....	25
<b>Bibliography.....</b>	<b>27</b>
<b>Appendix .....</b>	<b>29</b>

# Introduction

Due to rising costs for energy across the globe, it has become increasingly important how much money is spent on the energy we use in our day to day life. Whilst there are several commercial methods of monitoring household energy usage such as through gas/electricity providers, there are only a small number of software implementations available which enable users to monitor the energy consumption of appliances wirelessly. This project will discuss the implementation of a web application that allows users to monitor the energy consumption of the appliances in their home and provide estimated costs so they can make smarter decisions to reduce the overall expenditure on electricity. The web application is to be used in conjunction with a wireless sensor system built by Lenojan Jeyarajan, who was my partner for the project.

## Aims and Objectives

This project report aims to give a complete explanation and evidence of the software development process from the requirements, design and implementation as well as analysis, testing and evaluation. Additionally, a detailed description of the hardware in relation to the software will be provided. The aim of the project is to aid users in making smarter monetary decisions based on the energy consumption of the household appliances they use. The project aims to give users an estimated cost of how much money they are spending on energy consumption.

The objective of this project is to implement a web application to wirelessly monitor appliance energy consumption and provide the user with an estimation of how much money they are spending per given time. Users will be able to select what appliance information they would like to see. The user will be able to view the energy consumption of all household appliances connected to the database. In order to connect to the required database (MongoDB), the chosen backend development platform is Node.js. The implementation will also use Express.js to create a REST API which enables an efficient method of retrieving, updating and deleting energy consumption of home appliances connected to the database.

## **Motivation**

The drastic rise in popularity of the Internet of Things and smart devices in recent years has enabled the creation of products and services which improve the convenience for consumers. One of the most prominent applications of smart devices is home automation. Home automation allows consumers to automate household activities and appliances. Specifically, home automation systems allow consumers to control appliances and lighting as well as other electronic processes such as Television. The core motivation of this project came from the goal of implementing a software implementation which improves the convenience for users in tracking the electrical appliances in their home. In particular, the aim was to create an application which allows users to monitor the energy consumption of energy appliances within the home.

After completing thorough research of existing home automation systems on the market, I knew that creating an application that only monitors energy consumption would not be enough for improving the user's convenience. I concluded that I would add another element to the application which provides the user with more useful information than just energy consumption. The solution was to provide the user with an estimated cost of each appliance based on the energy consumption. By adding the estimated cost of energy consumption to the application, this allows users to make smarter monetary decisions such as using an appliance less if they are spending more money than they would like.

## **Project Structure**

The dissertation is structured by first explaining the background of the key technologies and concepts that are used to build the web application. Chapter 2 will also discuss related works that are available for home automation. In chapter 3 there is an analysis of the requirements needed for the implementation, followed by the design and finally the implementation itself. Chapter 4 includes the analysis of the implementation and evaluates the implementation and scalability. Chapter 5 concludes the report and suggests improvements as well as future work.

# Background

## Wireless Sensor Network

With the rapid technological advancements in the internet of things and smart devices in recent years, one of the most prominent applications of this has been through the creation of Wireless Sensor Networks. Wireless Sensor Networks (WSN) are composed of a system that can sense the environment through sensors. There is an abundance of sensors on the market that are cheaply available that allow the monitoring of properties such as humidity, temperature, proximity, light and pressure. The wireless sensor network used in this project uses a current sensor to monitor the current and voltage of electrical appliances in the home.

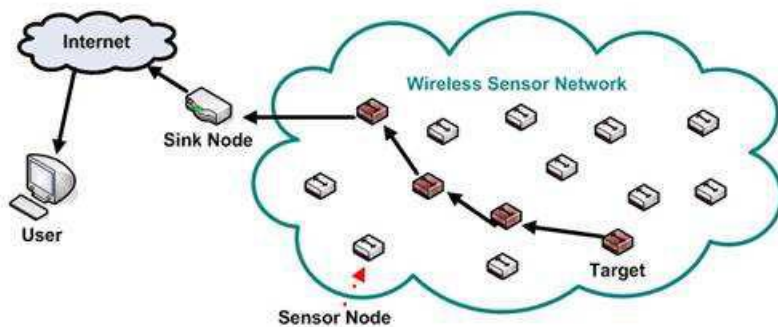


Figure 1: Diagram of the structure of a Wireless Sensor Network.

Generally, wireless sensor networks consist of sensor nodes that connect with each other using radio signals. Each sensor node contains a transceiver to communicate with other sensors (Matin, 2012). Sensor nodes also have a module which supplies power (Matin, 2012). As shown in the figure below, sensor nodes relay information back to the sink node through multi-hop communication. The sink node acts as a gateway and is responsible for sending sensor data to the internet. The sensor data that is collected is usually stored in a database (either local or in the cloud) and users can use the database to access information about each sensor in the network. Normally, this process is completed by submitting queries with conditions so users can access specific information about the sensors.

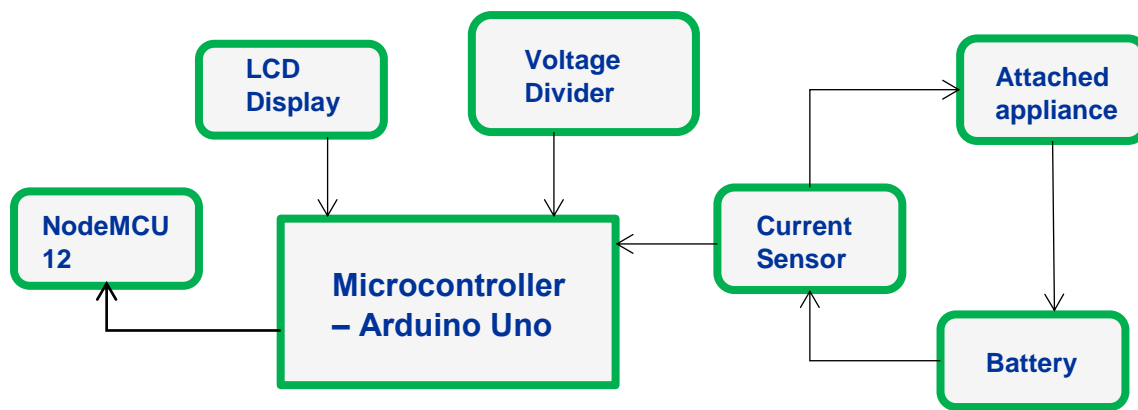


Figure 2: Diagram of hardware implementation. Courtesy of: Lenojan Jeyarajan

The wireless sensor system that is used in this project (Figure 2) monitors energy consumption through a current sensor. The battery supplies the power to the host microcontroller which is an Arduino Uno. The microcontroller acts as a host and receives the energy data and processes it to calculate the power, voltage and current. The current sensor, voltage divider, and NodeMCU module are all directly connected to the Arduino. The purpose of the NodeMCU module is to take the incoming data received by the Arduino and sends the sensor data to a database where it can be queried by the user. Regarding the attached appliances, there are a range of home appliances that can be used to measure the energy consumption. For example, appliances could include a microwave, kettle, television, games console or even a desktop computer. The design of the hardware allows for the monitoring of a range of appliances. In order to send the energy data to a database successfully, the Arduino must be programmed using the Arduino IDE to take pre-calculated values and send it to the NodeMCU module. The NodeMCU module enables the Arduino to connect to the internet, which is an essential component. By using the Arduino IDE, the energy consumption data can be passed to the NodeMCU module and then this data can be uploaded to the online database. The user should then be able to use the web application to access all the data for appliances.

### REST Application Programming Interface (API)

REST, short for **R**epresentation **S**tate **T**ransfer, is a software architecture that is normally used for website applications. It uses specific protocols to enable communication between a client and server. Typically for web applications the REST architecture uses HTTP (HyperText Transfer Protocol) to manage the transfer of data between a client and server.



REST uses HTTP to define operations and gather data depending on which request types are used. There are 4 main principles REST uses to interact with data, these are:

- **GET**, which allows to retrieve and read information.
- **POST**, which allows users to create new entities.
- **PUT**, which allows users to edit and update an entity.
- **DELETE**, which deletes an entity.

REST was chosen to be a good match for the requirements of the web application for several reasons. Firstly, the interaction between the client and server is stateless. This means that every HTTP call between the client and server already contains all the information needed to carry out the request. Both the client and server do not need to remember previous set states to fulfil the request (BBVA, 2016). Another reason why REST provides functionality to the application is the fact that the interface is uniform. REST makes use of 4 principles which provide a simple way for users to interact between the server and client.

## **Node.js**

Node.js is an open source JavaScript runtime build using Google Chrome's latest V8 engine and it often used to build network applications that are scalable (Nodejs, 2019). An application that is built using Node.js removes the need for making new threads for every request that is made, instead it runs on a single process. Node uses a framework that it driven by events and provides an architecture that allows of asynchronous input/output (I/O). This means that when operations are performed, instead of 'blocking' each request and using CPU cycles to wait for a response, Node will only resume operations when a response is made (Nodejs, 2019). One of the main advantages of using Node.js to create a web application is the fact that all the client side and server side can be programmed using one language, which is JavaScript.

## **Express.js**

Express.js is a framework that is 'flexible and minimal' that allows a fast method of creating web applications. Essentially, Express provides features which enables developers to define routes between the application's endpoints and requests from the client. The benefit of using Express in conjunction with Node is that all Express components work on top of node's main modules without limiting any of the functionality of these modules. Express is

composed of three core principles, which are the middleware, the router and routes. Functions that use middleware allow the execution of code as well as access to the response and request objects (Express, 2017).

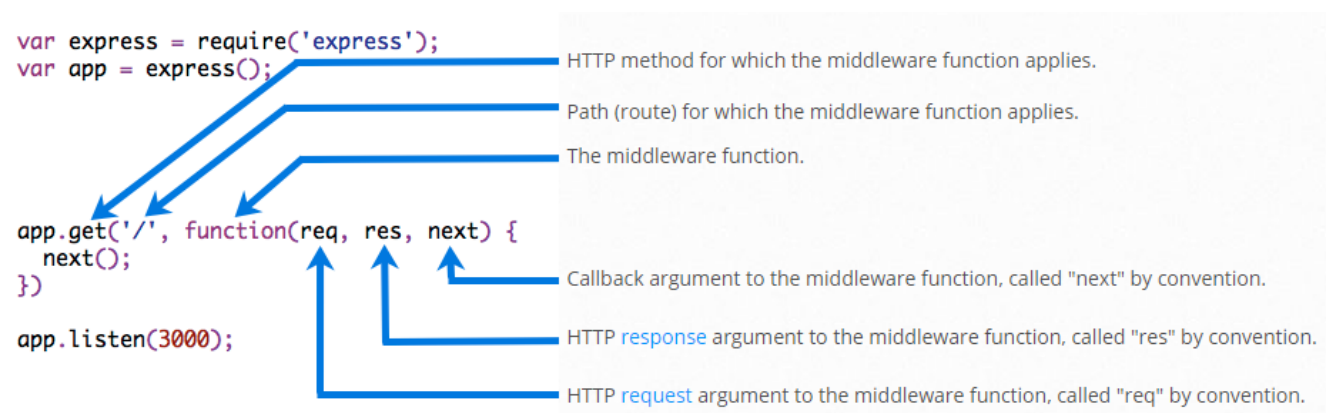


Figure 3: Components of an Express statement and its uses

Figure 3 shows how the three core principles of Express are applied within a program. On line 1, the code calls a *require* method which imports all the needed Express components to carry out the execution of the code. The function `app.get` specifies a route to a location where the application must retrieve information from, using the 'get' component. Next there is a call to a middleware function that handles the routing of every request and response call.

## MongoDB

MongoDB is non-relational NoSQL database that collects and stores documents formatted in JSON. The database can store documents that contain data such as arrays, lists and other forms of complex data (w3resource, 2019). For every document that is stored in MongoDB, each entity within that document has a unique ID that users can use to easily query information from. For example, if a user wanted to access a database that stores products and needed to find a specific item, instead of looking through the whole list of entities to find that specific item they can query using a specific ID that filters down the exact product they are looking for. An example of what a MongoDB could look like is illustrated below.

```

{
  "firstName": "Harry",
  "lastName": "Potter"
  _id: "673f346gew24568vv42004" }

```

*Figure 4: Example MongoDB document in JSON format*

For the purpose of the scalability and security, instead of hosting the database locally it is hosted online in the cloud. The cloud service is provided for free by Amazon Web Services (AWS) and the location of the cloud database is in Ireland. As mentioned before, hosting the database in the cloud provides some benefits. Firstly, databases hosted on MongoDB are secured. MongoDB makes sure data that is in transit or resting is encrypted (MongoDB, 2019). This prevents any unidentified user from accessing confidential data since only the user who created the database can access it. Encryption is provided through Secure Sockets Layer (SSL) and Transport Layer Security (TLS) (Parsons, 2017). These encryption protocols safeguard all the data stored in the database and protect data whilst requests and responses are being made between the database and local server. Additionally, MongoDB allows users to set a list of IPs' that can access the database. This is a good method of providing another layer of security to keep out intruders.

Another reason why the database is hosted in the cloud is because it provides benefits when the web application is finally deployed and scaled. MongoDB creates sets called 'shards' and distributes data across them. As the size of the database increases and more data is collected, MongoDB distributes this data equally across these shards automatically (Parsons, 2017). The technique of sharding allows deployments to be scaled easily without having to worry about restrictions if the database was hosted locally, such as disk input/output or memory usage (Parsons, 2017).

## **Related Work**

As the popularity of smart devices and home automation has drastically increased in recent years due to technological advancements, this has led to the creation of devices which can connect to the internet and can automate processes in the home that were not previously possible. An example of this is smart switches. Smart switches function by having embedded computing devices and internet capability within the switch itself. This allows for the possibility of monitoring how much energy is being consumed for that specific switch. Additionally, certain smart switches have capabilities of controlling whether electricity is passed through the switch or not. This allows consumers to turn off their electrical appliances/lights wirelessly and creates the possibility of automation. For example, consumers may wish to turn their living room lights on during set hours of the day.

Technology in the home automation industry is continuing to grow at a rapid rate. In fact, the home automation industry is valued at approximately \$20 billion with an annual growth of 22% (as of 2018), and this figure continues to grow as more and more companies dive into the world of Internet of Things (Caccavale, 2018). One of the most popular adoptions of smart home technology on a global scale has been through Amazon's Alexa and Google's Home Hub. Both devices are smart hubs that allow connection to smart devices in the home. This allows consumers to control lighting, heating, entertainment and home appliances by simply using their voice, providing the ultimate convenience. Smart hubs work with other smart devices by communicating over the local internet, however, certain smart hubs also support Zigbee or Bluetooth protocols to transfer data.

A good example of home automation technology that is worth mentioning is Home Assistant (Hass.io). Home Assistant works by using a Raspberry Pi to control and automate devices within the household. The platform is open-source and enables consumers to automate processes in the home that would otherwise need to be manually configured. For example, Home Assistant allows consumers to control lighting, monitor heating, control home appliances and control entertainment devices such as games consoles and TV's. Home Assistant provides users with an easy to use interface that runs on the local host. This can be accessed by the Home Assistant application available on Android or iPhone (iOS), or even using a browser and typing in local host along with the port Home Assistant is using. One of the main advantages of Home assistant are the available add-ons which increase the functionality of the automation within the household. For example, Mosquitto is an add-on which enables Home Assistant to communicate to devices that use MQTT, effectively it is a broker. Another add-on which provides excellent improvements in functionality is InfluxDB, which is a platform for storing data and provides real-time analytics by use of graphs. There are dozens of add-ons available which consumers can add to their Home Assistant to suit their requirements for automation and monitoring.

The accelerated growth of smart home technology can be accredited due to several reasons. Firstly, the rising costs of energy has forced consumers to think about how much money they are spending on energy consumption and what steps can be taken to reduce it. Additionally, consumers have access to more technologies at a lower cost, which drives them to adopt smart home technology. It has never been more important to keep track of how much energy we are consuming on a day to day basis. Electricity providers such as British Gas have even created devices to help keep track of our energy consumption.

However, whilst there has been an exponential growth in smart home technology adoption, there are still barricades to overcome if this technology is implemented on a global scale. Firstly, not every consumer has several hundred pounds (GBP) to implement smart devices in their home to control and automate processes within their home. Whilst prices for these technologies has decreased, they are still not at a value where it can be adopted to the masses (Caccavale, 2018). In order to overcome this issue, the most basic technologies must become more cheaply available. Another method could be for governments to partially subsidise smart devices for consumers if they reduce their energy footprint, however this is unlikely.

A significant factor which hinders further growth is security risks (Caccavale, 2018). Many consumers of smart home technology are unaware of the potential risk of an intruder accessing their data whilst they do not even know. Therefore, more research needs to be done to ensure consumers are completely aware and safe from these threats. This includes informing them of the potential risks but also manufacturers should install encryption protocols with their products as standard. Lastly, the lack of a standard between devices limits compatibility and presents a difficulty when it comes to integration of all smart devices into one system (Caccavale, 2018). More research needs to be done on developing solutions that enable an 'all-in-one' system that is easy for consumers to install and use.

# Design and Implementation

This chapter will analyse the requirements for the software implementation and discuss the process of designing the web application. This includes the architecture of the server and database. The implementation will be discussed through the sections of important code that provide the core functionalities of the application. Each method will be first defined by what it does followed by how the desired action is achieved.

## Requirements Analysis

With any creation of a software implementation, the requirements define the goals of the application as well as the tasks that it must perform. The requirements are split into functional requirements and non-functional requirements. Functional requirements are requirements that provide the system functionality, they provide a detailed insight to how the interaction between the user and application will take place. On the other hand, non-functional requirements specify how the system should act. Factors that determine how a system should act fall into several categories, these are: Security, Scalability, Performance, Availability.

### Functional Requirements

- Users must be able to view all house appliances on the database.
- Users must be able to view the energy consumption for appliances on the database.
- Users must be able to manually add an appliance to the database.
- Users must be able to delete an appliance from the database.
- Users must be able to view a certain appliance given its special ID as a parameter.
- Users must be able to edit and update an appliance information.
- The estimated cost per day/hour/month for each appliance must be correct and accurate.
- For each appliance added to the database, a unique ID must be created for that appliance.

### Non-Functional Requirements

- **Performance**
  - Each GET request must be processed by the application within 100ms.
  - Each response must be processed entirely within 100ms.

- Each POST request must be processed by the application within 100ms.
- Each DELETE request must be processed by the application within 100ms.
- Each PATCH request must be processed by the application within 100ms.
- **Scalability**
  - The database must be able to store 5000 total appliances.
  - The database must be able to store 2000 total documents.
  - The database must be able to handle 50 appliances being added per 30 minutes.
  - The database must be able to handle 50 appliances being added per 30 minutes.
- **Availability**
  - The API must be available 24 hours a day/7 days a week (once the server is running).
  - The database (MongoDB) must be running 24 hours a day/7 Days a week.
  - Users must be able to retrieve appliance information 24 hours a day (00:00-00:00)
- **Security**
  - For each request that is made, the data within the request must be encrypted.
  - For every response that is made, the data within the response must be encrypted.
  - Every appliance entity secured in the database must be secure.

## Design

This section will discuss the design process from the start and will move on to the implementation itself. The design process includes discussion of the Node.js architecture as well as the MongoDB architecture. Additionally, the REST API design will be discussed.

### Node.js

Since the application is programmed in Node.js, it is worth discussing what functionalities it brings when creating the web application. It is an essential concept within the design of the web application because it provides the user with responses to requests.

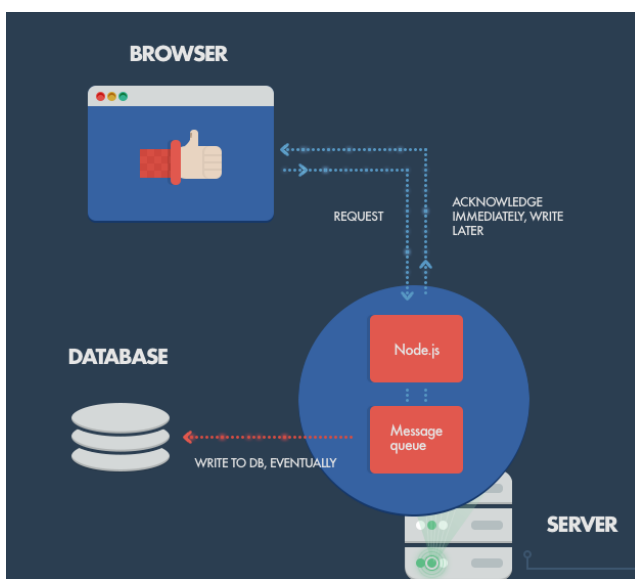


Figure 5: Diagram showing flow of Node.js events

Figure 5 shows the basic steps Node.js makes when a request is made. First a request is made from the browser, in this case, this will be a request to view an appliance from the database. From making a request in the browser, this request is sent to the local server where the request waits in a message queue (if there is any) and then proceeds to pass the request on to the database. The database reads the request and sends a response, which is sent to the server and then finally relayed back to the user in the browser.

### REST architecture

Whilst Node.js is used to make simple requests and responses, the main interaction between the server and client happens with the REST API. The REST API provides commands for users to interact with MongoDB and retrieve information about the appliances in their household.



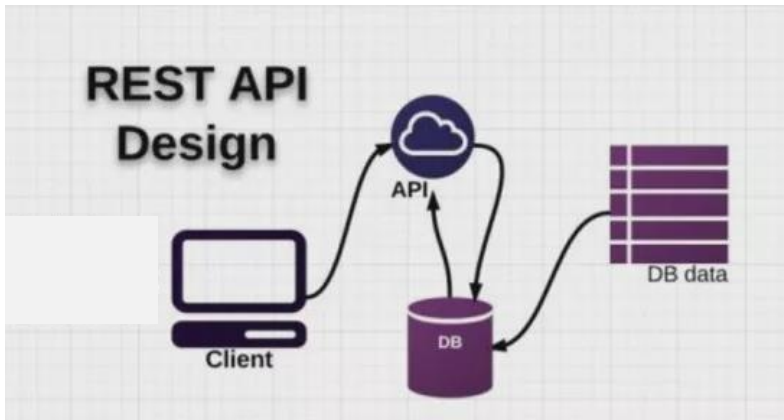


Figure 6: Diagram showing the design of REST API

Figure 6 shows the design of the of the REST API and how information is retrieved from the database. Firstly, the client makes a request using a specific request type, which could be GET, POST, PATCH or DELETE (shown in the figure below). After a request is made, the API sends the request information to the database. The database sends a response to the API once the identified database data has been retrieved. The API then sends the data back to the client where the user can view the desired request.

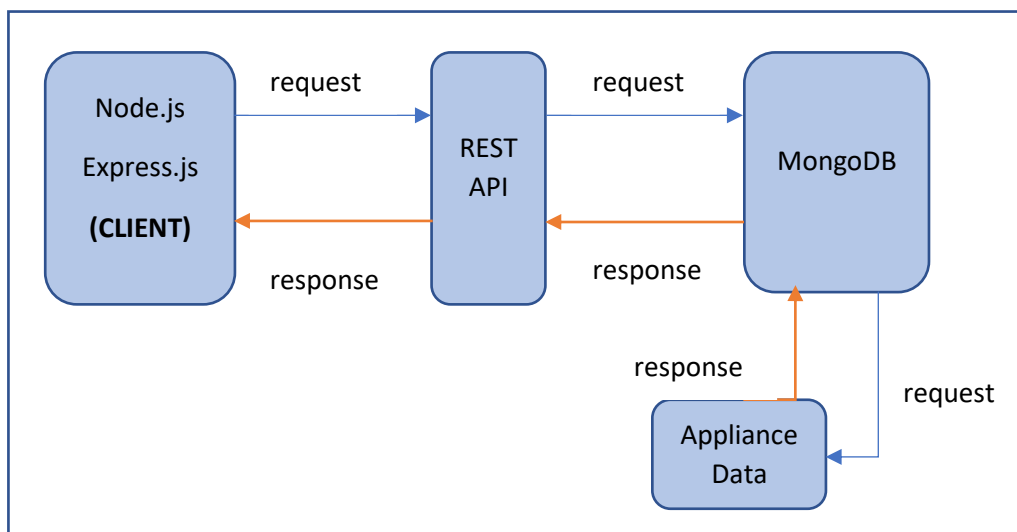


Figure 7: Final design of proposed system

Figure 7 shows the final design of the proposed system. As shown, Node.js and Express.js are used to program the client and server side. The REST API provides the necessary protocols to access MongoDB and retrieve the desired request. The flow of events is also shown in the diagram as the request is made from the client, the REST API sends that request to MongoDB with a specific request type. MongoDB then processes the request and sends back a response with the requested data back to the client.

## Implementation

This section discusses how the design is implemented and provides an overview of how each function of the REST API is used. More importantly, each method will be discussed in detail regarding what functions are being performed and how they enable the application to successfully handle requests of all types.

```
{
  "name": "app",
  "version": "1.0.0",
  "description": "Node.js REST API project",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1",
    "start": "nodemon server.js"
  },
  "keywords": [
    "node"
  ],
  "author": "Sakib Chughtai",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.18.3",
    "express": "^4.16.4",
    "mongodb": "^3.2.3",
    "mongoose": "^5.5.3",
    "morgan": "^1.9.1"
  },
  "devDependencies": {
    "nodemon": "^1.18.11"
  }
}
```

Figure 8: Screenshot showing the package.json file

Before any coding can be started, all the required dependencies and packages must be installed for the code to work. Dependencies can be installed by entering “npm install –save (dependency name)” in terminal. Npm is a package manager for JavaScript and enables dependencies and required packages to be installed. The main packages that must be installed before hand are:

- Node.js, main language used.
- Express.js, framework used for routing methods and middleware.
- MongoDB, for the database
- Mongoose, used to handle MongoDB requests.
- Nodemon, which detects changes in the source code and restarts the server automatically
- Body-Parser, which is used to parse incoming requests.
- Morgan, logs HTTP requests in the console.

Once all the dependencies and packages are installed, the source code can be successfully run without any problems. The server can now be setup for requests to be handled.

```
8  var app = require('./app');
9
10 var port = 2000;
11
12 var svr = http.createServer(app);
13
14 svr.listen(port);
15
```

Figure 9: Snippet of code for setting up server

Figure 9 shows how the server is initially set up. First, on line 8, all of the application files must be imported into the server. Imports are handled by 'require' followed by the directory where the files are located. The port on which the local server is run on is then set to 2000. This is done by creating a variable port and assigning a value of 2000 to it. The server is then created by the function http.createServer which takes app as an argument. Finally, the 'listen' function defines how the server listens to the function passed to createServer and listens on port 2000 for incoming responses.

```
13 var routeSensor = require('./api/routes/sensors');
14
15 mongo.connect('mongodb+srv://node-api:r3stfulapi@node-api-4svbs.mongodb.net/test?retryWrites
16   { useNewUrlParser: true }
17 );
18
19 app.use(parse.json());
20
21 app.use('/sensors', routeSensor);
22
23 app.use((req, res) => {
24   var err = new Error('Sensor not located');
25   err.status = 404; //404-not found
26   next(err);
27 });
```

Figure 10: Snippet of code for setting up database and error handling

The next step is to setup the routes for the sensors which will handle how requests are transmitted between the client and server. Here the routes are an example of Express.js in action. The route for the sensor requires the whole sensors route file as within this file the requests are coded. Next, the database hosted in the cloud must be connected to. This step is simplified thanks to Node.js easy commands. To connect to the database, '.connect' must be used to initiate a connection with the database, this is followed by an argument which contains the URL to the database cluster including the name of the database (node-

api) and the password (r3stfulapi). The password and username are passed along with the URL so the user has less steps to do in order to connect to the database. Next on line 19 the body parser is set up which parses the body of incoming requests. On line 21 the routes that handle each request must be setup, this takes an argument of the directory of the sensors routes folder as well as the imported routes which were setup before. Lastly, error handling is setup to throw an error if a wrong input is submitted. The method takes an argument of each request/response that is made and throws an error accordingly. On line 37 an error variable is created, and the next line sets the error status to 404, which is not found. On line 39, next(err) passes the error variable to other source files so errors can be found in all files.

```
10 var senSchema = mongo.Schema({
11   _id: mongo.Schema.Types.ObjectId,
12   name: String,
13   voltage: Number,
14   watt: Number,
15   costperHour: Number,
16   costperDay: Number,
17   costperMonth: Number,
18 });
```

Figure 11: Schema for a Sensor

Figure 11 shows the schema for each sensor that is stored in the database. Each sensor (home appliance) is first allocated a unique ID provided by the database. This allows each appliance added to be identified as a primary key if there were two of the same appliances in the database. Next, each sensor has a name of type string, an average voltage of type number, an average wattage of type number and finally the cost per hour, cost per day and cost per month also of type number.

```

19 finder.post('/', (req, res) => {
20     var sensor = new Sensor ({
21         _id: new mongo.Types.ObjectId(),
22         name: req.body.name,
23         voltage: req.body.voltage,
24         watt: req.body.watt,
25         costperHour: req.body.costperHour,
26         costperDay: req.body.costperDay,
27         costperMonth: req.body.costperMonth
28     });
29     sensor.save()
30     .then(output => {
31         console.log(output);
32         res.json({
33             message: 'Sensor created',
34             createdSensor: {
35                 name: output.name,
36                 voltage: output.voltage,
37                 watt: output.watt,
38                 costperHour: output.costperHour,
39                 costperDay: output.costperDay,
40                 costperMonth: output.costperMonth,
41                 _id: output._id,
42             }
43         });
44     })
45     .catch(err => console.log(err));

```

Figure 12: Snippet of code for posting sensor data

Figure 12 shows the method that is used when a sensor is manually added to the database. For a sensor to be added to the database, the request type POST must be used. Each post request takes an argument of the directory where the application is located and a request or response. Next on line 20 a new variable is created and equals to a new Sensor which corresponds to the schema of each sensor. The ID is assigned by the database, next the user inputs each detail such as name, voltage, watt and costPerHour in JSON format so it can be read by the database and processed. The variable is then saved and then an output is created. First the output is logged in the console in case there are any errors then the inputted details are mapped into a new array to be stored in the database. A response is generated saying “Sensor created” along with the details of the new sensor created. Finally, a .catch operator is placed to catch any errors and will return error status 404 if that is true.

```

52 finder.get('/', (req, res) => {
53   Sensor.find()
54   .select('name voltage watt costperHour costperDay costperMonth')
55   .exec().then(output => {
56     var reply = {
57       sensors: output.map (outputs => { |
58         return {
59           name: outputs.name,
60           voltage: outputs.voltage,
61           watt: outputs.watt,
62           costperHour: outputs.costperHour,
63           costperDay: outputs.costperDay,
64           costperMonth: outputs.costperMonth,
65           _id: outputs._id
66         }
67       })
68     };
69     res.json(reply);
70   })
71   .catch(err => console.log(err));
72 });

```

Figure 13: Code for retrieving all appliances in database

Figure 13 shows the method for retrieving all the appliances stored within the database. The method `.get` takes the same arguments as `POST`. However, on line 53, the sensor variable is used with `.find` to find all appliances in the database. The `.select` operation sets parameters for what appliance details should be returned. In this case, we want all appliance details to be returned, so all the variables are selected. On line 56 a reply variable is created which is formed of what details should be returned. Each variable that was mentioned in the `.select()` is mapped to the same variable name that is stored in the database. This is the process of matching the variables and determining which ones the user wants to see. On line 69 the response is generated and is formatted in JSON and all appliances are returned for the user to view. Again, errors are handled using `.catch`.

```

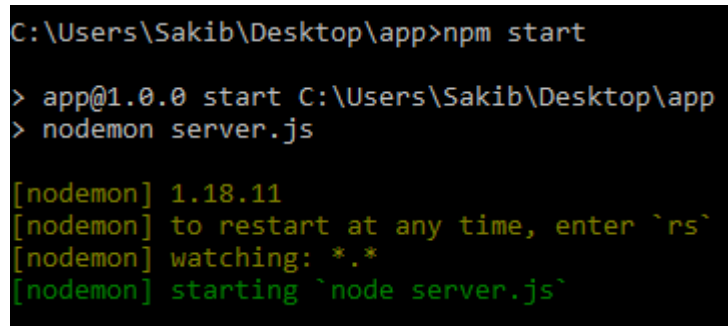
128 finder.delete("/:sensorId", (req, res) => {
129   var id = req.params.sensorId;
130   Sensor.remove({_id: id})
131   .exec()
132   .then(output => {
133     res.json({ message: 'Sensor removed from database' })
134   })
135   .catch(err => console.log(err));
136 });

```

Figure 14: Code for deleting appliance in database

Figure 14 illustrates a method to delete appliances given an ID. On line 130 the Sensor is removed by using the `.remove` operator. The `.exec` operator executes the removal of the appliance given the ID of that appliance. Then using the `.then` operator a response is

generated the user that lets them know that the “sensor was removed from database”. Errors are handled with `.catch`.



```
C:\Users\Sakib\Desktop\app>npm start
> app@1.0.0 start C:\Users\Sakib\Desktop\app
> nodemon server.js

[nodemon] 1.18.11
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node server.js`
```

*Figure 15: Terminal command to start server*

Figure 15 shows how the server is started. The server is started by entering ‘npm start’ in the terminal which initiates the `server.js` file. Once this is complete, nodemon will log in the console that the server has started using green text and write ‘starting node server.js’.

# Evaluation

This chapter evaluates the implementation as well the use of the database in the project. Evaluation is an important stage of the project because it identifies the strengths and weaknesses of the implementation through the discussion of results. It is imperative to critically evaluate how the implementation performs through testing because there are always improvements to be made. Additionally, testing allows bugs to be discovered.

## Results

This section of the report details the results that were obtained from testing the application. Testing includes making sure the right outputs (responses) are creating. Additionally, testing is done to ensure that all the requirements are met and the implementation functions as it should.

```
_id: ObjectId("5cbf8b41c04ad54be4722eba")
name: "Microwave"
voltage: 5.76
amps: 22
costperHour: 0.034
costperDay: 0.82
costperMonth: 24.48
__v: 0

_id: ObjectId("5cbfec3f77f3bc4b2869382d")
name: "Kettle"
voltage: 10.2
watt: 2000
costperHour: 0.052
costperDay: 1.25
costperMonth: 37.44
__v: 0

_id: ObjectId("5cbfed0777f3bc4b2869382e")
name: "TV"
voltage: 110
watt: 150
costperHour: 0.042
costperDay: 1.08
costperMonth: 30.24
__v: 0
```

Figure 16: Screenshot of data inputted by the user by using the POST method.

Figure 16 shows the output of a GET function (figure 13) that returns a list of all appliances. As shown, all the desired appliance variables are shown and MongoDB has successfully created a unique ID for each appliance. It is important that the unique ID is created because it acts as a primary key for identifying a specific appliance when queries are performed.



```

GET /sensors/ 200 42.567 ms - 687
GET /sensors/ 200 22.215 ms - 687
GET /sensors/ 200 21.903 ms - 687
GET /sensors/ 200 25.716 ms - 687
GET /sensors/ 200 20.607 ms - 687
GET /sensors/ 200 27.275 ms - 687
GET /sensors/ 200 26.944 ms - 687
GET /sensors/ 200 20.989 ms - 687
GET /sensors/ 200 30.661 ms - 687
GET /sensors/ 200 21.482 ms - 687
GET /sensors/ 200 23.125 ms - 687
GET /sensors/ 200 25.852 ms - 687
GET /sensors/ 200 20.214 ms - 687
GET /sensors/ 200 23.205 ms - 687
GET /sensors/ 200 20.539 ms - 687
GET /sensors/ 200 29.492 ms - 687
GET /sensors/ 200 42.713 ms - 687
GET /sensors/ 200 20.948 ms - 687
GET /sensors/ 200 20.304 ms - 687
GET /sensors/ 200 25.082 ms - 687

```

Figure 17: Response times for GET

Figure 17 shows the response times for retrieving all sensor data from the database. The GET request was submitted 20 times within 20 seconds so that the response time could be accurately identified and measured. As shown from the screenshot of the console log, the response times were in between 20.082 and 42.713 milliseconds (ms). In the requirements analysis there was a non functional requirement for performance that each GET request must be responded within 100ms. This requirement has been satisfied as response times for GET requests were more than twice as fast as expected.

```

GET /sensors/5cbf8b41c04ad54be4722eba 200 24.056 ms - 127
GET /sensors/5cbf8b41c04ad54be4722eba 200 20.361 ms - 127
GET /sensors/5cbf8b41c04ad54be4722eba 200 20.115 ms - 127
GET /sensors/5cbf8b41c04ad54be4722eba 200 22.113 ms - 127
GET /sensors/5cbf8b41c04ad54be4722eba 200 19.525 ms - 127
GET /sensors/5cbf8b41c04ad54be4722eba 200 19.860 ms - 127
GET /sensors/5cbf8b41c04ad54be4722eba 200 19.556 ms - 127
GET /sensors/5cbf8b41c04ad54be4722eba 200 20.881 ms - 127
GET /sensors/5cbf8b41c04ad54be4722eba 200 20.133 ms - 127
GET /sensors/5cbf8b41c04ad54be4722eba 200 21.454 ms - 127

```

Figure 18: Response times for GET given ID

Figure 18 shows the response times for a GET request for retrieving the details of a specific appliance (5cbf8b41c04ad54be4722eba). This request was tested 10 times in 10 seconds and the response time was significantly faster than expected. The range for the response times was between 19.525 and 24.056 ms. Each request returned with a status of 200, which the request was carried out with no problems.

```

{ _id: 5cc6372b45fd8345b8bbad7f,
  name: 'TV',
  voltage: 5.76,
  costperHour: 0.034,
  costperDay: 0.82,
  costperMonth: 24.48,
  __v: 0 }
POST /sensors/ 200 313.380 ms - 165
{ _id: 5cc6372c45fd8345b8bbad80,
  name: 'TV',
  voltage: 5.76,
  costperHour: 0.034,
  costperDay: 0.82,
  costperMonth: 24.48,
  __v: 0 }
POST /sensors/ 200 24.423 ms - 165
{ _id: 5cc6372d45fd8345b8bbad81,
  name: 'TV',
  voltage: 5.76,
  costperHour: 0.034,
  costperDay: 0.82,
  costperMonth: 24.48,
  __v: 0 }
POST /sensors/ 200 32.674 ms - 165

```

Figure 19: Response for POST request

Figure 19 shows the response times for a POST request. Within the post request, a TV is added 3 times to test how fast the request could be completed. When the first post request was made, it took 313 milliseconds. Although this is still a short time, it was 3 times the expected response time that was mentioned in the requirements. However, the following POST requests were completed between 24 and 32 milliseconds. A potential reason why the first request took much longer than the following two could be because the server was trying to establish a connection with the database. Another reason could be that there was some server lag and the response was simply delayed.

```

DELETE /sensors/5cc6372d45fd8345b8bbad81 200 30.696 ms - 42
DELETE /sensors/5cc6372d45fd8345b8bbad81 200 18.737 ms - 42
DELETE /sensors/5cc6372d45fd8345b8bbad81 200 18.307 ms - 42
DELETE /sensors/5cc6372d45fd8345b8bbad81 200 19.117 ms - 42
DELETE /sensors/5cc6372d45fd8345b8bbad81 200 47.279 ms - 42

```

Figure 20: Response for DELETE request

The last test for response times was completed using the DELETE request which deletes an appliance stored within the database given an ID. As shown in figure 20, the response times were between 18.307 and 47.279 milliseconds. These results satisfy the requirements which state that delete requests must be completed within 100 milliseconds.

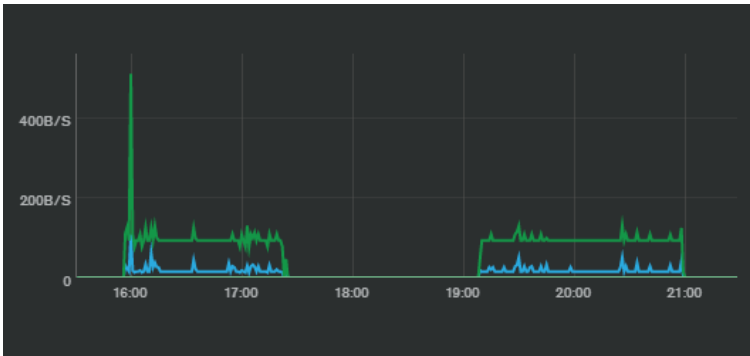


Figure 21: Graph showing network usage (Bytes per second) for two different time intervals, where green = bytes in, blue = bytes out.

Figure 21 shows the network usage of the database at two different time intervals whilst processing GET requests continuously. GET requests were submitted every 5 minutes for around an hour and a half for each time interval. The green line indicates the bytes per second incoming (requests) whilst the blue line illustrates the bytes per second outgoing (response). As shown by the graph, the database was stable throughout both intervals when GET requests were submitted. In the first interval, there is a spike in bytes incoming as it reaches 500B/s, however this can be regarded as an anomaly as it does not match the other incoming requests across both intervals which only use approximately 90-140 B/s. Responses averaged between 5 and 40 bytes/second. The reason why two time intervals were used was to make sure there were no discrepancies the processing of data. These tests were also carried out to make sure that the database was available at all times during the day (24/7), which was a non-functional requirement.

## Feedback

The finished prototype of the implementation was sent to a colleague (who will remain anonymous) for testing purposes. The aim was for him to install all the required packages and dependencies and run the application to see if he could use the application and retrieve appliance information on the same database ("node-api") that was setup in the implementation mentioned earlier. The results were as expected, he was able to connect to the database through his local server and retrieved appliance information stored in MongoDB. He used all available request types (GET, PATCH, POST, DELETE) and had success with all requests. There was a case where there was an error though. Specifically, when he tried to get appliance data for a specific ID, the console was throwing an 'UnhandledPromiseRejection' and the connection to the database was closed. This error

resulted from an error catch being created the wrong way, which led to the promise rejection. After the error catch was coded in the correct format, there were no problems.

## **Strengths and Weaknesses**

After completing the implementation of the REST API for a wireless sensor network, it is clear that there are some strengths but there are also weaknesses that can be improved on. Firstly, one of the core strengths of the implementation is that it is scalable. At no point during testing were there issues with bottlenecking. This can be accredited due to MongoDB and its sharding techniques which prevent bottlenecking. Additionally, the response times for each request type were significantly faster than expected in the requirements analysis. On average the response time was 50 milliseconds quicker than expected. It is worth mentioning that the use of a REST API to manage the database is very efficient, scalable and provides users with a simple way of interacting with data.

Whilst there are several strengths, there are also weaknesses present within the implementation. One of the most obvious weaknesses is that there is no front-end for users to interact with. The lack of the front end means that users have to use the browser and local host to manage queries, or an application (such as Postman) which allows the testing of an API. A simple front end would improve the user's convenience and also increase the overall usability of the implementation. Another weakness is the lack of graphs to visualise trends in energy consumption. Whilst there are entities such as cost per month that predict how much money you are spending on energy consumption for an appliance, there is no visual aid to help the user to make a smarter decision and cut down costs. Additionally, another weakness is that the implementation is limited to house appliances because of the setup of the hardware. This is a weakness because it prevents other processes in the home that use electricity from being monitored. An example is lighting, which consumes a large proportion of total energy consumption within a household. The fact that lighting is not being monitored highlights a weakness within the implementation because it fails to consider another source of energy consumption that is quite important.

# Conclusion

The goal of the project was to implement a scalable web application for a Wireless Sensor Network. Based on the research and testing completed, it can be concluded that REST API provides one of the most efficient and scalable solutions for managing Databases. The combination of Node.js, Express.js, REST and MongoDB provides an effective method of creating a useful web application, all programmed with one language (JavaScript).

The implementation has been evaluated through load testing which guarantees that the application will still perform effectively under high load. The application can handle over 50 requests per minute without crashing or sending the wrong response. The response times to requests illustrate that the implementation performs processes very quickly (average of 30 milliseconds). Whilst the requirements have been met, there is still room for improvement. Specifically, a front end must be implemented to allow users to access appliance data in an easier way. Additionally, more unit testing should be done to find any bugs in the code and assure stability of the implementation.

## Future Work

In order to improve the project, there are several additions that would be made in the future. Firstly, a front end would be added that is simple and easy to use. This will make it easier for users to interact with the application when retrieving house appliance energy consumption data from the database. Specifically, a graphical user interface (GUI) would be greatly suited for this improvement. The GUI would be created using React since it could be easily implemented with the back end of the web application. Another improvement for future work would be to add graphs for each appliance where the energy consumption is shown per day/week/month. The benefit of adding graphs is to visualise data to users so they can understand trends in their energy consumption, and this could help them make smarter decisions to save money. Graphs could be added to show energy consumption (y-axis) against time (x-axis) for all appliances or just a specific one. Additionally, graphs could also show estimated cost (y-axis) against time (x-axis).

One other useful addition to the project would be to add recommendations to the user based on trends in energy consumption. A method of deciding how each recommendation would be created would be applying an algorithm to identify trends or patterns in the data. From the identification of a trend, a recommendation could be made, such as "Turn your

computer off between 7-10pm". Several different algorithms could be applied such as linear regression, logistic regression, classification or Naïve Bayes. The advantage of using algorithms to detect trends in data is that they could provide valuable insight into where energy consumption is costing the most money. Additionally, these algorithms could provide predictions of future energy consumption based on the same values used to identify patterns. Overall, the analysis of data provides the user with a much clearer idea of where they may need to cut down on energy for certain appliances and in turn save the most money. Lastly, queries could be improved to return more useful information. For example, a query could return all the appliances that are costing more than £25 per month.

# Bibliography

- Anon, (2016). *REST API: What is it, and what are its advantages in project development?*. [online] Available at: <https://bbvaopen4u.com/en/actualidad/rest-api-what-it-and-what-are-its-advantages-project-development> [Accessed 1 Jan. 2019].
- Anon, (2019). *Introduction to Node.js*. [online] Available at: <https://nodejs.dev/> [Accessed 19 Feb. 2019].
- Caccavale, M. (2018). *Council Post: The Impact Of The Digital Revolution On The Smart Home Industry*. [online] Forbes.com. Available at: <https://www.forbes.com/sites/forbesagencycouncil/2018/09/24/the-impact-of-the-digital-revolution-on-the-smart-home-industry/#263f06483c76> [Accessed 29 Jan. 2019].
- Docs.mongodb.com. (2019). *Read Data from MongoDB With Queries*. [online] Available at: [https://docs.mongodb.com/guides/server/read\\_queries/](https://docs.mongodb.com/guides/server/read_queries/) [Accessed 17 Feb. 2019].
- Docs.mongodb.com. (2019). *TLS/SSL (Transport Encryption) — MongoDB Manual*. [online] Available at: <https://docs.mongodb.com/manual/core/security-transport-encryption/> [Accessed 12 Jan. 2019].
- Expressjs.com. (2019). *Express routing*. [online] Available at: <https://expressjs.com/en/guide/routing.html> [Accessed 10 Jan. 2019].
- Expressjs.com. (2019). *Writing middleware for use in Express apps*. [online] Available at: <https://expressjs.com/en/guide/writing-middleware.html> [Accessed 15 Feb. 2019].
- Fredrich, T. (2018). *HTTP Methods for RESTful Services*. [online] Restapitutorial.com. Available at: <https://www.restapitutorial.com/lessons/httpmethods.html> [Accessed 14 Feb. 2019].
- Home Assistant. (2019). *Hass.io*. [online] Available at: <https://www.home-assistant.io/hassio/> [Accessed 15 Jan. 2019].
- Kocakulak, M. (2017). *An overview of Wireless Sensor Networks towards internet of things - IEEE Conference Publication*. [online] Ieeexplore.ieee.org. Available at: <https://ieeexplore.ieee.org/document/7868374> [Accessed 8 Jan. 2019].
- Matin, M. (2012). *Overview of Wireless Sensor Network*. [online] Intechopen. Available at: <https://www.intechopen.com/books/wireless-sensor-networks-technology-and-protocols/overview-of-wireless-sensor-network> [Accessed 15 Jan. 2019].
- Medium. (2017). *Why Would You Use Node.js*. [online] Available at: <https://medium.com/the-node-js-collection/why-the-hell-would-you-use-node-js-4b053b94ab8e> [Accessed 16 Feb. 2019].
- Node.js. (2019). *Overview of Blocking vs Non-Blocking | Node.js*. [online] Available at: <https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/> [Accessed 11 Jan. 2019].
- Parsons, N. (2017). *MongoDB Atlas — Technical Overview & Benefits*. [online] Medium. Available at: <https://medium.com/@nparsons08/mongodb-atlas-technical-overview-benefits-9e4cff27a75e> [Accessed 12 Feb. 2019].

Singhal, R. (2018). *Restful API Design*. [online] Medium. Available at: <https://medium.com/@rachna3singhal/restful-api-design-95b4a8630c26> [Accessed 12 Jan. 2019].

w3resource. (2018). *MongoDb Tutorial - w3resource*. [online] Available at: <https://www.w3resource.com/mongodb/introduction-mongodb.php> [Accessed 22 Dec. 2018].



# Appendix

```
75 finder.get('/:sensorId', (req, res) => {
76   var id = req.params.sensorId;
77   Sensor.findById(id)
78     .select('name voltage watt costperHour costperDay costperMonth')
79     .exec()
80     .then(output => {
81       if (output) {
82         res.json(output);
83       } else
84       {
85         res.json({ message: "No valid entry found for provided ID" });
86       }
87     })
88     .catch(err => console.log(err));
89 });
```

Figure 22: Method to find sensor given unique ID

Request Type	Parameter	Description
GET	/sensor/{id}	Get information of specific sensor using ID
POST	/sensor/new	Create new sensor by posting information of new sensor
PATCH	/sensor/{id}	Update information of existing sensor
DELETE	/sensor/{id}	Delete Sensor from database

Table 1: Table showing request types and parameters