Smart Home

A system for monitoring and controlling the home.

btRooke

Contents

Analysis

Introduction

Thanks to almost exponential advances in technology as well as the constantly growing smartphone market, components for creating smart Internet of Things devices are becoming cheaper, more advanced and easier to work with.

Smart homes are becoming more and more of a common occurrence due to this. Although many of the current systems that are very effective and work well, a vast majority have several critical flaws including the following:

- They are subscription based; take for example the company 'Ring'[1]. Their product (a smart doorbell) can function without a subscription however when the subscription service is not used it loses the critical feature to review video of past events due to needing cloud storage space on Ring's servers.

- Can rely very heavily on companies' services or servers hence they need an internet connection. Many current systems will lost their 'smart' functionality when an internet connection is lost[2].

- Furthermore, many of these companies harvest data from the devices, which the user may not approve of or even cease to function if the internet is down. An example of this is the company 'wifiplug'[3] who state in their privacy policy[4] that they will record the users 'energy consumption, usage patterns, trends or habits' which may be undesirable to a user who wants their data to be confidential.

This is why my aim for this project is to build a system where the user has their own server in the house, with no to very little dependency on external companies and the internet which can control IoT devices in their home. Additionally, the user has full access of their data as it is stored in a local server in their home and will (hopefully) be cheaper to implement.

---

[1] https://shop.ring.com/pages/faq

[2] https://www.techadvisor.co.uk/how-to/digital-home/what-happens-smart-heating-when-wi-fi-or-power-is-down-3588749/

[3] https://wifiplug.co.uk/

[4] https://wifiplug.co.uk/privacy/

End Users

The end users for this system will be me, my family with the main supervisor being my father.

So to get a grasp of what they would like to see in a system such as the one I posed them the question: “what features would you want in a smart home system?”

The following are some of the answers I received:

- The ability to “see if devices are switched on or not”.
- The ability to “access the system easily”.
- Feature that allows you to “switch devices on and off”.
- A “device scheduling” feature.
- To be “easy to use”.

I thought that these were all very valid and useful features. For example, seeing the status of a device and then turning it off with a system will allow the user to reduce energy consumption and, furthermore, can let the user have peace of mind that they have turned a device off. This was mentioned in the context of hair straighteners by my sister.

The suggestion to “access the system easily” could imply many things, so I questioned further and found out that they would like the system to be quick to get onto and use, unlike loading up a computer and opening a program and typing in a multitude of passwords to get onto.

Additionally, on this topic, they said that they would much rather access it from their mobile devices than a computer as this would be much easier and much more convenient. I completely agree with this point as it would almost defeat the point of a system if it was more difficult or tool longer to accomplish a task with the system than without and using a mobile device, which we have on us virtually all the time, is much more convenient than a PC.

Another feature mentioned is device scheduling, setting when devices will be turned on and off. This has the potential to be a very useful feature as it could be used to switch on lights when the time goes past a certain time boundary.

Current Systems

To decide further what features I would like to implement in my system I researched the current leading systems and examined and evaluated which features of theirs I may like to include.

Taking ‘wifiplug’ as the first company to examine, some features they include are:

- ‘Control your home from your phone’ - This feature allows you to monitor the statuses of the plugs in your home from a mobile device. I think this would be a key feature of my system as being accessible quickly from mobile device is one of the main points of a smart home. Furthermore, seeing the data live is an feature which I would like to include.

- ‘See real-time energy data for your appliances’ - This feature is quite self explanatory and very interesting however I think perhaps slightly redundant in the modern day as the user doesn’t really need to know how much current their devices are drawing in everyday use. On the other hand a device such as this could be very useful for the diagnosis of faulty electrical appliances.

- ‘Schedule timers’ - This is a very useful feature; allowing users to schedule when devices will turn on is very powerful as they could accomplish many simple tasks with it.

Secondly, looking at the company ‘La Crosse Technologies’[5], a feature of their home monitoring systems is temperature and humidity sensing; I think that this could be a very useful and interesting feature to implement.

The company ‘ring’[6] makes smart doorbells which have features including:
- Motion detection
- Video recording
- Voice chatting to someone at the door

Many of these features are quite advanced however I think inspiration can be drawn from their system.

[5] https://www.lacrossetechnology.com/products/specialty-items/monitoring-systems/temperature-humidity
[6] https://en-uk.ring.com/

Most of these systems are interfaced with by a mobile app, I can see the merits of this doing this: it is very easy to use and will look nice. However, it is hard to develop apps for multiple systems like this for reasons explained later in this section.

Proposed Solution

After reviewing the research from the previous sections and deciding what features to include, I have come up with this proposed solution to the problem.

The server and control hub for this system will be a Raspberry Pi from which many IoT devices will be managed via a python script. A Pi is a good way of controlling the devices as it can always be on: it uses low power and yet has enough processing power for what we need.

It will offer up a API-like set of commands which can be sent to it to control the system, I think that this is a good way to manage the system as then the system is incredibly modular.

For example, the interface I plan to include out of the box is a website, however someone who is more tech-savvy may wish to design and program their own interface or they may even write a program to log data about when devices are turned on and analyse it.

The controller API will utilise an SQL database to store its user, device and any other data that may be needed.

Referring back the website I mentioned I would be including in the box to control the system, this will be hosted on the same Pi that is running the IoT device manager but could easily be hosted elsewhere. The website will be able to use the device manager API to provide an easy to use human interface for controlling the devices. It will make requests to the controller server's IP to control the system.

Using a website for this has several advantages:

- A website can be accessed from almost any device; this makes it easy to control your home from any smart device, much like many of the current systems. If a website was not to be used an application for every device would have to be created which would be very difficult to manage and maintain.

- A website can be port forwarded through the user's router at home allowing access to their smart home's control and monitoring from anywhere in the world with an internet connection.

Having the controller be controlled by an API is useful for many reasons:

- If I wrote an all in one controller and interface program, it could only really be accessed by 1 person at a time which is a problem if a family of many people need to access it at the same time.

- The API server can be threaded meaning many users can connect and interface with the system at once.

- Web browsers are available on almost all smart devices and hence so will my system; the need to develop multiple versions is eliminated.

To make the IoT devices, I have researched and chosen WiFi capable micro controllers; using a micro controller rather than a fully fledged OS on a Raspberry Pi, for example, is better for the following reasons:

- It is more secure: only the code written on the device is executed and hence vulnerabilities such as a default credential ssh service could not be accidentally left on.

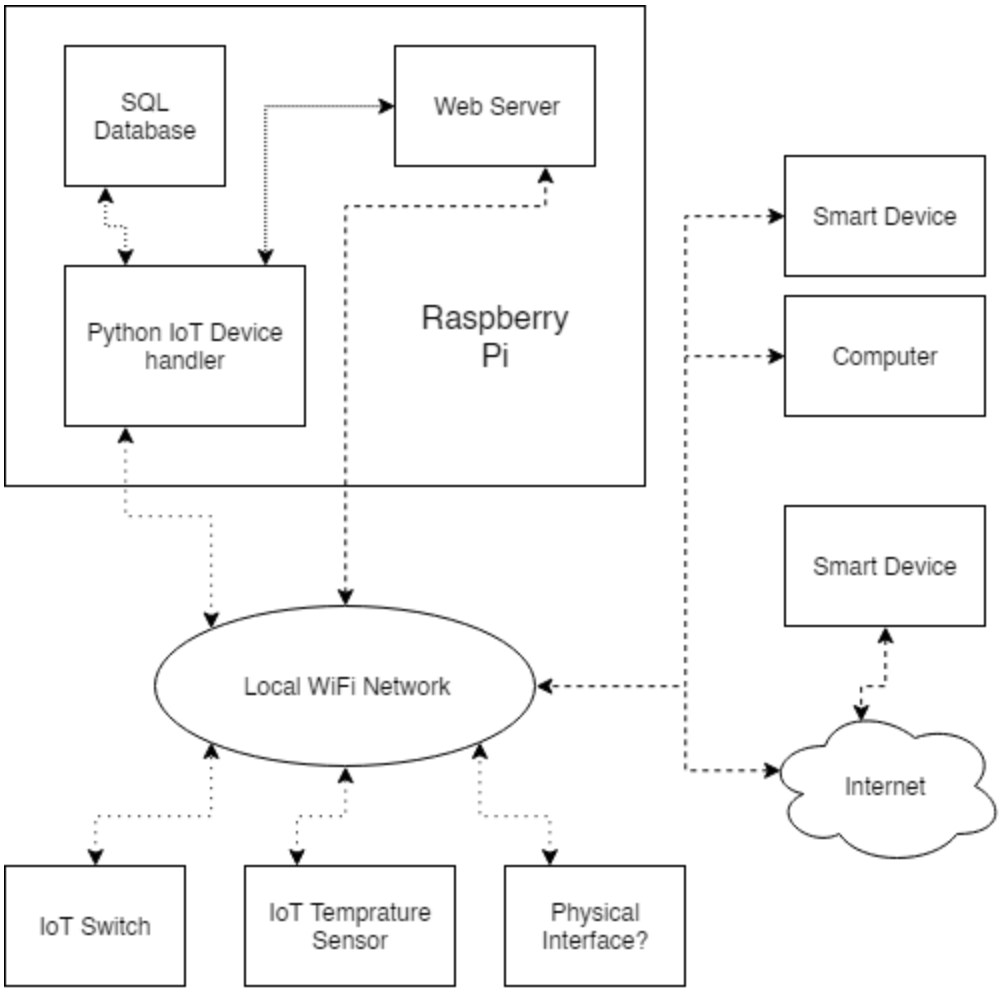
- Consumes less power as is not doing as much as a full OS would be.

- Has less redundant computing power than a raspberry pi, for example, would do; this reduces the price of the system considerably.

The IoT devices will be communicated with via sockets over the client's home network; the controller server/ API and the IoT devices will be on the same network. The devices will be secured using an authentication token to ensure that they can't be controlled by any malicious device that isn't the controller on the network,

To conclude, the system will have 3 main parts: the controller API, the website and the IoT devices. The website will talk to the controller API and the controller API will talk to and manage the IoT devices. The diagram below shows the interaction between the three:

Data Flow Diagram



Data flow diagram showing the proposed system; the patterns of the lines represent which nodes can talk directly to others.

Acceptable Limitations

After talking in more detail to my main client - my father - who is interested in the potential of smart home systems we have agreed the following acceptable limitations over discussion:

- Limited functionally of devices - Due to my lacking electrical knowledge, the devices may not be fulfill their purposes fully. However the programming side of the devices will be fully functional and will server as a solid proof of concept that an electrical system could be attached successfully.

- Lack of configuration options regarding the user system - When designing a user system, I think that there are an incredibly large amount of features that could be implemented. For example username changing, password resets, privacy settings and profile pages. Therefore we have agreed that only the necessary features for the project needed would be added to this system.

Objectives

Objectives for the program to fulfil:

- Must have a controller API which controls IoT devices on the client's network:
    - Will store data about users and devices in an SQL database.
    - Has a user and permissions system so only certain people can use certain devices.
    - Many users can access the system at once.
    - Allow clients to send and receive parameters.
    - Will allow devices to be interacted with.
        - Data can be sent to, and be received from, the devices.

- A website interface to access and control the system:
    - Will call call the API functions and generate and modify the webpage based. upon the responses.
    - Must be easy to access and use.
    - Accessible from mobile devices.

- IoT devices:
    - Must be easy to use.
    - Take commands from a controller server via sockets.
    - Be able to be connected to the user's WiFi network.
    - Only take commands from the controller, not other devices; perhaps by the user of an authentication token.

Documented Design

As seen in my analysis, this system is split into 3 main parts: the API, the devices and the website. The following sections contain the design of each of them.

Controller API

Flask

The first thing I designed was the API for the controller. After some research on how to execute a good API, I decided to use a RESTful API and use the python framework Flask[7] to do so.

The Flask framework allows you to assign a python function to a URL using a flask function wrapper; the device which requested that url will get back what the function returns.

```
# An example of a function set to a directory with Flask

from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"
```

I will send and return values using the JSON format which is very common for RESTful APIs. Here is an example command that could be sent to the API using the linux command line tool ‘curl’:

```
curl localhost:5000/create_user --data '{"new_username": "bob",
"new_password": "123"}' -H 'Content-Type: application/json'
```

[7] http://flask.pocoo.org/

This command would sent the values “new_username” and “new_password” via POST to the directory create_user. The Flask API can then access these values with the dictionary request.json[parameter].

The header “Content-Type: application/json” must be applied to the request to allow Flask to process the incoming data as json.

Flask can also read GET parameters, which will be useful for later in the project.

API Commands Table

In the table below I lay out what commands are going to be in the API, what parameters are to be used and and what parameters are to be returned.

This is the final table, however I started by designing simply the first few commands such as log_in and create user and added commands when designing further sections when needed:

Command	**Parameters**	**Returns**	**Misc**
login	● username ● password	● status ○ code ○ text ● session_key	session_key is random 32 random characters.
log_out	● session_key	● status ○ code ○ text	Deactivates a session_key; removes it from SQL database.
create_user	● new_username ● new_password ● email*	● status ○ code ○ text	email could be used for account recovery if password is forgotten.
list_devices	● session_key	● status ○ code ○ text	

		• devices (list)
		  ○ id
		  ○ name
		  ○ owner
		  ○ type	

add_device	• device_ip
	• auth_key
	• device_name
		• status
		  ○ code
		  ○ text	

is_session_key_valid	• session_key	• status
		  ○ code
		  ○ text
		• is_valid
			is_valid will be text string “true” or “false”

get_device_info	• session_key
	• device_id
		• status
		  ○ code
		  ○ text
		• device
		  ○ id
		  ○ name
		  ○ owner
		  ○ type	

call_command_on_device	• session_key
	• device_id
	• command_code
		• status
		  ○ code
		  ○ text
		  ○ (extra parameters dependant on command called)
			device_response may contain different things depending on the command called.; see device response table.
			
			May also not exist if not needed.

get_device_permission_info	• session_key
	• device_id
		• status
		  ○ code
		  ○ text
		• permission
		  ○ permission_id
		  ○ type	

		○ users (list)	
get_users_list	None	● status ○ code ○ text ● users (list)	users is simply a list of usernames of users of the system.
get_permission_types	None	● status ○ code ○ text ● types (list)	types is simply a list of types of permission on the system.

Values with * after are optional to supply.

API Parameter Validation

Parameters Constant

At the top of many of my API functions, a constant, `PARAMETERS`, may be found; its use is to hold information about the parameters to be supplied.

Is a dictionary with keys of the parameter names to be supplied which map to other dicts with more information about the parameter such as whether it is required, it's datatype or it's RegEx validation string.

However for those that do have it, it is very space efficient as then an if statement doesn't have to be written for validation for every parameter.

Here is an example from the create_user command:

```
PARAMETERS = {
    "new_username": {
        "RegEx":"^[\w\d]+$",
        "required": True},
    "new_password": {
        "RegEx": "^(?=.*[^\w\d]+.*)(?=.*[A-z]+.*)(?=.*[\d]+.*).{8,}$",
        "required": True},
    "email": {
```

```
        "RegEx": "^[^\s]+@[^\s]+\.[^\s]+$",
        "required": False}
    }


    parameter_is_valid = lambda parameter:
re.match(PARAMETERS[parameter]["RegEx"], request.json[parameter])

    invalid_parameters = list(filter(lambda param: not
parameter_is_valid(param), parameters_included))
```

Regex Strings Table

The below table shows parameters of commands and their RegEx string (if they use it).

API Command	Parameter	ReGex String	Misc
login	username	^[\w\d]+$	
	password	^.{8,}$	
create_user	new_username	^[\w\d]+$	
	new_password	^(?=.*[^\w\d]+.*)(?=.*[A-z]+.*)(?=.*[\d]+.*).{8,}$	Must be more than 8 chars and contain a letter, a number and a special character.
	email	^[^\s]+@[^\s]+\.[^\s]+$	Must be in form something@example.net
get_device_info	device_id	^[0-9]+$	
add_device	device_ip	^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$	4 dot-separated groups of numbers of length 1 to 3 digits I.E. an ipv4 address.
	auth_key	^[A-z0-9]{32}$	
	device_name	^(\w|)+$	

call_command_on_device	device_id	^[0-9]+$	
	command_code	^[a-z]{3}$	
get_device_permission_info	device_id	^[0-9]+$	
set_device_permission	device_id	^[0-9]+$	
	permission_type	^\w+$	
	users	^\w+$	users is a list so this string must be checked against all elements in the list.

Session Key Validation

Session keys are validated using the command from utilities: is_session_key_valid(key).

The validation process is first using a ReGex string to validate that it is a session key in valid form and then simply checking it against a list of session keys which are stored in the database

API Response Object

Normally, Flask simply returns the string you return from the function, however using a Flask Response object, you can add headers to a response which are needed for reasons such as allowing the API to be called from any host.

To make this even easier for my usage, I wrote a class for a JSON API response:

```
class JSONAPIResponse:
    def __init__(self):
        self._content = {}
        self.set_status(1)
```

```
    def set_status(self, code, text=""):
        if not text:
            text = get_error_message(code)

        self._content["status"] = {"code": code, "text": text}

    def add_content(self, content, content_id):
        self._content[content_id] = content

    def build(self):
        response = Response(json.dumps(self._content))

        response.headers["Access-Control-Allow-Origin"] = '*'
        # Allows API to be called from any origin.

        response.headers["Content-type"] = "application/json"
        # Tells receiving machine that it's getting json.

        return response
```

The set_status method sets the status parameter of the JSON response and also pulls the corresponding text (if none is supplied) from a file called error_message.json in the same directory of the program.

API Response Codes

As seen in the previous class, my API always returns a parameter called status; this holds 2 more parameters status.code and status.text. This way a client can tell if their request was successful.

Below is a list of response codes.

Code	Text	Misc
1	Success	
2	API Use Error	
3	Server Error	

4	Missing Parameters	Missing parameters listed in text.
5	Username already exists	
6	Invalid Parameters	Parameters don't match ReGeX string.
7	Incorrect Password	
8	Username doesn't exist	
9	Insufficient Permissions	
10	deviceID not in use.	
11	Problem connecting to device.	
12	Device, if exists, did not respond in time.	Request timed out.
13	General problem executing a command on a device.	See controller log for more detail.
14	Auth key incorrect.	Applies only to add_device command.
15	Non successful device response.	
16	Device ID doesn't exist.	
17	Invalid command for this device.	

User System

I designed this system is such that a user can log in and has a session_key. When the user logs into an account, a session key is generated and stored in the SQL database, it should also be stored by the user (in our case this will be in a cookie for the website). They then have to send this cookie to call any more commands which require user permissions, this works well for the following reasons:

- The API client doesn't have to store the user's username and password for every time a command is called, it only has to store a session key which gets deactivated every so often so even if a malicious entity gets hold of it, it will not work forever.

- The user doesn't have to log in every time they want to access the the system: if a session key is stored as a cookie for example then it simply needs to be pulled from the cookies and sent.

The session keys are linked to the users with many to one to relationship in the SQL database.

SQL Database

The database I am using runs on mariadb, which is virtually MySQL. It runs on the controller hub Pi and is connected to by the python controller as needed.

Below is a description of the SQL database I used. Much like the API Commands table, this is not the final table, I simply started by designing the user and password tables and added as needed.

Only especially important column properties are shown, see code appendix for more detail.

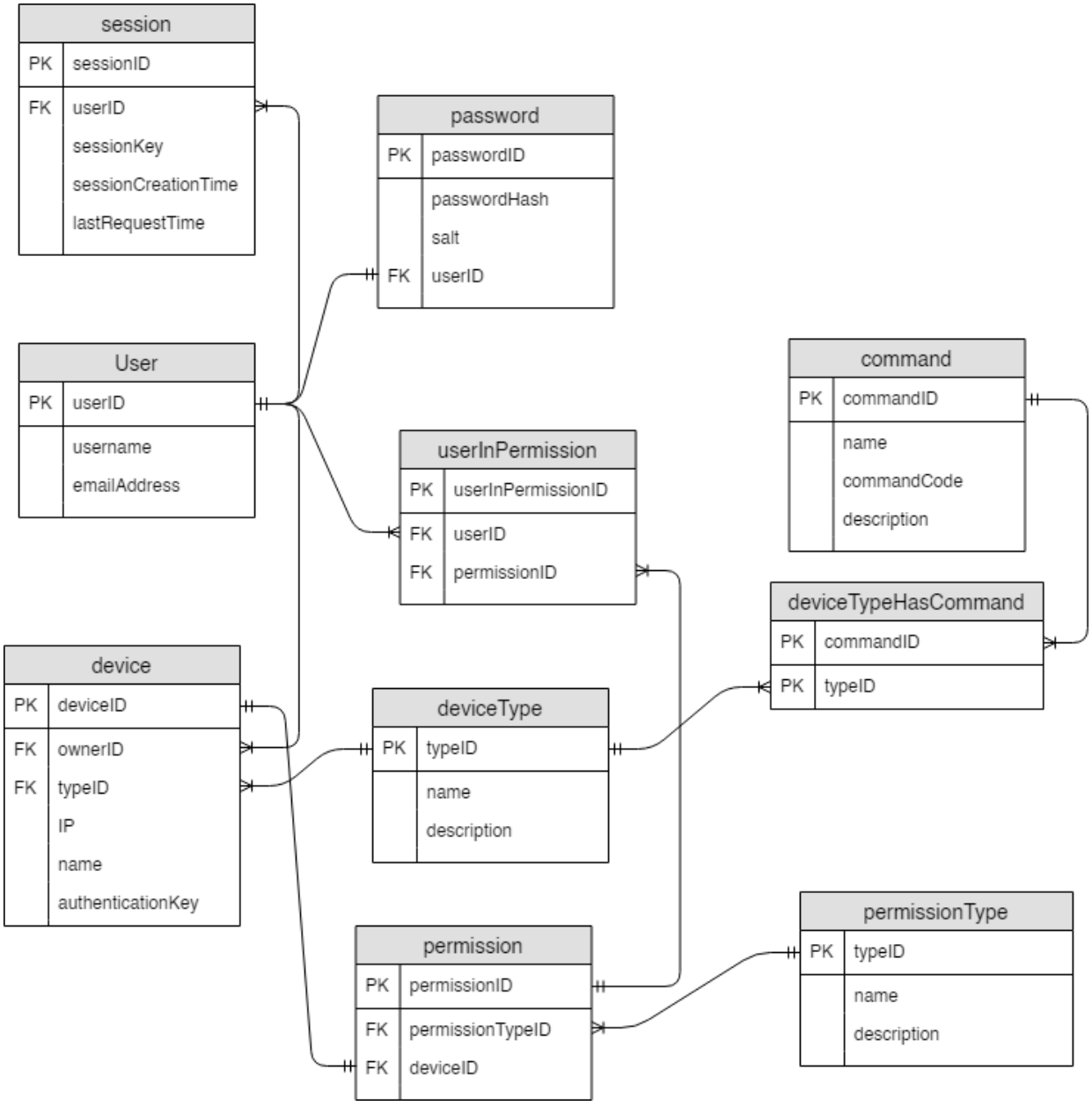
SQL Tables

Only especially important data types are shown here; to see in more detail see the SQL section of the technical solution.

user	**password**	**session**
PK userID UNIQUE NOT NULL username emailAddress	PK passwordID passwordHash salt FK NOT NULL UNIQUE userID	PK sessionID FK NOT NULL userID sessionKey UNIQUE sessionExpiryTime lastRequestTime
device	**command**	**deviceType**
PK deviceID FK ownerID	PK commandID name	PK typeID name

FK typeID authenticationKey IP name	commandCode description	description
deviceTypeHasCommand	**permission**	**permissionType**
PK commandID, typeID	permissionID FK permissionTypeID UNIQUE deviceID	PK typeID name
userInPermission		
PK userInPermissionID FK userID FK permissionID		

ER Diagram



Brief Description of each table

- **user** - Holds a userID, unique username (which they will use to log in) and (optional) email address which could be used for account recovery.

- **password** - Stores a SHA256 hash of the user's password. See later section for more detail. userID is unique meaning a 1 to 1 relationship is formed between user and password.

- **session** - Holds a random 32 character string which is the session key and has a userID field forming a many to one relationship with the user table: a user can have many sessions.

- **device** - Holds the device name, authentication key, IP and typeID. This forms a 1 to 1 relationship between the device table and the deviceType table; a device can have 1 type.

- **command** - Holds a command's name, description (used for website) and the actual command that is sent to a device, a 3 character command code.

- **deviceType** - Stores the different device types, their names and a description.

- **deviceTypeHasCommand** - Many to many linking table between device Type and command as not all devices have all commands, all take some generic ones but others are more specialised for specific device types.

- **permission** - A permission for a device used to store data about which users can access said device; has a permissionType, see later pages for list of types.

- **permissionType** - Stores the permission types.

- **userInPermission** - Links users and permissions in a many to many relationship.

Storing Passwords

Passwords are not stored directly; they are salted with a random 32 character salt appended on the end and then hashed with SHA256.

This is not a 100% secure method of storing passwords as SHA256 is not a suitable cryptographic hash function: it doesn't take long to generate a SHA256 hash of a string.

Therefore a hacker, if they got hold of the salt and hash, may just run a brute force program with my salt appended to all of the most common passwords which would not take too long to crack the hash.

However this is certainly better than storing the passwords in plaintext.

SQL Device Types

The values in the below table are stored in the deviceType table and are referenced as foreign keys in the device table to ensure database normalisation.

Device SQL Name	**Description**
Thermometer	A device which can be switched on and off via the control system.
Switch	A device for viewing the temperature of wherever the device is placed.

Database Connection Class

For simplicity and the code's cleanness, I wrote a database connection class to allow me to connect to the database easily:

```
DATABASE_DETAILS = {
    "host":"localhost",
    "user":"controllerUser",
    "passwd":"_b>/8KT4>-U/wKv$",
    "db":"controllerDB"
}

class DBConnection:

    """
    Class to be used in conjunction with a 'with' statement as to
    ensure correct usage; connection is closed and opened correctly.
    """

    def call(self, command):
        cursor = self._db.cursor()
        cursor.execute(command)
        to_return = cursor.fetchall()
        cursor.close()
```

```
        return to_return

    def last_insert_id(self):
        return self._db.insert_id()

    def __enter__(self, details=DATABASE_DETAILS):
        self._db = db = pymysql.connect(**DATABASE_DETAILS)
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        # exc parameters passed automatically by python

        self._db.commit()
        self._db.close()
```

As mentioned in the docstring, this is to be used in with clause to ensure proper closing and opening of the database.

Permissions

Before executing a command on a device, the system must check if the user has permission to do so.

The way I have implemented this is such that each a device has a related permission record in the pemission table. This permission record has a type and then (via a many to many linking table) users are assigned to a permission.

When a user calls a function from the API that is related to a device they must supply a session key; the session key is linked to their account in SQL and so we can see which users are linked with which device permission and find out whether they should be allowed access.

Permission Types

Permission SQL Name	**Description**
blacklist	Disallows all people on this list.
whitelist	Allows only people on this list.

disallow_all	Allows no one but owner.
allow_all	Allows everyone.

The system works out whether the user requesting a command is allowed to do so based on what the permission type is and whether they are linked to the permission.

If users are linked to a permission to which they are irrelevant, they are simply ignored E.G. if I set the permissionType to be allow all and linked some users to that permission, nothing unexpected would happen.

Website

Site Overview

Before programming I created this basic overview which shows the directories I want my site to have and what they will have on them:

All sites redirect to /login if session_key is not valid or present

Index	**GET Parameters**	**Misc**	**Design Overview**
/		Redirects to /login	Nothing
/login		Redirects to list_devices if valid sessionKey already present in cookies.	Login field with username and password fields.
/list_devices		Redirects to login if sessionKey is missing or invalid.	List of devices currently on the system with their type and owner. The user should be able to click on them to access more detailed settings. A menu bar at the top to access the other settings.
/device	deviceID	Focused page for managing a device. GET parameter is used certain device page can be saved in browser as favourite.	A page of controls for a device such as main device features and permission controls. A menu bar at the top to access the other settings. See Fig.1 below for a mock up of the permission element of the design. Created this as was quite complex and so a reference was needed during creation.
/create_user			Fields for a new username and password.

/add_device			Fields for the device new name, IP and authentication key.

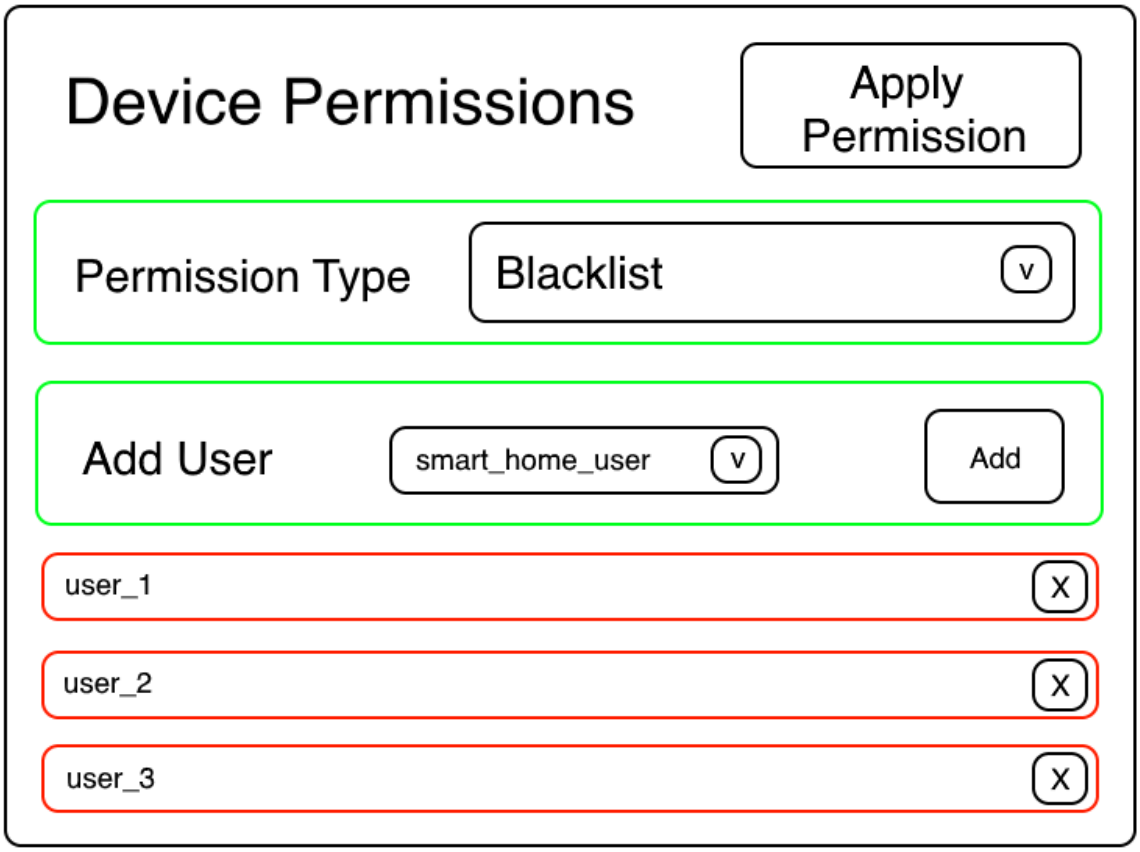


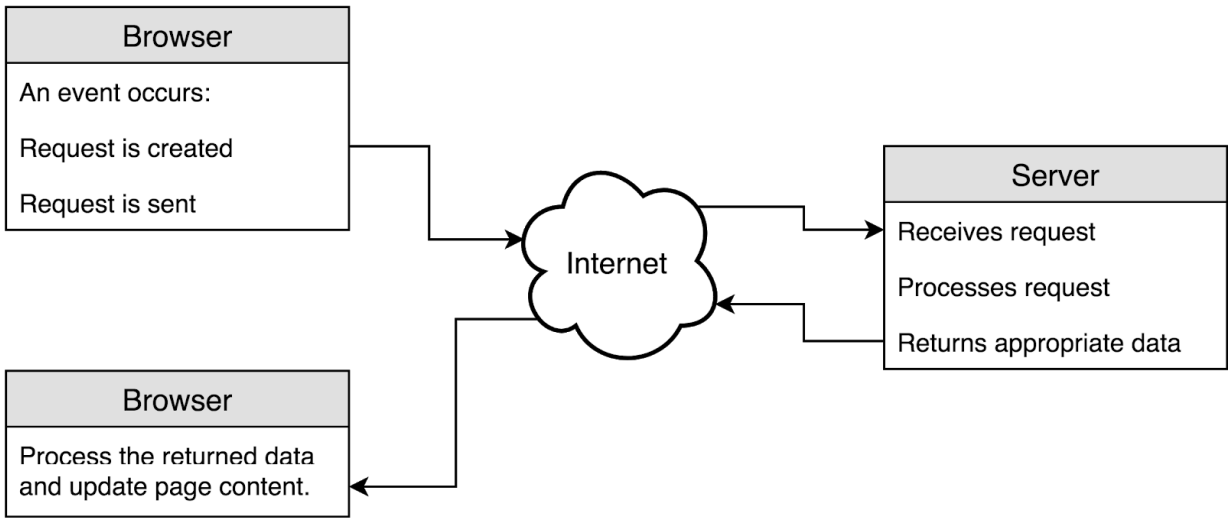
Figure 1: A mock up of the permissions element.

AJAX

The website I am designing will have no server side code execution; it will work entirely with AJAX, Asynchronous Javascript And XML. AJAX is a methodology allowing a site to be changed after creation based upon events and requests.

I think that the diagram below (inspired by W3schools'[8] explanation) explains it very well.

[8] https://www.w3schools.com/xml/ajax_intro.asp



AJAX works very well for this system as when buttons are clicked (JS functions are invoked) on the website, requests can be sent to the controller API asking it to do and return things.

Javascript API Request Class

To make calling the API from the website a lot easier, I have designed the following API request class:

ApiRequest(command)

- Private Attributes
  - postValues- POST data which we are going to send.
  - command - The index to which we want to send the request I.E. log_out.
  - calledOnReponse - Function which is called on response from the API.

    Due to the asynchronous nature of these AJAX requests, a function must be set to be called when the request is responded to.

    Must take a response parameter for the response object to be passed to it, response text can be accessed with response.responseText.

    The default function called on reponse is simply:
    `function(response){alert(response.responseText)}`

- Public Methods
  - setCalledOnResponse(func)
  - setCommand(command)

- addContent(content, parameter) - Adds content to the postValues object, with key as parameter and content as content.
- send - sends the request and calls the function calledOnResponse when it is received (if successful).

Here is an example of how it could be used:

```
req = new ApiRequest("get_user_list")

req.setCalledOnResponse(
  function(reponse){
    if (response.status.code == 0){
      alert(response.users)
    };
  }
);

req.send();
```

This *very* basic example shows a request for a list of users and the function called on response alerts the user list when it is received successfully.

JS HTML Manipulation

To display the results of my request to the system nicely, I can't just simply `alert(response)`; instead I can use JS to manipulate the HTML Document Object Model (DOM).

JS has very nice integration with this; for example you can get an element (any set of `<these id="idName"></these>`) using `document.getElementById(idName)` which will return an element object. This can be manipulated very easily with methods such as `appendChild()` and attributes such as `innerHTML`.

Using these I can build dynamic pages which change based on what the responses are from the API.
I won't go into full detail on each page here, however below is a pseudocode walkthrough of how a page is manipulated.

Pseudocode for HTML DOM Manipulation

The pseudocode below represents adding a device description box, much like one that would be used in list_devices, to the page.

```
addDeviceToPage function is called with the parameter of a device object:

deviceElement = create element of type div
Set the classname of deviceElement so it gets the correct CSS applied
Set its id to be the device.id of the device parameter that was passed

Create sub element we want to go in our device element such as text and
buttons for the device name etc and store them in a list: subElements

For element in sub elements:
      Append child element to deviceElement

Set specific attributes for each sub element such as id or onClick

Append child deviceElement to the body of the HTML document
```

Devices

Hardware

As mentioned in the analysis (for the reasons stated in the analysis), I am using a programmable microcontroller for the devices. The one I chose, in the end, is the NodeMCU ESP8266[9]. This device is perfect for the following reasons:

- It is relatively cheap, sitting at £5-10 on the internet
- Has WiFi connectivity (a key needed feature)
- Has a micro USB serial interface port, making incredibly more easy to program as it does not need other hardware to interface with PC.

Other options included the Arduino with WiFi shield[10] (which was very expensive) and the Raspberry Pi zero W[11] which is hard to get hold of and poses security threats and wastes computing (and physical) power due to the reasons stated in the analysis.

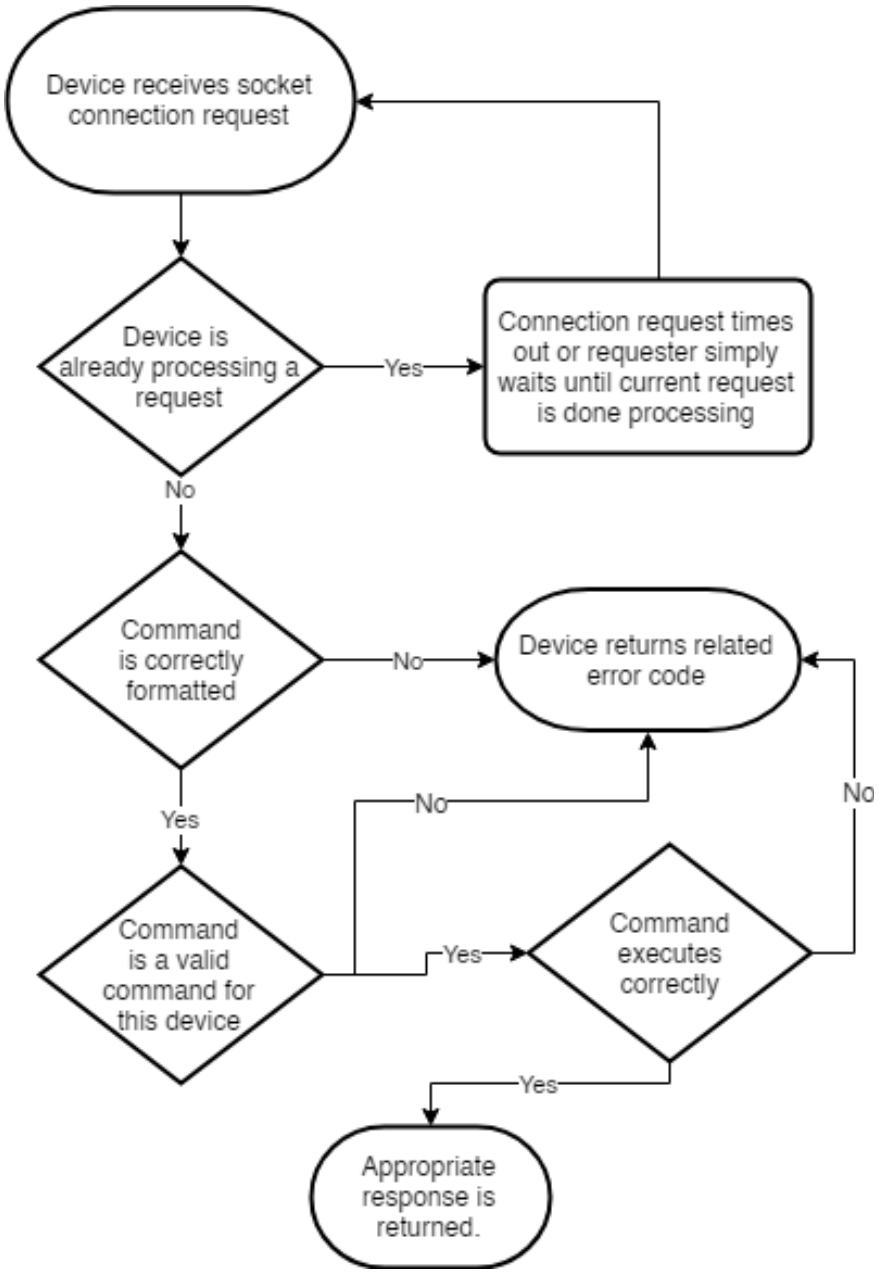
I decided to install the micropython[12] firmware on the ESP8266s which allows me to write in python (a language which I am far more familiar with) rather than C/ C++.

Communicating with the devices

The devices work by being on the same network as the controller and listening for a socket connection request from said controller.

See below for a diagram of what happens when a request is made to the device.

[9] https://en.wikipedia.org/wiki/NodeMCU
[10] https://www.arduino.cc/en/Reference/WiFi
[11] https://www.raspberrypi.org/products/raspberry-pi-zero-w/
[12] http://micropython.org/



Device Interface Language

The way the commands which are sent to the device are structured are represented here in BNF and ReGeX:

```
<authentication_key> ::= [A-z0-9]{32}
<command_code> ::= [a-z]{3}
```

```
<command_to_send> ::= <authentication_key> "," <command_code>
```

The `<command_to_send>` is simply encoded in UTF-8 and sent via sockets to the device.

Generic Commands

All the devices can execute these commands.

Commands	**Code**	**Returns**	**What does it do?**
enquire	enq	• code	Asks for device to send an acknowledgement back.

Device Specific Commands

These commands are specific to certain devices.

Commands - Temperature Sensor	**Code**	**Returns**	**What does it do?**
get_temprature	gtm	• code • temperature	Temperature is in celcius to 3 DP.

Commands - Switch	**Code**	**Returns**	**What does it do?**
get_current_state	gcs	• code • switch_state	Switch state 1 for closed 0 for open.
toggle	tgl	• code	
turn_off	tof	• code	
turn_on	ton	• code	

Device Response

Much like the API, the device will return a JSON Object; it will always (unless a fatal error occurs) contain the “code” parameter, stating if the request was executed correctly.

Before appending the device response JSON to the API response JSON, the .code attribute is removed, as the API client need not see this, only an API status.

If this leaves the JSON object empty, it is not appended.

Device Response Codes

Code	**Meaning**
0	Okay
1	Invalid Authentication Key
2	Request Incorrectly Formatted
3	Command is not valid for this device
4	Something went wrong with the device

Device File Structure

All devices have 4 base files I have written on them:

- configuration.py
- utilities.py
- main.py
- device.py

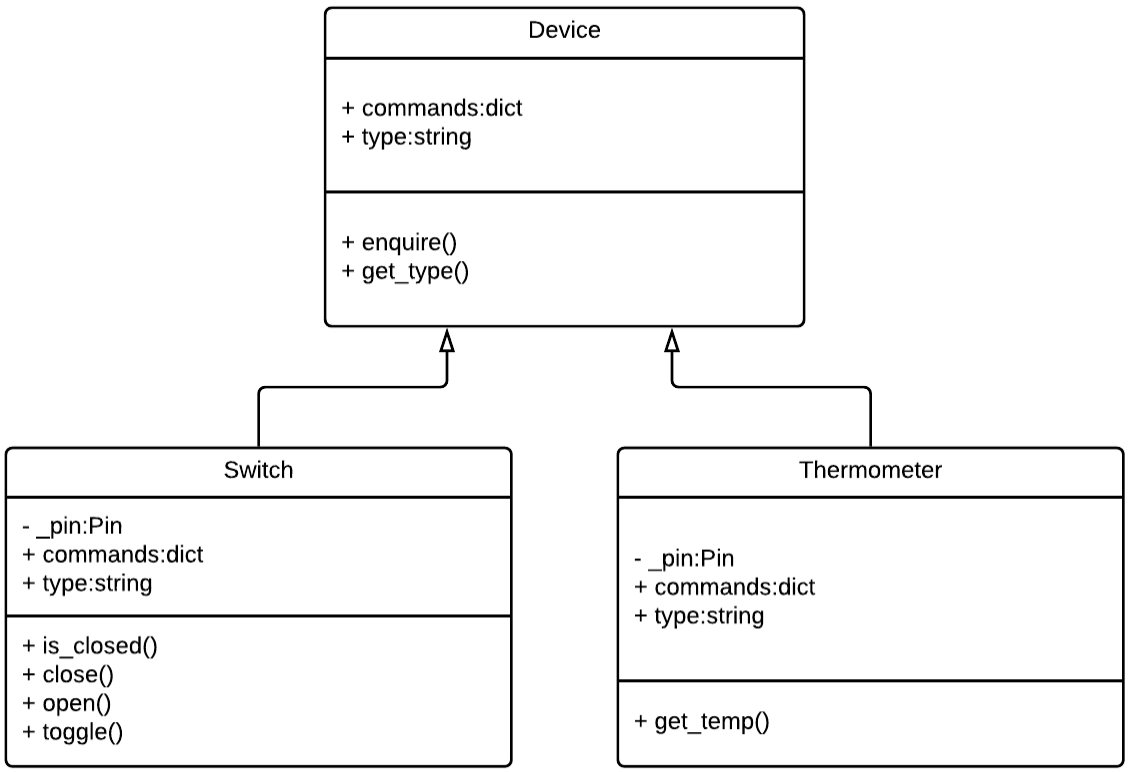
Main is called on device startup and imports utilities, a file with useful functions for the devices such as generating a random number or generating authentication keys.

Configuration is also imported at the start of main and will be explained in the later section “Adding a device to the system”.

Device Classes

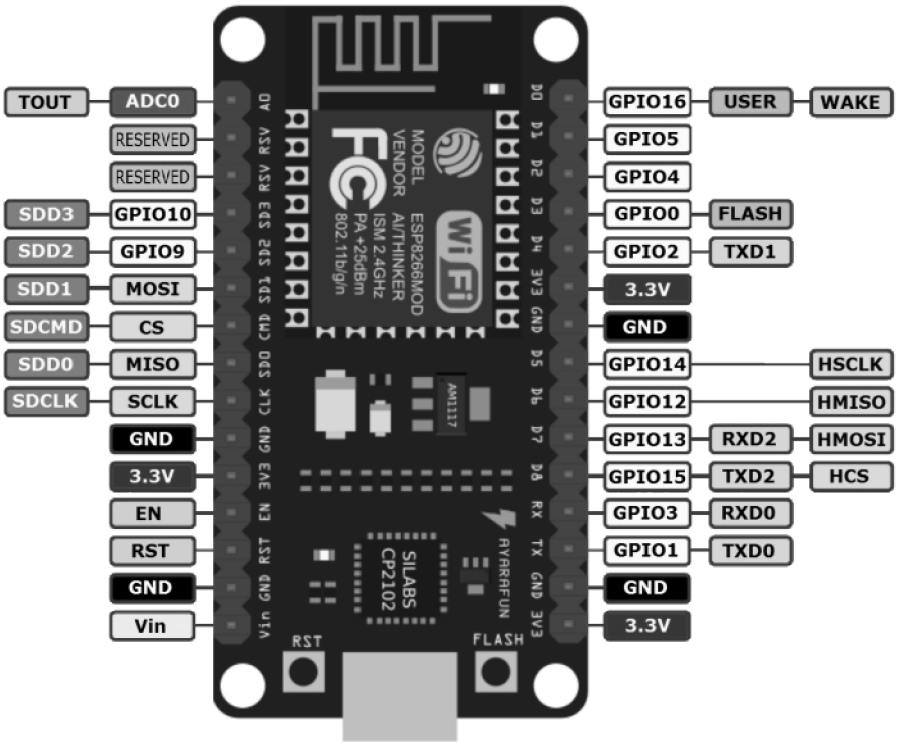
Device is imported by a device file (switch.py or thermometer.py) and is used as the parent class from which Switch and Thermometer inherit. It contains all of the functions which both devices share.

See below class definition diagrams:



Private attributes are not actually private due to python however they employ the python standard of an underscore before the variable name signifying that they should be used as such.

The Pin class is imported from the micropython library and allows control of the GPIO pins of the ESP8266; a pin is selected for the thermometer or switch to use upon instantiation of the respective objects. See below for a diagram of the GPIO pins of the device.



13

The attribute `commands` is a dictionary and maps the 3 character command codes to functions of the class; the new methods are added to this in each subclass. This is so when a command sent to the device a command such as this can be run:

```
switch.command["enq"]()
```

Such a command would return what needs to be sent back via sockets to the controller to let it know what has occurred.

I think that this is quite an elegant way of implementing this system as it reduces the amount of special cases needed E.G:

```
switch.commands[command]()

# Instead of

if command == "enq":
      switch.enqire()
```

[13] Sourced from google image search; no original owner could be found.

```
elif command == "tgl":
      switch.toggle()

# etc…
```

Adding a device to the System

Adding a device with no physical interface to a WiFi network is difficult: my first thought was to try and utilise the WiFi Protected Setup (WPS) button on a combined device to simply allow a device to connect without a password, however the ESP8266 with micropython does not support this yet.

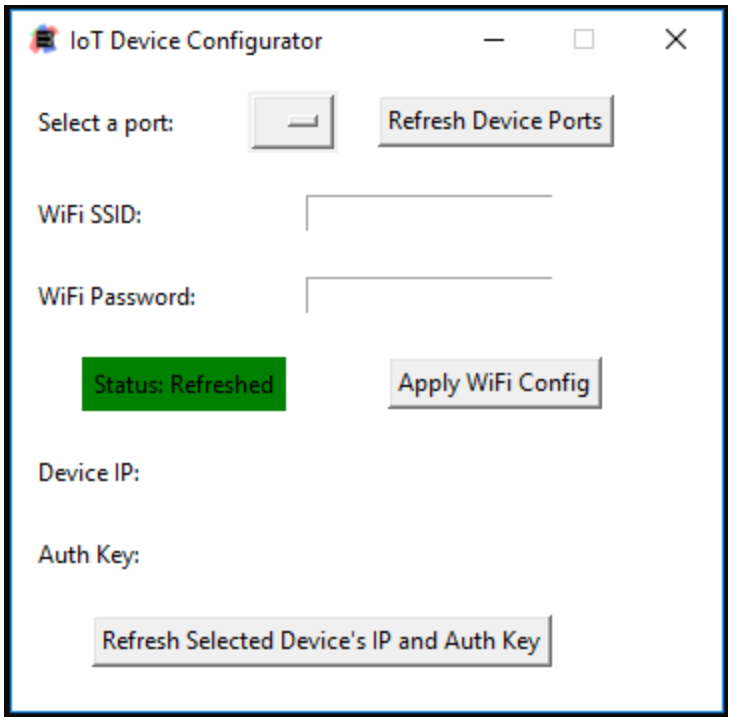
Therefore I had to come up with a different method.

Python's pyserial library allows communication over the serial ports of a PC and the ESP8266 has a serial interface with a micro usb cable; anything that is sent into that serial port of the ESP is put into a python command line. Using these features I could come up with a set of python functions which allow - if given a port name for a device that is plugged in via micro usb on a windows machine - set the WiFi network a device will be connected to.

I also wrote some functions to pull the authentication key and local ip of the device which will be use in the later stages of adding a device.

As mentioned in "Device File Structure", configuration is imported in main.py. It imports the functions which return the respective values when queried by my serial interface program.

To make this more user friendly, I wrote a tkinter GUI for the serial interface program such that one of my end users will be able to set up a device.



The configurator's GUI.

Once the device has been connected to the network and the IP and authentication key have been obtained, the next stage can begin.

The user would then call the command `add_device` of the API with the parameters of the auth key we have just retrieved from the device, the IP of the device we want to add and the session key of the user who we are adding it for.

The controller would then attempt to talk to the device via sockets on the IP provided and will validate that the authentication key that has been supplied is correct. If this works, the device is added to the database. Otherwise an appropriate error message is returned.

Device Security

The authentication key is needed to ensure only the controller server can interact with the devices; if it did not exist, anyone could send a socket message to a device asking it do something.

However, as we take the code from the device and input it into the server by hand we can be sure that only the server and device know the code and hence if a command is called with a valid authentication key, we know it is the server.

Technical Solution

Code Standardisation

For all the programs I have written in python, I have tried to adhere to the PEP-8 Python style guide[14] and hence the variable names are `written_like_this`, using underscores.

For all Javascript I have used camelCase variable names and bracketing like so:

```
function helloWorld(){
    alert("Hello World");
}
```

For SQL, I have used camelCase column names and for all API request related things E.G. parameters I have used underscores.

File Structure

List of Files and Locations

- Controller
    - controller_api.py
    - error_messages.json
    - password_hashing.py
    - utilities.py
    - run.sh

- Website
    - js
        - add_device.js

[14] https://www.python.org/dev/peps/pep-0008/

        - create_user.js
        - device.js
        - list_devices.js
        - login.js
        - main.js

    - add_device.html
    - create_user.html
    - device.html
    - list_devices.html
    - smart.css
    - logo.png
    - favicon.png

- Device
    - main.py
    - device.py
    - switch.py/ thermometer.py
    - configuration.py
    - utilities.py

Information about program files

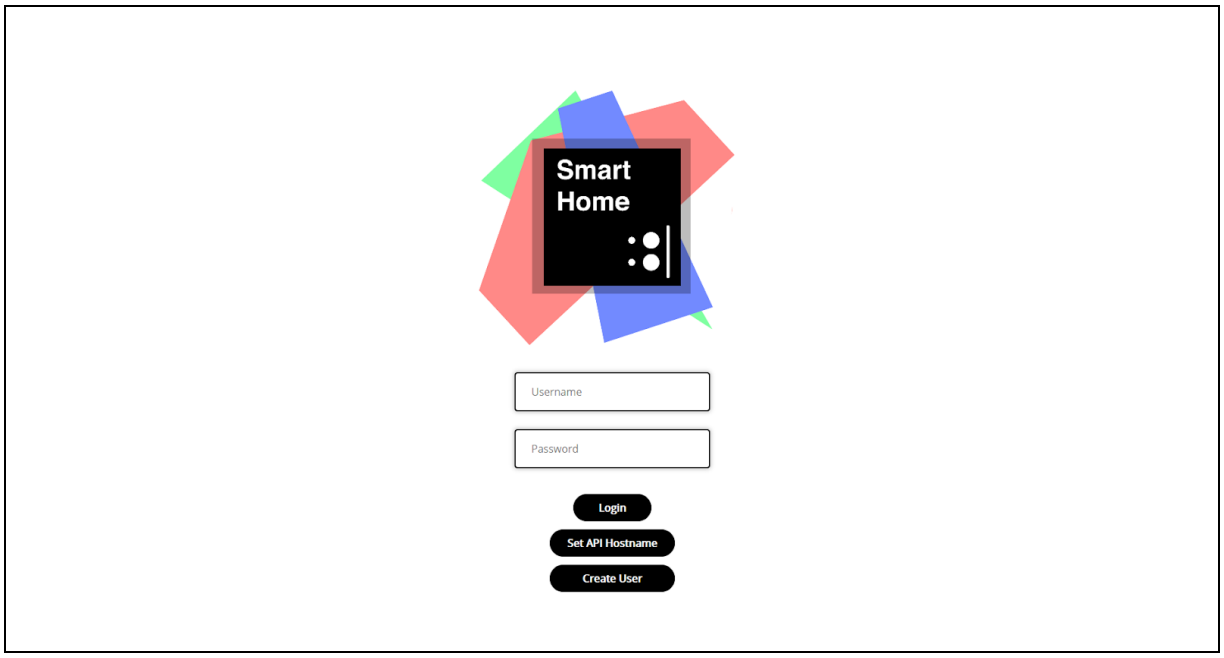
File	Explanation	Features to look out for
controller_api.py	Hold flask functions for interfacing with devices and user system.	• Advanced SQL Functions. • Communication over a network. • Advanced programming techniques (higher order functions). • Defensive Programming. • List operations
error_messages.json	Holds a json object mapping an error code to an error messages, read by other files.	
password_hashing.py	Holds the code for generating salts and	

	hashing passwords.	
utilities.py	Holds useful classes and functions which are called throughout controller_api. Examples include the device connection object.	• OOP. • Client-Server client connection object for device communication. • Database communication object.
run.sh	Simply bash script file to start the web server and the flask controller. Simply run `./run.sh` to start the server and website.	
add_device.js	Contains functions able to pull data from the add_device page and send it to the API to process the adding user request.	• OOP • Calling APIs
device.js	Contains functions relating to querying devices and permissions.	• OOP • Calling APIs
list_devices.js	Contains functions for showing which devices are on the system currently.	• OOP • Calling APIs
login.js	Contains functions able to pull data from the login page and send it to the API to process the login.	• OOP • Calling APIs
main.js	Called on every page; contains the ApiRequest class and various other useful	• ApiConnection class definition • List and string operations

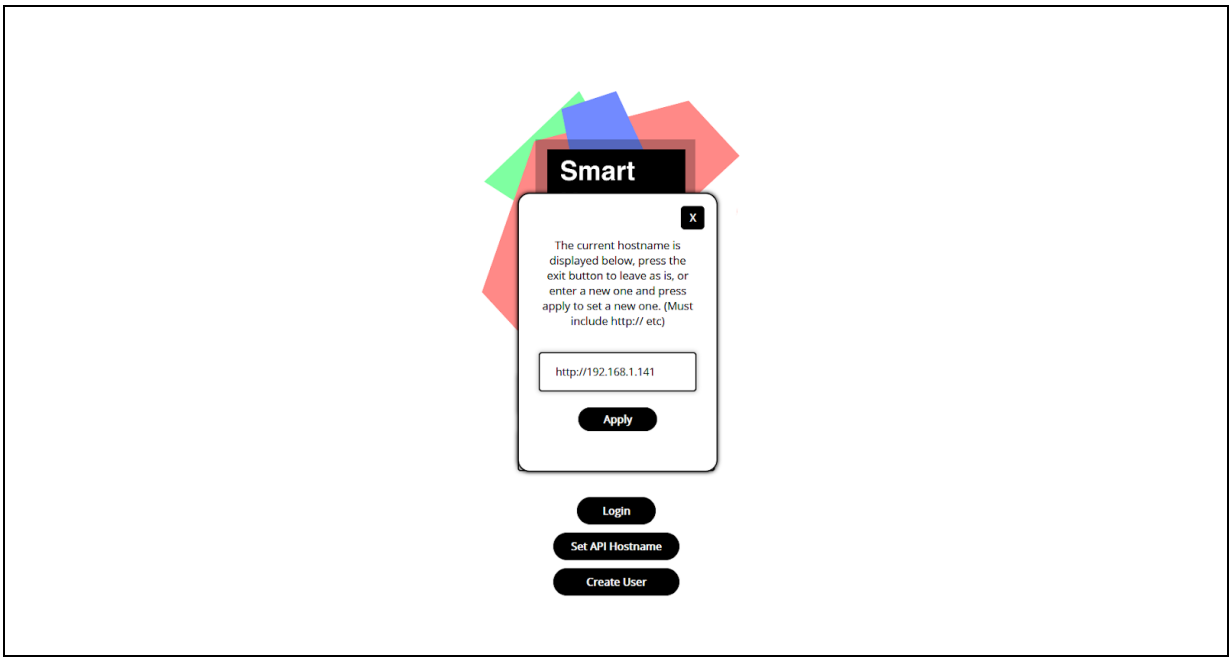
	functions such as getCookie().	
main.py	Called on startup of device, runs the socket listening loop for device command calling requests.	• Low level network communication.
device.py	Base class for other devices.	
switch.py/ thermometer.py	Class containing functions which the respective device can execute, inherits from Device.	• OOP - Inheritance
configuration.py	Contains functions which are all imported and are to be called by the serial WiFi configurator program.	
utilities.py	Contains useful function such as generating a random number, reading the contents of a file or logging a status message to log file.	
device_configurator.py	Base functions for communicating and configuring a device's WiFi connection via serial.	
device_configurator.py	GUI wrapper for the device_configurator.pu functions.	Threading used to allow tkinter to not crash whilst serial is waiting to see if a device was connected to a network successfully or running any long function.

GUI Screenshots

Login Page

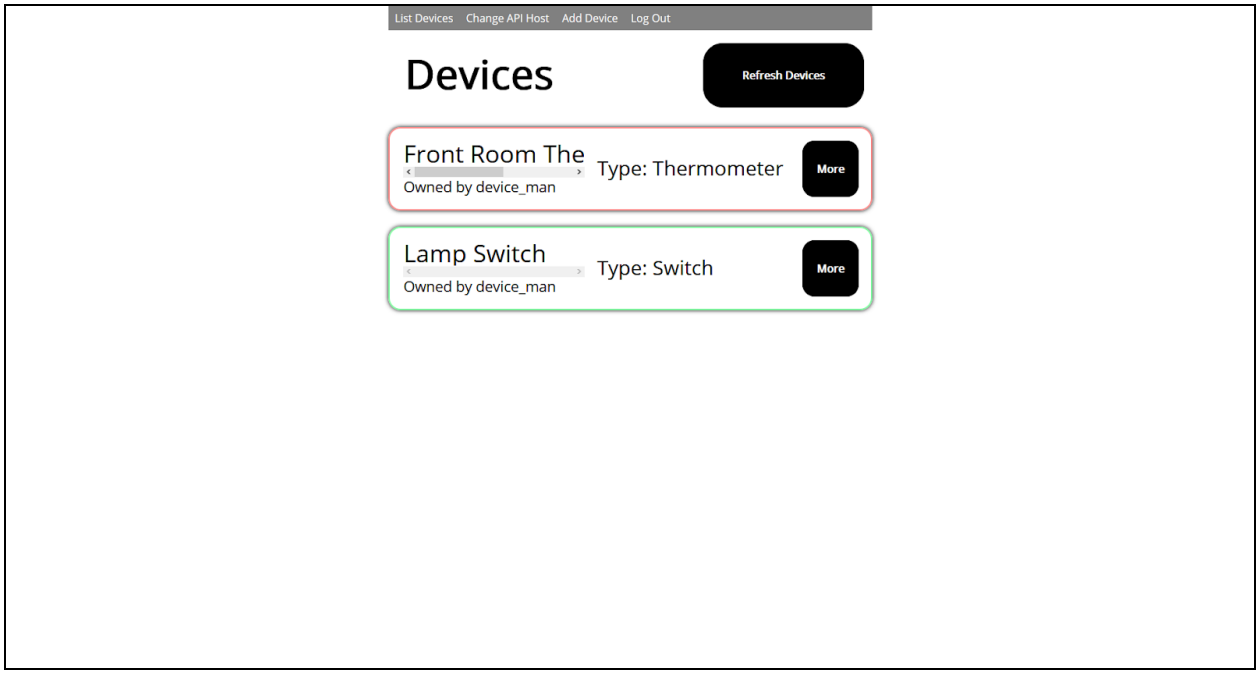


Change API Hostname Pop Up

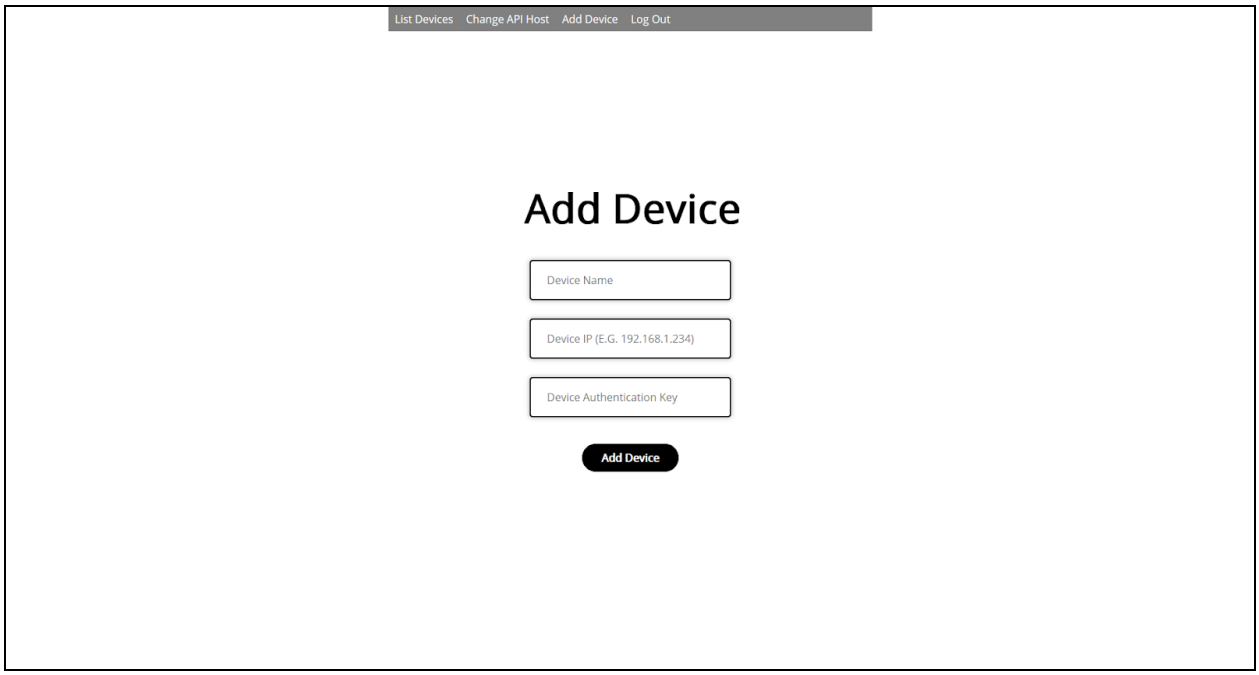


Create User Page

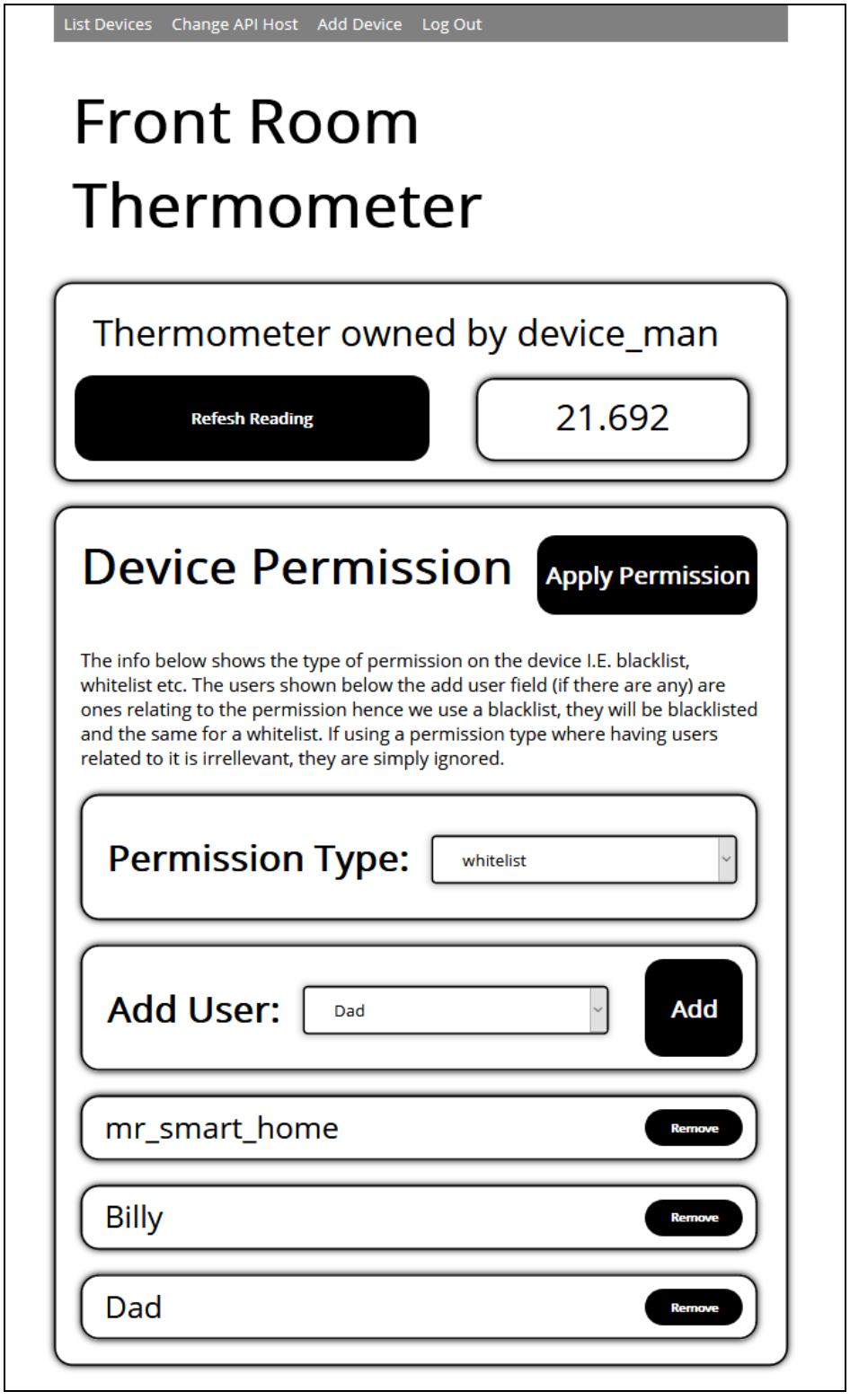
List Devices Page



Add Device Page



Detailed Device Page



Full Codebase

Controller Files

controller_api.py

```
from flask import Flask, request
from flask_cors import CORS, cross_origin
import datetime
import re

from utilities import *
import password_hashing as pw

VERBOSE = True
DEVICE_PORT = 24601

app = Flask(__name__)
CORS(app) # Needed to ensure API can be called from anywhere.

@app.route("/create_user", methods=["POST"])
def create_user():
    if VERBOSE:
        print_posted_values(request)

    # Dict for info about what parameters are meant to be supplied to this function
(apart from session_key)

    PARAMETERS = {
        "new_username": {
            "RegEx":"^[\w\d]+$",
            "required": True},
        "new_password": {
            "RegEx": "^(?=.*[^\w\d]+.*)(?=.*[A-z]+.*)(?=.*[\d]+.*).{8,}$",
            "required": True},
        "email": {
            "RegEx": "^[^\s]+@[^\s]+\.[^\s]+$",
            "required": False}
        }

    # First checks which parameters the user has supplied with intersection of 2
```

```
sets.

    parameters_included = set(PARAMETERS.keys()) & set(request.json.keys())

    # Next checks if the required parameters are in parameters_include with
difference of sets, gives items in a but not b.

    set_of_required_parameters = set(filter(lambda param:
PARAMETERS[param]["required"], PARAMETERS.keys()))
    required_parameters_not_included = set_of_required_parameters -
parameters_included

    if required_parameters_not_included:
        api_response = JSONAPIResponse()
        api_response.set_status(4, "Parameters missing: " + ",
".join(required_parameters_not_included))

        return api_response.build()

    # Checks if the parameters the user has included are valid

    parameter_is_valid = lambda parameter: re.match(PARAMETERS[parameter]["RegEx"],
request.json[parameter])
    invalid_parameters = list(filter(lambda param: not parameter_is_valid(param),
parameters_included))

    if invalid_parameters:
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Parameters invalid: " + ",
".join(invalid_parameters))

        return api_response.build()

    # Checking if the username is not already in use.

    with DBConnection() as conn:
        query = "SELECT username FROM user;"
        username_records = conn.call(query)

    # Values come out in like so: ((username, ), ) so need to pull them out of the
tuples

    usernames = [username_record[0] for username_record in username_records]

    if value_is_in(request.json["new_username"], usernames):
        api_response = JSONAPIResponse()
```

```
        api_response.set_status(5, "Usename already exists")

        return api_response.build()

    with DBConnection() as conn:
        # Inserts the new user

        query = "INSERT INTO user(username) VALUES('{}');"
        query = query.format(request.json["new_username"])

        conn.call(query)

        user_id = conn.last_insert_id()

        # Generates a salt for the user's password, adds it to the end, hashes that
hen stores it in the DB.

        password_salt = pw.generate_salt()
        salted_password_hash = pw.hash_string((request.json["new_password"] +
password_salt))

        password_query = "INSERT INTO password(passwordHash, salt, userID) VALUES
('{}', '{}', {});"
        password_query = password_query.format(salted_password_hash, password_salt,
str(user_id))

        conn.call(password_query)

        # Adds the extra parameters (if supplied)

        generic_query = "UPDATE user SET {SQL_column_name}='{value}' WHERE
userID={sql_user_id};"

        if value_is_in("email", parameters_included):
            query = generic_query.format(
                SQL_column_name="emailAddress",
                value=request.json["email"],
                sql_user_id=str(user_id)
                )

            conn.call(query)

    api_response = JSONAPIResponse()
    return api_response.build()

@app.route("/login", methods=["POST"])
```

```
def login():
    if VERBOSE:
        print_posted_values(request)

    # Dict to store information about the parameters to be supplied

    PARAMETERS = {
        "username": {"RegEx":"^[\w\d]+$"},
        "password": {"RegEx": "^.{8,}$"}
        }

    # Checks if the required parameters are in included with difference of sets,
gives items in a but not b

    parameters_not_included = set(PARAMETERS.keys()) - set(request.json.keys())

    if parameters_not_included:
        api_response = JSONAPIResponse()
        api_response.set_status(4, "Parameters missing: " + ",
".join(parameters_not_included))

        return api_response.build()

    # Checks if the parameters the user has included are valid

    parameter_is_valid = lambda parameter: re.match(PARAMETERS[parameter]["RegEx"],
request.json[parameter])
    invalid_parameters = list(filter(lambda param: not parameter_is_valid(param),
PARAMETERS.keys()))

    if invalid_parameters:
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Parameters invalid: " + ",
".join(invalid_parameters))

        return api_response.build()

    # Checks if the user's password is the same as the salted and hashed one

    with DBConnection() as conn:
        query = "SELECT password.passwordHash, password.salt, user.userID FROM
password INNER JOIN user ON password.userID = user.userID WHERE user.username =
'{}';"
        query = query.format(request.json["username"])
        records = conn.call(query)
```

```
    if not records:
        api_response = JSONAPIResponse()
        api_response.set_status(8)

        return api_response.build()

    password_hash, salt, user_id = records[0][0], records[0][1], records[0][2]

    if not pw.hash_string(request.json["password"] + salt) == password_hash:
        api_response = JSONAPIResponse()
        api_response.set_status(7)

        return api_response.build()

    # Generates a session key for this session and stores it in the DB

    session_key = generate_session_key()

    current_date_and_time = datetime.datetime.today().strftime("%Y-%m-%d %H:%M:%S")

    with DBConnection() as conn:
        query = """INSERT INTO session(userID, sessionKey, sessionCreationTime,
lastRequestTime)
        VALUES ({user_id}, '{session_key}', '{session_creation_time}',
'{last_request_time}');"""

        query = query.format(
            user_id=user_id,
            session_key=session_key,
            session_creation_time=current_date_and_time,
            last_request_time=current_date_and_time)

        conn.call(query)

    api_response = JSONAPIResponse()
    api_response.add_content(session_key, "session_key")

    return api_response.build()

@app.route("/is_session_key_valid", methods=["POST"])
def check_if_session_key_valid():
    if VERBOSE:
        print_posted_values(request)

    # Checks if session key is present and valid
```

```
    if "session_key" not in request.json.keys():
        api_response = JSONAPIResponse()
        api_response.set_status(4, "Missing session_key")
        return api_response.build()

    api_response = JSONAPIResponse()
    is_valid = is_session_key_valid(request.json["session_key"])

    update_session_last_request_time(request.json["session_key"])

    if is_valid:
        is_valid = "true"
    else:
        is_valid = "false"

    api_response.add_content(is_valid, "is_valid")
    return api_response.build()

@app.route("/log_out", methods=["POST"])
def log_out():
    if VERBOSE:
        print_posted_values(request)

    # Checks if session key is present and valid

    if not value_is_in("session_key", request.json.keys()):
            api_response = JSONAPIResponse()
            api_response.set_status(4, "Session key Missing.")
            return api_response.build()

    if not is_session_key_valid(request.json["session_key"]):
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Session key not valid (not active or simply not
a sessionKey). Your session may have expired, try logging in again.")
        return api_response.build()

    # Removes key from DB

    with DBConnection() as conn:
        query = "DELETE FROM session WHERE sessionKey='{}';"
        query = query.format(request.json["session_key"])
        conn.call(query)

    api_response = JSONAPIResponse()
    return api_response.build()
```

```
@app.route("/list_devices", methods=["POST"])
def list_devices():
    if VERBOSE:
        print_posted_values(request)

    # Checks if session key is present and valid

    if not value_is_in("session_key", request.json.keys()):
            api_response = JSONAPIResponse()
            api_response.set_status(4, "Session key Missing.")
            return api_response.build()

    if not is_session_key_valid(request.json["session_key"]):
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Session key not valid (not active or simply not
a sessionKey)")
        return api_response.build()

    # Update the last request on the session key time

    update_session_last_request_time(request.json["session_key"])

    # Fetches the data from the database

    with DBConnection() as conn:
        query = """
        SELECT device.deviceID, device.name, user.username, deviceType.name
        FROM device
        INNER JOIN user on device.ownerID = user.userID
        INNER JOIN deviceType on device.typeID = deviceType.typeID
        """;

        device_records = conn.call(query)

    device_records_dict = [{"id": record[0], "name": record[1], "owner": record[2],
"type": record[3]} for record in device_records]

    api_response = JSONAPIResponse()
    api_response.add_content(device_records_dict, "devices")

    return api_response.build()

@app.route("/get_device_info", methods=["POST"])
def get_device_info():
    if VERBOSE:
        print_posted_values(request)
```

```
    # Dict for info about what parameters are meant to be supplied to this function
(apart from session_key)

    PARAMETERS = {
        "device_id": {"RegEx":"^[0-9]+$"}
        }

    # Checks if session key is present and valid

    if not value_is_in("session_key", request.json.keys()):
            api_response = JSONAPIResponse()
            api_response.set_status(4, "Session key Missing.")
            return api_response.build()

    if not is_session_key_valid(request.json["session_key"]):
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Session key not valid (not active or simply not
a sessionKey)")
        return api_response.build()

    # Update the last request on the session key time

    update_session_last_request_time(request.json["session_key"])


    # Checks if the required parameters are in included with difference of sets,
gives items in a but not b

    parameters_not_included = set(PARAMETERS.keys()) - set(request.json.keys())

    if parameters_not_included:
        api_response = JSONAPIResponse()
        api_response.set_status(4, "Parameters missing: " + ",
".join(parameters_not_included))

        return api_response.build()

    # Checks if the parameters the user has included are valid

    parameter_is_valid = lambda parameter: re.match(PARAMETERS[parameter]["RegEx"],
request.json[parameter])
    invalid_parameters = list(filter(lambda param: not parameter_is_valid(param),
PARAMETERS.keys()))

    if invalid_parameters:
```

```
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Parameters invalid: " + ",
".join(invalid_parameters))

        return api_response.build()

    # Check the user has permission to do this action

    if not session_has_permission_for_device(request.json["session_key"],
request.json["device_id"]):
        api_response = JSONAPIResponse()
        api_response.set_status(9)
        return api_response.build()

    # Pulls the details from the DB

    with DBConnection() as conn:
        query = """
        SELECT device.deviceID, device.name, user.username, deviceType.name
        FROM device
        INNER JOIN user on device.ownerID = user.userID
        INNER JOIN deviceType on device.typeID = deviceType.typeID
        WHERE device.deviceID = {};
        """;

        query = query.format(request.json["device_id"])

        device_records = conn.call(query)

    if not device_records:
        api_response = JSONAPIResponse()
        api_response.set_status(10)
        return api_response.build()

    record = device_records[0]

    device_dict = {"id": record[0], "name": record[1], "owner": record[2], "type":
record[3]}

    api_response = JSONAPIResponse()
    api_response.add_content(device_dict, "device")

    return api_response.build()

@app.route("/add_device", methods=["POST"])
def add_device():
```

```
    if VERBOSE:
        print_posted_values(request)

    # Dict for info about what parameters are meant to be supplied to this function
(apart from session_key)

    PARAMETERS = {
        "device_ip": {"RegEx":"^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$"},
        "auth_key": {"RegEx": "^[A-z0-9]{32}$"},
        "device_name": {"RegEx": "^(\w| )+$"}
        }

    # Checks if session key is present and valid

    if not value_is_in("session_key", request.json.keys()):
            api_response = JSONAPIResponse()
            api_response.set_status(4, "Session key Missing.")
            return api_response.build()

    if not is_session_key_valid(request.json["session_key"]):
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Session key not valid (not active or simply not
a sessionKey)")
        return api_response.build()

    # Update the last request on the session key time

    update_session_last_request_time(request.json["session_key"])

    # Checks if the required parameters are in included with difference of sets,
gives items in a but not b

    parameters_not_included = set(PARAMETERS.keys()) - set(request.json.keys())

    if parameters_not_included:
        api_response = JSONAPIResponse()
        api_response.set_status(4, "Parameters missing: " + ",
".join(parameters_not_included))

        return api_response.build()

    # Checks if the parameters the user has included are valid

    parameter_is_valid = lambda parameter: re.match(PARAMETERS[parameter]["RegEx"],
request.json[parameter])
    invalid_parameters = list(filter(lambda param: not parameter_is_valid(param),
```

```
PARAMETERS.keys()))

    if invalid_parameters:
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Parameters invalid: " + ",
".join(invalid_parameters))

        return api_response.build()

    # Adding the device

    # Try to connect to the device, if not, return an error.
    try:
        conn =  DeviceConnection((request.json["device_ip"], DEVICE_PORT),
request.json["auth_key"])

    except Exception as error:
        if VERBOSE:
            print("Error:")
            print(error)

        api_response = JSONAPIResponse()
        api_response.set_status(11, "Could not talk to the device, perhaps the IP
is wrong.")
        return api_response.build()

    response = conn.call_command("enq")
    conn.close()

    if not response:
        # If this happens, the command call has timed out.

        api_response = JSONAPIResponse()
        api_response.set_status(11)
        return api_response.build()

    elif response["code"] == 1:
        # Invalid auth_key

        api_response = JSONAPIResponse()
        api_response.set_status(14)
        return api_response.build()

    elif response["code"] != 0: # Any other error
        if VERBOSE:
            print("Problem with device, code: " + str(response["code"]))
```

```
        api_response = JSONAPIResponse()
        api_response.set_status(13)
        return api_response.build()

    # Requesting the type from the device.

    try:
        conn =  DeviceConnection((request.json["device_ip"], DEVICE_PORT),
request.json["auth_key"])

    except Exception as error:
        if VERBOSE:
            print("Error:")
            print(error)

        api_response = JSONAPIResponse()
        api_response.set_status(11)
        return api_response.build()

    response = conn.call_command("gtp")
    conn.close()

    if not response:
        # If this happens, the command call has timed out.

        api_response = JSONAPIResponse()
        api_response.set_status(12)
        return api_response.build()

    elif response["code"] != 0: # Any other error
        if VERBOSE:
            print("Problem with device, code: " + str(response["code"]))

        api_response = JSONAPIResponse()
        api_response.set_status(13)
        return api_response.build()

    # Putting the new device in the DB

    with DBConnection() as conn:
        query = """
        INSERT INTO device (ownerID, typeID, IP, name, authenticationKey)
        VALUES
        (
        (SELECT userID FROM session WHERE sessionKey = '{session_key}'),
```

```
        (SELECT typeID FROM deviceType WHERE name = '{device_type}'),
        '{device_ip}',
        '{device_name}',
        '{auth_key}'
        );
        """

        query = query.format(
            session_key = request.json["session_key"],
            device_type = response["type"],
            device_ip = request.json["device_ip"],
            device_name = request.json["device_name"],
            auth_key = request.json["auth_key"]
        )

        conn.call(query)

        query = """
        INSERT INTO permission (permissionTypeID, deviceID)
        VALUES
        (
        (SELECT typeID FROM permissionType WHERE name = 'disallow_all'),
        {}
        );
        """

        query = query.format(conn.last_insert_id())

        conn.call(query)

    api_response = JSONAPIResponse()
    return api_response.build()

@app.route("/call_command_on_device", methods=["POST"])
def call_command_on_device():
    if VERBOSE:
        print_posted_values(request)

    # Dict for info about what parameters are meant to be supplied to this function
(apart from session_key)

    PARAMETERS = {
        "device_id": {"RegEx":"^[0-9]+$"},
        "command_code": {"RegEx": "^[a-z]{3}$"}
        }
```

```
    # Checks if session key is present and valid

    if not value_is_in("session_key", request.json.keys()):
            api_response = JSONAPIResponse()
            api_response.set_status(4, "Session key Missing.")
            return api_response.build()

    if not is_session_key_valid(request.json["session_key"]):
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Session key not valid (not active or simply not
a sessionKey)")
        return api_response.build()

    # Update the last request on the session key time

    update_session_last_request_time(request.json["session_key"])

    # Checks if the required parameters are in included with difference of sets,
gives items in a but not b

    parameters_not_included = set(PARAMETERS.keys()) - set(request.json.keys())

    if parameters_not_included:
        api_response = JSONAPIResponse()
        api_response.set_status(4, "Parameters missing: " + ",
".join(parameters_not_included))

        return api_response.build()

    # Checks if the parameters the user has included are valid

    parameter_is_valid = lambda parameter: re.match(PARAMETERS[parameter]["RegEx"],
request.json[parameter])
    invalid_parameters = list(filter(lambda param: not parameter_is_valid(param),
PARAMETERS.keys()))

    if invalid_parameters:
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Parameters invalid: " + ",
".join(invalid_parameters))

        return api_response.build()


    # Pulling the device's IP and auth_key from the DB
```

```
    with DBConnection() as conn:
        query = "SELECT IP, authenticationKey FROM device WHERE deviceID = {};"
        query = query.format(request.json["device_id"])
        records = conn.call(query)

    if not records: # DeviceID doesn't exist
        api_response = JSONAPIResponse()
        api_response.set_status(16)
        return api_response.build()

    ip, auth_key = records[0]

    # Checking if the command code that the user has provided can be executed by
said device

    with DBConnection() as conn:
        query = """
        SELECT command.commandCode FROM device
        INNER JOIN deviceType ON device.typeID = deviceType.typeID
        INNER JOIN deviceTypeHasCommand ON deviceType.typeID =
deviceTypeHasCommand.typeID
        INNER JOIN command ON deviceTypeHasCommand.commandID = command.commandID
        WHERE device.deviceID = {};
        """

        query = query.format(request.json["device_id"])
        records = conn.call(query)

    if not records:
        api_response = JSONAPIResponse()
        api_response.set_status(17)
        return api_response.build()

    # Check the user has permission to do this action on this device

    if not session_has_permission_for_device(request.json["session_key"],
request.json["device_id"]):
        api_response = JSONAPIResponse()
        api_response.set_status(9)
        return api_response.build()

    # Try to connect to the device using my DeviceConnection class, if not, return
an error

    if VERBOSE:
        print("Connecting to: ")
```

```
        print(ip, DEVICE_PORT, auth_key)

    try:
        conn =  DeviceConnection((ip, DEVICE_PORT), auth_key)

    except Exception as error:

        """
        Could be considered bad practise not to specify an error, however
        there are simply too many errors that could occur here, they user can check
        in the logs later if they wish to do so.
        """

        if VERBOSE:
            print("Error:")
            print(error)

        api_response = JSONAPIResponse()
        api_response.set_status(11)
        return api_response.build()

    # Connection has been successfully established, now we send a command

    response = conn.call_command(request.json["command_code"])

    conn.close()

    if not response:
        # If this happens, the command call has timed out

        api_response = JSONAPIResponse()
        api_response.set_status(12)
        return api_response.build()

    if VERBOSE:
        print("Recieved from device:")
        print(json.dumps(response))

    """
    Although the device responded correctly, doesn't mean it executed the
    command correctly, the code below returns appropriate responses for such a
    situation.

    Only returns specific error codes for things that someone calling the API could
have done wrong.
    The rest can be checked in the controller log.
```

```
    """

    if response["code"] != 0:
        if response["code"] == 3: # Invalid auth_key
            api_response = JSONAPIResponse()
            api_response.set_status(17)
            return api_response.build()

        else: # Any other problem that is irrelevent to an API user as if something
went wrong, it was not their fault
            api_response = JSONAPIResponse()
            api_response.set_status(13)
            return api_response.build()

    del response["code"] # The caller need not know the device response code, only
data it sends back (if any).

    api_response = JSONAPIResponse()

    if response:
        api_response.add_content(response, "device_response")

    return api_response.build()

@app.route("/get_device_permission_info", methods=["POST"])
def get_device_permission_info():
    if VERBOSE:
        print_posted_values(request)

    # Dict for info about what parameters are meant to be supplied to this function
(apart from session_key)

    PARAMETERS = {
        "device_id": {"RegEx":"^[0-9]+$"}
        }

    # Checks if session key is present and valid

    if not value_is_in("session_key", request.json.keys()):
            api_response = JSONAPIResponse()
            api_response.set_status(4, "Session key Missing.")
            return api_response.build()

    if not is_session_key_valid(request.json["session_key"]):
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Session key not valid (not active or simply not
```

```
a sessionKey)")
        return api_response.build()

    # Update the last request on the session key time

    update_session_last_request_time(request.json["session_key"])

    # Checks if the required parameters are in included with difference of sets,
gives items in a but not b

    parameters_not_included = set(PARAMETERS.keys()) - set(request.json.keys())

    if parameters_not_included:
        api_response = JSONAPIResponse()
        api_response.set_status(4, "Parameters missing: " + ",
".join(parameters_not_included))

        return api_response.build()

    # Checks if the parameters the user has included are valid

    parameter_is_valid = lambda parameter: re.match(PARAMETERS[parameter]["RegEx"],
request.json[parameter])
    invalid_parameters = list(filter(lambda param: not parameter_is_valid(param),
PARAMETERS.keys()))

    if invalid_parameters:
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Parameters invalid: " + ",
".join(invalid_parameters))

        return api_response.build()

    # Check the user has permission to do this action

    if not session_has_permission_for_device(request.json["session_key"],
request.json["device_id"]):
        api_response = JSONAPIResponse()
        api_response.set_status(9)
        return api_response.build()

    # Pulling the permission type

    with DBConnection() as conn:
        query = "SELECT permissionType.name FROM permission INNER JOIN
permissionType on permission.permissionTypeID = permissionType.typeID WHERE
```

```
permission.deviceID = {};"
        query = query.format(request.json["device_id"])
        records = conn.call(query)

    if not records: # DeviceID must not exist
        api_response = JSONAPIResponse()
        api_response.set_status(16)
        return api_response.build()

    perm_type = records[0][0]

    # Pulling the users related to the permission from the DB

    with DBConnection() as conn:
        query = """
        SELECT user.username FROM permission
        INNER JOIN userInPermission ON permission.permissionID =
userInPermission.permissionID
        INNER JOIN user on userInPermission.userID = user.userID
        INNER JOIN device ON permission.deviceID = device.deviceID
        WHERE permission.deviceID = {} AND NOT user.userID = device.ownerID;
        """

        query = query.format(request.json["device_id"])
        records = conn.call(query)

    usernames = [record[0] for record in records]

    permission = {
        "type": perm_type,
        "users": usernames
    }


    api_response = JSONAPIResponse()
    api_response.add_content(permission, "permission")
    return api_response.build()

@app.route("/get_permission_types", methods=["POST"])
def get_permission_types():
    # No parameters are sent here!

    with DBConnection() as conn:
        query = "SELECT name, description FROM permissionType;"
        records = conn.call(query)
```

```
    types = [{"name": record[0], "description": record[1]} for record in records]

    api_response = JSONAPIResponse()
    api_response.add_content(types, "types")
    return api_response.build()

@app.route("/get_users_list", methods=["POST"])
def get_users_list():
    # No parameters are sent here!

    with DBConnection() as conn:
        query = "SELECT username FROM user;"
        records = conn.call(query)

    users = [record[0] for record in records]

    api_response = JSONAPIResponse()
    api_response.add_content(users, "users")
    return api_response.build()

@app.route("/set_device_permission", methods=["POST"])
def set_device_permission():
    if VERBOSE:
        print_posted_values(request)

    # Dict for info about what parameters are meant to be supplied to this function
(apart from session_key)

    PARAMETERS = {
        "device_id": {
            "RegEx":"^[0-9]+$",
            "isList": False
            },

        "permission_type": {
            "RegEx": "^\w+$",
            "isList": False
            },

        "users": {
            "RegEx": "^\w+$",
            "isList": True
            }
        }

    # Checks if session key is present and valid
```

```
    if not value_is_in("session_key", request.json.keys()):
            api_response = JSONAPIResponse()
            api_response.set_status(4, "Session key Missing.")
            return api_response.build()

    if not is_session_key_valid(request.json["session_key"]):
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Session key not valid (not active or simply not
a sessionKey)")
        return api_response.build()

    # Update the last request on the session key time

    update_session_last_request_time(request.json["session_key"])

    # Checks if the required parameters are in included with difference of sets,
gives items in a but not b

    parameters_not_included = set(PARAMETERS.keys()) - set(request.json.keys())

    if parameters_not_included:
        api_response = JSONAPIResponse()
        api_response.set_status(4, "Parameters missing: " + ",
".join(parameters_not_included))

        return api_response.build()

    # Checks if the parameters the user has included are valid

    parameter_is_valid = lambda parameter: re.match(PARAMETERS[parameter]["RegEx"],
request.json[parameter])

    invalid_parameters = []

    for parameter in PARAMETERS.keys():
        if PARAMETERS[parameter]["isList"]:
            for value in request.json[parameter]:
                if not re.match(PARAMETERS[parameter]["RegEx"], value):
                    invalid_parameters.append(parameter)
                    break
        else:
            if not parameter_is_valid(parameter):
                invalid_parameters.append(parameter)
```

```
    if invalid_parameters:
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Parameters invalid: " + ",
".join(invalid_parameters))

        return api_response.build()

    # Check the user has permission to do this action

    if not session_has_permission_for_device(request.json["session_key"],
request.json["device_id"]):
        api_response = JSONAPIResponse()
        api_response.set_status(9)
        return api_response.build()

    # Pulling list of permission types

    with DBConnection() as conn:
        query = "SELECT permissionType.name FROM permissionType;"
        records = conn.call(query)

    types = [record[0] for record in records]

    if not value_is_in(request.json["permission_type"], types):
        api_response = JSONAPIResponse()
        api_response.set_status(6, "Not a valid permission type.")
        return api_response.build()

    # Check if all the users supplied actually exist

    with DBConnection() as conn:
        query = "SELECT username FROM user;"
        records = conn.call(query)

    users = [record[0] for record in records]

    for proposed_username in request.json["users"]:
        if not proposed_username in users:
            api_response = JSONAPIResponse()
            api_response.set_status(8)
            return api_response.build()

    # Removing the owner name if provided as no point having them in the permission

    with DBConnection() as conn:
        query = "SELECT user.username FROM session INNER JOIN user ON
```

```
session.userID = user.userID WHERE session.sessionKey = '{}'"
        query = query.format(request.json["session_key"])
        records = conn.call(query)

    owner = records[0][0]

    if owner in request.json["users"]:
        request.json["users"].remove(owner)

    # Putting the permission in the DB

    with DBConnection() as conn:
        # Set permission type

        query = """
        UPDATE permission
        SET permission.permissionTypeID = (SELECT typeID FROM permissionType WHERE
name = '{permission_type}')
        WHERE deviceID = {device_id};
        """

        query = query.format(
            permission_type = request.json["permission_type"],
            device_id = request.json["device_id"]
            )

        conn.call(query)

        # Remove all users related to permission

        query = "DELETE userInPermission FROM userInPermission INNER JOIN
permission ON userInPermission.permissionID = permission.permissionID WHERE
permission.deviceID = {};"
        query = query.format(request.json["device_id"])

        conn.call(query)

        # Add users user has sent

        query = """
        INSERT INTO userInPermission (userID, permissionID)
        VALUES
        ((SELECT userID FROM user WHERE username = '{username}'),
        (SELECT permissionID FROM permission WHERE deviceID = {device_id}));
        """
```

```
        for username in request.json["users"]:
            temp_query = query.format(
                username = username,
                device_id = request.json["device_id"]
                )

            conn.call(temp_query)

    api_response = JSONAPIResponse()
    return api_response.build()
```

utilities.py

```
import random
import pymysql
import string
from flask import Response
import json
import re
import datetime
import socket

DATABASE_DETAILS = {
    "host":"localhost",
    "user":"controllerUser",
    "passwd":"_b>/8KT4>-U/wKv$",
    "db":"controllerDB"}

ERROR_MESSAGE_FILE_NAME = "error_messages.json"
VERBOSE = True

# Classes


class DBConnection:

    """
    Class to be used in conjunction with a 'with' statement as to
    ensure correct usage; connection is closed and opened correctly.
    """

    def call(self, command):
        cursor = self._db.cursor()
        cursor.execute(command)
```

```
        to_return = cursor.fetchall()
        cursor.close()

        return to_return

    def last_insert_id(self):
        return self._db.insert_id()

    def __enter__(self, details=DATABASE_DETAILS):
        self._db = db = pymysql.connect(**DATABASE_DETAILS)
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):  # Passed automatically by
python.
        self._db.commit()
        self._db.close()


class JSONAPIResponse:
    def __init__(self):
        self._content = {}
        self.set_status(1)

    def set_status(self, code, text=""):
        if not text:
            text = get_error_message(code)

        self._content["status"] = {"code": code, "text": text}

    def add_content(self, content, content_id):
        self._content[content_id] = content

    def build(self):
        response = Response(json.dumps(self._content))
        response.headers["Access-Control-Allow-Origin"] = '*'     # Allows API to
be called from any origin.
        response.headers["Content-type"] = "application/json"   # Tells recieving
machine that it's getting json.

        return response

class DeviceConnection:
    def __init__(self, host, auth_key, timeout=3.5):
        self._host = host
        self._auth_key = auth_key
```

```
        self._sk = socket.socket()
        self._sk.settimeout(timeout)

        """
        The line below will intentionally throw an error if it can't connect
        to a device, this will be dealt with in controller.py.
        """

        self._sk.connect(host)

    def close(self):
        self._sk.close()

    def call_command(self, command):
        to_send = self._auth_key + "," + command

        if VERBOSE:
            print("Sending Command:")
            print(to_send)

        try:
            self._sk.send(to_send.encode())
            response = self._sk.recv(1024)

        except (socket.timeout, ConnectionResetError):
            return False

        response = response.decode()

        return json.loads(response)

# Functions

def print_posted_values(flask_response_object):
    print("Recieved post values: " + "".join(["\n" + str(key) + ": " +
str(flask_response_object.json[key]) for key in
flask_response_object.json.keys()]))


def generate_session_key(length=32):
    return "".join([random.choice(string.ascii_letters + string.digits) for i in
range(length)])

def is_session_key_valid(key): # not to be used to check if a session key is
expired
```

```
    KEY_REGEX = "^[A-z0-9]{32}$"

    if not re.match(KEY_REGEX, key):
        return False

    with DBConnection() as conn:
        query = "SELECT sessionKey FROM session WHERE sessionKey='{}';"
        query = query.format(key)
        records = conn.call(query)

    if not records:
        return False
    else:
        return True

def get_error_message(code):
    with open(ERROR_MESSAGE_FILE_NAME, "r") as f:
        data = f.read()

    codes = json.loads(data)

    if str(code) in codes.keys():
        return codes[str(code)]
    else:
        return ""

def update_session_last_request_time(session_key):
    with DBConnection() as conn:
        current_date_and_time = datetime.datetime.today().strftime("%Y-%m-%d
%H:%M:%S")
        query = "UPDATE session lastRequestTime='{}' WHERE sessionKey =
'{}';".format(current_date_and_time, session_key)


def session_has_permission_for_device(session_key, device_id):
    with DBConnection() as conn:
        query = "SELECT userID FROM session WHERE sessionKey = '{}'"
        user_id = conn.call(query.format(session_key))[0][0]

        query = "SELECT ownerID FROM device WHERE deviceID = {}"
        owner_id = conn.call(query.format(device_id))[0][0]

    if user_id == owner_id:
        if VERBOSE:
            print("Permission granted as caller is owner.")
        return True
```

```
    with DBConnection() as conn:
        query = "SELECT permissionType.name FROM permission INNER JOIN
permissionType ON permission.permissionTypeID = permissionType.typeID WHERE
permission.deviceID = {}"
        permission_type = conn.call(query.format(device_id))[0][0]

    if permission_type == "disallow_all":
        if VERBOSE:
            print("Permission disallowed as disallow_all is active.")
        return False

    elif permission_type == "allow_all":
        if VERBOSE:
            print("Permission granted as allow_all is active.")

        return True

    with DBConnection() as conn:
        query = "SELECT userInPermission.userID FROM userInPermission INNER JOIN
permission ON userInPermission.permissionID = permission.permissionID WHERE
permission.deviceID = {}"
        users_in_permission = conn.call(query.format(device_id))

    users_in_permission = [record[0] for record in users_in_permission]

    if value_is_in(user_id, users_in_permission):
        if permission_type == "blacklist":
            if VERBOSE:
                print("Permission disallowed as caller is in blacklist.")

            return False

        elif permission_type == "whitelist":
            if VERBOSE:
                print("Permission allowed as caller is in whitelist.")

            return True
    else:
        if permission_type == "blacklist":
            if VERBOSE:
                print("Permission allowed as caller is not in blacklist.")
            return True

        elif permission_type == "whitelist":
            if VERBOSE:
```

```
                print("Permission disallowed as caller is not in whitelist.")
            return False
```

password_hashing.py

```
import hashlib
import random
import string

def generate_salt(length=16):
    """
    random salt of 16 characters
    """

    return "".join([random.choice(string.ascii_letters + string.digits) for i in
range(length)])

def hash_string(string_to_hash):
    h = hashlib.sha256(string_to_hash.encode("utf-8"))
    return h.hexdigest()
```

error_messages.json

```
{
    "1": "Success.",
    "2": "API Use Error.",
    "3": "Server Error.",
    "4": "Missing Parameters.",
    "5": "Username already exists.",
    "6": "Invalid Parameters.",
    "7": "Incorrect password.",
    "8": "Username doesn't exist.",
    "9": "Insufficient Permissions.",
    "10": "deviceID not in use.",
    "12": "Request timed out",
    "11": "General problem connecting to device. Ensure the device is powered and
connected to the network.",
    "13": "General problem executing a command on a device.",
    "14": "Authentication Key incorrect.",
    "15": "Non successful device response.",
    "16": "deviceID doesn't exist.",
    "17": "Invalid command for this device."
```

```
}
```

run.sh

```
(cd Website;
python3 -m http.server 8000) &

(cd Controller;
export FLASK_APP=controller_api.py;
python3 -m flask run -h 0.0.0.0;)
```

Database

```
DROP DATABASE IF EXISTS controllerDB;
CREATE DATABASE controllerDB;
USE controllerDB;

CREATE TABLE user(
    userID INT UNSIGNED AUTO_INCREMENT NOT NULL UNIQUE,
    username VARCHAR(64) NOT NULL UNIQUE,
    emailAddress VARCHAR(256),
    PRIMARY KEY(userID)
);

CREATE TABLE password(
    passwordID INT UNSIGNED AUTO_INCREMENT NOT NULL,
    passwordHash VARCHAR(64) NOT NULL,
    salt varchar(16) NOT NULL,
    userID INT UNSIGNED NOT NULL UNIQUE,
    PRIMARY KEY (passwordID),
    CONSTRAINT `user_password_FK` FOREIGN KEY (userID) REFERENCES
user(userID) ON DELETE CASCADE
);

CREATE TABLE session(
    sessionID INT UNSIGNED AUTO_INCREMENT NOT NULL,
    userID INT UNSIGNED NOT NULL,
```

```
    sessionKey VARCHAR(32) NOT NULL,
    sessionCreationTime DATETIME NOT NULL,
    lastRequestTime DATETIME NOT NULL,
    PRIMARY KEY(sessionID),
    CONSTRAINT `user_session_FK` FOREIGN KEY (userID) REFERENCES
user(userID) ON DELETE CASCADE
);

CREATE TABLE deviceType(
    typeID INT UNSIGNED AUTO_INCREMENT NOT NULL,
    name VARCHAR(32),
    description VARCHAR(256),
    PRIMARY KEY(typeID)
);

CREATE TABLE device(
    deviceID INT UNSIGNED AUTO_INCREMENT NOT NULL,
    ownerID INT UNSIGNED NOT NULL,
    typeID INT UNSIGNED NOT NULL,
    IP VARCHAR(15), /* In form "xxx.xxx.xxx.xxx" */
    name VARCHAR(128),
    authenticationKey VARCHAR(32),
    PRIMARY KEY(deviceID),
    CONSTRAINT `device_owner_FK` FOREIGN KEY (ownerID) REFERENCES
user(userID)  ON DELETE CASCADE,
    CONSTRAINT `device_type_FK` FOREIGN KEY (typeID) REFERENCES
deviceType(typeID)  ON DELETE CASCADE
);

CREATE TABLE command(
    commandID INT UNSIGNED AUTO_INCREMENT NOT NULL,
    name VARCHAR(64),
    commandCode VARCHAR(3),
    description VARCHAR(512),
    PRIMARY KEY(commandID)
);

CREATE TABLE deviceTypeHasCommand(
    commandID INT UNSIGNED NOT NULL,
    typeID INT UNSIGNED NOT NULL,
    CONSTRAINT `linking_commandID_FK` FOREIGN KEY (commandID) REFERENCES
```

```
command(commandID),
    CONSTRAINT `linking_deviceType_FK` FOREIGN KEY (typeID) REFERENCES
deviceType(typeID),
    PRIMARY KEY(commandID, typeID)
);

CREATE TABLE permissionType(
    typeID INT UNSIGNED AUTO_INCREMENT NOT NULL,
    name VARCHAR(64),
    description VARCHAR(512),
    PRIMARY KEY(typeID)
);

CREATE TABLE permission(
    permissionID INT UNSIGNED AUTO_INCREMENT NOT NULL,
    permissionTypeID INT UNSIGNED NOT NULL,
    deviceID INT UNSIGNED NOT NULL UNIQUE,
    PRIMARY KEY(permissionID),
    CONSTRAINT `permission_type_FK` FOREIGN KEY (permissionTypeID)
REFERENCES permissionType(typeID)  ON DELETE CASCADE,
    CONSTRAINT `permission_device_FK` FOREIGN KEY (deviceID) REFERENCES
device(deviceID)  ON DELETE CASCADE
);

CREATE TABLE userInPermission(
    userInPermissionID INT UNSIGNED AUTO_INCREMENT NOT NULL,
    userID INT UNSIGNED NOT NULL,
    permissionID INT UNSIGNED NOT NULL,
    PRIMARY KEY(userInPermissionID),
    CONSTRAINT `user_in_permission_FK` FOREIGN KEY (userID) REFERENCES
user(userID)  ON DELETE CASCADE,
    CONSTRAINT `permission_FK` FOREIGN KEY (permissionID) REFERENCES
permission(permissionID)  ON DELETE CASCADE
);

INSERT INTO permissionType(name, description)
VALUES
('blacklist', 'Disallows all people on this list.'),
('whitelist', 'Allows only people on this list.'),
('disallow_all', 'Allows no one but owner.'),
('allow_all', 'Allows everyone.');
```

```
INSERT INTO deviceType(name, description)
VALUES
('Switch', 'A device which can be switched on and off via the control
system.'),
('Thermometer', 'A device for viewing the temperature of wherever the
device is placed.')
;

INSERT INTO command(name, commandCode, description)
VALUES
('Enquire', 'enq', 'Check to see if a device is still on the system.'),
('Validate Auth Key', 'val', 'Used upon adding a device to the system, asks
a device to send its auth key to verify it what we think it is.'),
('Get Temperature', 'gtm', 'Gets the temprerature of a thermometer.'),
('Get Current State', 'gcs', 'Get the current state of a switch device.'),
('Toggle', 'tgl', 'Toggles the state of a switch device.'),
('Turn off', 'tof', 'Turns off (opens) a switch device.'),
('Turn on', 'ton', 'Turn on (closes) a switch device'),
('Get Type', 'gtp', 'Returns the type of a device.')
;

INSERT INTO deviceTypeHasCommand(commandID, typeID)
VALUES
((SELECT commandID FROM command WHERE commandCode = 'enq'), (SELECT typeID
FROM deviceType WHERE name = 'Switch')),
((SELECT commandID FROM command WHERE commandCode = 'enq'), (SELECT typeID
FROM deviceType WHERE name = 'Thermometer')),
((SELECT commandID FROM command WHERE commandCode = 'val'), (SELECT typeID
FROM deviceType WHERE name = 'Switch')),
((SELECT commandID FROM command WHERE commandCode = 'val'), (SELECT typeID
FROM deviceType WHERE name = 'Thermometer')),
((SELECT commandID FROM command WHERE commandCode = 'gtm'), (SELECT typeID
FROM deviceType WHERE name = 'Thermometer')),
((SELECT commandID FROM command WHERE commandCode = 'tgl'), (SELECT typeID
FROM deviceType WHERE name = 'Switch')),
((SELECT commandID FROM command WHERE commandCode = 'tof'), (SELECT typeID
FROM deviceType WHERE name = 'Switch')),
((SELECT commandID FROM command WHERE commandCode = 'ton'), (SELECT typeID
FROM deviceType WHERE name = 'Switch')),
((SELECT commandID FROM command WHERE commandCode = 'gcs'), (SELECT typeID
```

```
FROM deviceType WHERE name = 'Switch')),
((SELECT commandID FROM command WHERE commandCode = 'gtp'), (SELECT typeID
FROM deviceType WHERE name = 'Switch')),
((SELECT commandID FROM command WHERE commandCode = 'gtp'), (SELECT typeID
FROM deviceType WHERE name = 'Thermometer'))
;
```

Device

main.py (for switch)

```
import os
from utilities import *
from configuration import *
import socket
import json
from thermometer import Thermometer

log("File imports sucessful!")

VERBOSE = True

HOST = "0.0.0.0"
DEVICE_PORT = 24601
ADC_PIN = 0

generate_auth_key_if_not_exists()

"""
Unfortunately micropython on the ESP8266 does not support threading at this point
and so only 1 request can be processed at once, however this should be fine as
request collisions should not occur very often.
"""

thermometer = Thermometer(ADC_PIN)

"""
log() logs the string to a text file ("log.txt") which we can have a look at
incase anything goes wrong or we need to see what requests have been occuring
on the device.
"""
```

```
log("Thermometer object instantiated!")

# Connection Manager

sk = socket.socket()
sk.bind((HOST, DEVICE_PORT))
sk.listen(1)

while True:
    log("Listening...")
    conn, addr = sk.accept()

    log("Request from " + addr[0])

    data = conn.recv(2048).decode()

    log("They sent " + data)

    command = data.split(",")

    if len(command) != 2:
        conn.send(json.dumps({"code": 2}).encode())
        log("Incorrectly formatted request")
        conn.close()
        continue

    if command[1] not in thermometer.device_commands.keys():
        conn.send(json.dumps({"code": 3}).encode())
        log("The command was not valid for this device")
        conn.close()
        continue

    if not auth_key_is_valid(command[0]):
        conn.send(json.dumps({"code": 1}).encode())
        log("Their auth key wasn't valid")
        conn.close()
        continue

    """
    Bad practise not to specify an error type, however we don't care what
    it is, we just want to the device to let the controller know. Therefore
    it is appropriate in this case.
    """

    log("They called " + command[1])
```

```
    try:
        to_send = thermometer.device_commands[command[1]]()
        log("Command calling successful!")
    except Exception as e:
        log("Problem executing command occurred...")
        to_send = json.dumps({"code": 4, "error":e})

    log("We will send back " + to_send)

    to_send = to_send.encode()
    conn.send(to_send)
    conn.close()
```

main.py (for thermometer)

```
import os
from utilities import *
from configuration import *
import socket
import json
from switch import Switch

log("File imports sucessful!")

VERBOSE = True

HOST = "0.0.0.0"
DEVICE_PORT = 24601
SWITCH_PIN_NUM = 5

generate_auth_key_if_not_exists()

"""
Unfortunately micropython on the ESP8266 does not support threading at this point
and so only 1 request can be processed at once, however this should be fine as
request collisions should not occur very often.
"""

switch = Switch(SWITCH_PIN_NUM)

"""
log() logs the string to a text file ("log.txt") which we can have a look at
```

```
incase anything goes wrong or we need to see what requests have been occuring
on the device.
"""

log("Switch object instantiated!")

# Connection Manager

sk = socket.socket()
sk.bind((HOST, DEVICE_PORT))
sk.listen(1)

while True:
    log("Listening...")
    conn, addr = sk.accept()

    log("Request from " + addr[0])

    data = conn.recv(2048).decode()

    log("They sent " + data)

    command = data.split(",")

    if len(command) != 2:
        conn.send(json.dumps({"code": 2}).encode())
        log("Incorrectly formatted request")
        conn.close()
        continue

    if command[1] not in switch.device_commands.keys():
        conn.send(json.dumps({"code": 3}).encode())
        log("The command was not valid for this device")
        conn.close()
        continue

    if not auth_key_is_valid(command[0]):
        conn.send(json.dumps({"code": 1}).encode())
        log("Their auth key wasn't valid")
        conn.close()
        continue

    """
    Bad practise not to specify an error type, however we don't care what
    it is, we just want to the device to let the controller know. Therefore
    it is appropriate in this case.
```

```
    """

    log("They called " + command[1])

    try:
        to_send = switch.device_commands[command[1]]()
        log("Command calling successful!")
    except Exception as e:
        log("Problem executing command occurred...")
        to_send = json.dumps({"code": 4, "error":e})

    log("We will send back " + to_send)

    to_send = to_send.encode()
    conn.send(to_send)
    conn.close()
```

switch.py

```
import machine
import json
from utilities import *

from device import Device

class Switch(Device):
    def __init__(self, pin_number):
        # calls init of the device superclass.
        super().__init__()

        new_commands = {
            "tgl": self.toggle,
            "tof": self.open,
            "ton": self.close,
            "gcs": self.is_closed
        }

        self.type = "Switch"

        self.device_commands = merge_dicts(self.device_commands, new_commands)

        # Creates a pin object and specifies it is for output not inputself.
        self._pin = machine.Pin(pin_number, machine.Pin.OUT)
```

```
    def is_closed(self):
        return json.dumps({"code": 0, "state": str(self._pin.value())})

    def close(self):
        self._pin.value(1)
        return json.dumps({"code": 0})

    def open(self):
        self._pin.value(0)
        return json.dumps({"code": 0})

    def toggle(self):
        self._pin.value(not(self._pin.value()))
        return json.dumps({"code": 0})
```

thermometer.py

```
import machine
import json
from utilities import *

from device import Device

class Thermometer(Device):
    def __init__(self, pin_number):
        # calls init of the device superclass.
        super().__init__()

        new_commands = {
            "gtm": self.get_temperature,
        }

        self.type = "Thermometer"

        self.device_commands = merge_dicts(self.device_commands, new_commands)

        # Creates an analog to digital conversion pin
        self._pin = machine.ADC(pin_number)

    def get_temperature(self):
        """
        When _pin.read() is called, a value between 1024 and 0 gets returned,
        this can be scaled to return a temperature.
```

```
        500 reading are taken and a mean is found as the temperature fluxtates
        on single readings. This gives a nice, accurate value.
        """

        temperature = sum([(self._pin.read()/1024) * 3300 * 0.1 for i in
range(500)])/500

        return json.dumps({"code": 0, "temperature": temperature})
```

device.py

```
import json

class Device:
    def __init__(self):
        self.type = "Not Assigned"
        self.device_commands = {
            "enq": self.enquire,
            "gtp": self.get_type
        }

    def enquire(self):
        return json.dumps({"code": 0})

    def get_type(self):
        return json.dumps({"code": 0, "type": self.type})
```

utilities.py

```
import random
import math
import os
import time

AUTH_KEY_FILE_NAME = "authentication_key.txt"
LOG_FILE = "log.txt"
KEY_ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456789"

"""
ESP only has the random function getrandombits(n) where you get a random number
```

```
below or equal to the binary number with n bits. Therefore, functions need to be
written to get a random int and choice.
"""

# Recursion to figure out how many bits are needed to represent a base n number
def how_many_bits_to_represent(number, bits=1, base=2):
    if (base**bits - 1) // number:
        return bits
    else:
        return how_many_bits_to_represent(number, bits+1)

def randint(lower_bound, upper_bound):
    bits_needed = how_many_bits_to_represent(upper_bound - lower_bound)

    random.seed(time.ticks_cpu())
    random_number = random.getrandbits(bits_needed)

    while random_number > (upper_bound - lower_bound):
        random_number = random.getrandbits(bits_needed)

    return lower_bound + random_number

def random_choice(iterable):
    return iterable[randint(0, len(iterable) - 1)]

def read_file(filename):
    with open(filename, "r") as f:
        data = f.read()

    return data

def pretty_read_file(filename):
    data = read_file(filename)

    [print(line) for line in data.split("\n")]

def generate_auth_key_if_not_exists():
    if AUTH_KEY_FILE_NAME not in os.listdir():
        authentication_key = "".join([random_choice(KEY_ALPHABET) for i in
range(32)])

        with open(AUTH_KEY_FILE_NAME, "w") as f:
            f.write(authentication_key)

def auth_key_is_valid(supplied_auth_key):
    auth_key = read_file(AUTH_KEY_FILE_NAME)
```

```
    return auth_key == supplied_auth_key

def merge_dicts(dict1, dict2):
    # If duplicate keys exist, dict1's key-value pair is used.

    for key in dict1.keys():
        dict2[key] = dict1[key]

    return dict2

def log(text):
    with open(LOG_FILE, "a") as f:
        f.write(text + "\n")
```

Website

main.js

```
const API_PORT = 5000;
const VERBOSE = true;
const TIMEOUT = 5000; // in ms
const DEFAULT_API_HOSTNAME = "http://localhost";

function ApiRequest(command=""){
  var postValues = {};
  var command = command;

  /*
  Due to the ascynhronous nature of javascript and these types of requests, a
function
  must be defined to be called upon the response of the API. This can be set
  after a new object is instantiated so it can be done on a case by
  case basis.
  */

  var calledOnResponse = function(response){
    alert(response.status.code.toString() + ": " + response.status.text);
  };
```

```
  this.setCalledOnResponse = function(func){
    calledOnResponse = func;
  };

  this.setCommand = function(command){
    this.command = command;
  };

  this.addContent = function(content, parameter){
      postValues[parameter] = content;
  };

  this.send = function(){
    var req = new XMLHttpRequest();

    /*
    The next line waits (but lets the code carry on much like a new thread)
    for the time TIMEOUT then checks if the request has been completed
successfully,
    if not it alerts the user accordingly because the right hostname for the API
    probably hasn't been set.
    */

    setTimeout(
      function(){
        if (req.readyState != 4){
          alert("Sorry, there was a problem connecting to the controller server,
check the hostname is correct.")
          console.log("Server connection issue. Perhaps check that the server
hostname is set correctly.")
        };
      }, TIMEOUT
    );

    var url = getCookie("ApiHostname") + ":" + API_PORT.toString() + "/" + command;

    req.onreadystatechange = function() {
      if (this.readyState == 4 && this.status == 200){ // if the request is done
and was successful
        console.log(this.resonseText);
        calledOnResponse(JSON.parse(this.responseText));
      }

      else if(this.readyState == 4 && this.status != 200){ // if the request is
done and not successful
```

```
        alert("There was a problem with the API; contact help.");
        console.log("API Connection Problem: Error " + this.status.toString());
      };
    };

    req.open("POST", url, true);
    req.setRequestHeader("Content-Type", "application/json"); // this header must
be set to let the flask API know it's getting JSON

    if (VERBOSE){
      console.log("ApiRequest sent to " + command);
      console.log("With POST values " + JSON.stringify(postValues));
    }

    req.send(JSON.stringify(postValues));
  }
}

function getCookie(cookieName){
  var cookieData = document.cookie;

  /*
  There's no function to get a single cookie,
  instead you have to write one to parse a string of ; seperated
  cookies yourself.
  */

  if (VERBOSE){
    console.log("Cookies: " + cookieData);
  }

  var cookiePairs = cookieData.split("; ");

  var cookies = {};

  for (i = 0; i < cookiePairs.length; i++){
    var pair = cookiePairs[i].split("=");

    cookies[pair[0]] = pair[1];
  }

  return cookies[cookieName];
}

function getGETParameter(parameter){
 var getData = window.location.search.substr(1)
```

```
 var getPairs = getData.split("&");

 var gets = {};

 for (i = 0; i < getPairs.length; i++){
   var pair = getPairs[i].split("=");

   gets[pair[0]] = pair[1];
  }

  return gets[parameter];
}

function deleteCookie(cookie){
  /*
  Sets expiry date in the past and the browser takes care of it, no other way to do
it...
  */

  document.cookie = cookie + '=; expires=Thu, 01 Jan 1970 00:00:01 GMT;';
}

function goToLoginIfSessionKeyNotValid(){
  var sessionKey = getCookie("sessionKey");

  if (! sessionKey){
    document.location.href = "/login.html";
    return
  };

  var req = new ApiRequest("is_session_key_valid");

  req.setCalledOnResponse(
    function(response){
      if (response.status.code != 1){
        alert("Error, code " + response.status.code.toString() + ": " +
response.status.text)
        return
      };

      if (response.is_valid == "false"){
        document.location.href = "/login.html";
        return
      }
      else {
```

```
        console.log("Session key okay!");
        return
      };
    }
  );

  req.addContent(sessionKey, "session_key");

  req.send()
}

function log_out(){
  var req = new ApiRequest("log_out");

  req.setCalledOnResponse(
    function(response){
      if (response.status.code != 1){
        alert("Error, code " + response.status.code.toString() + ": " +
response.status.text)
        return
      }

      else {
        deleteCookie("sessionKey");
        document.location.href = "/login.html";
        return
      };
    }
  );

  var sessionKey = getCookie("sessionKey");
  req.addContent(sessionKey, "session_key")

  req.send();
}

function setApiHostname(hostname){
  var currentDate = new Date();
  currentDate.setTime(currentDate.getTime() + (30*365*24*60*60*1000)); // 30 years
exipry date, no other way to make no exipry cookie...
  document.cookie = "ApiHostname=" + hostname + "; expires=" +
currentDate.toGMTString() + "; path=/";
}

function setApiHostnameAsDefaultIfNotSet(){
  var hostname = getCookie("ApiHostname");
```

```
  if (! hostname){
    setApiHostname(DEFAULT_API_HOSTNAME);
  };
}

function closeHostnameSetter(){
  var menu = document.getElementById("hostnameMenu");
  menu.parentNode.removeChild(menu);
}

function openHostnameSetter(){
  if (document.getElementById("hostnameMenu") != undefined){
    return;
  };

  var hostnameDiv = document.createElement("div");
  hostnameDiv.className = "hostname_setter_container";
  hostnameDiv.setAttribute("id", "hostnameMenu");

  var instructionsDiv = document.createElement("div");
  instructionsDiv.className = "hostname_setter_label";
  var tempTextNode = document.createTextNode("The current hostname is displayed
below, press the exit button to leave as is, or enter a new one and press apply to
set a new one. (Must include http:// etc)");
  instructionsDiv.appendChild(tempTextNode);

  var exitButtonDiv = document.createElement("button");
  exitButtonDiv.className = "normal_button hostname_setter_exit";
  exitButtonDiv.setAttribute("onclick", "closeHostnameSetter()");
  var tempTextNode = document.createTextNode("X");
  exitButtonDiv.appendChild(tempTextNode);

  var hostnameEntry = document.createElement("input");
  hostnameEntry.className = "normal_text_entry hostname_setter_text_entry";
  hostnameEntry.setAttribute("id", "new_hostname");
  hostnameEntry.setAttribute("type", "text");
  hostnameEntry.setAttribute("value", getCookie("ApiHostname"));

  var applyButton = document.createElement("button");
  applyButton.className = "normal_button hostname_setter_apply";
  applyButton.setAttribute("onclick",
"setApiHostname(document.getElementById('new_hostname').value);closeHostnameSetter(
);");
  var tempTextNode = document.createTextNode("Apply");
  applyButton.appendChild(tempTextNode);
```

```
  elementsToAdd = [instructionsDiv, exitButtonDiv, hostnameEntry, applyButton];

  for (i = 0; i < elementsToAdd.length; i++){
    hostnameDiv.appendChild(elementsToAdd[i]);
  };

  document.body.appendChild(hostnameDiv);
}

function randomChoice(array){
  var randomIndex = Math.floor(Math.random() * array.length);
  return array[randomIndex]
}

function removeElementByAttributeAndValue(attribute, value){
  document.querySelectorAll("[" + attribute + "=" + value +
"]")[0].parentElement.removeChild(document.querySelectorAll("[" + attribute + "=" +
value + "]")[0]);
}
```

login.js

```
function login(){
  var details = {
    "username": document.getElementById("username").value,
    "password": document.getElementById("password").value
  };

  var req = new ApiRequest("login");

  req.setCalledOnResponse(
    function(response){
      if (response.status.code != 1){
        alert("Problem logging in: \n\n" + response.status.text);
      }

      else{
        // Expiry is set as late as possible.
        document.cookie = "sessionKey=" + response.session_key + "; expires=Fri, 31
Dec 9999 23:59:59 GMT; path=/";
        document.location.href = "/list_devices.html";
      };
```

```
      return;
    }
  );

  if (VERBOSE){
    console.log("Details added to ApiRequest object:");
  };

  for (detail in details){
    if (VERBOSE){
      console.log(detail + ": " + details[detail]);

    };
    req.addContent(details[detail], detail);
  }

  req.send();
}

function goToListDevicesIfSessionKeyValid(){
  var sessionKey = getCookie("sessionKey");

  if (! sessionKey){
    if (VERBOSE){
      console.log("No session key.")
    };

    return;
  };

  var req = new ApiRequest("is_session_key_valid");

  req.setCalledOnResponse(
    function(response){
      if (response.status.code != 1){
        alert("Error, code " + response.status.code.toString() + ": " +
response.status.text)
        return;
      };

      if (response.is_valid == "true"){
        document.location.href = "/list_devices.html";
        return;
      }
      else {
        console.log("Session key invalid or not active.");
```

```
      };
    }
  );

  req.addContent(sessionKey, "session_key");

  req.send()
}
```

list_devices.js

```
function addDeviceToPage(device){
  var deviceElement = document.createElement("div");  // creating the main div
container for the device
  deviceElement.className = "device_description_container";
  deviceElement.setAttribute("deviceID", device.id);

  // purely aesthetic random border colour setting below

  deviceElement.setAttribute("style", "border-color: " + randomChoice(["#FF8987",
"#728AFF", "#7FFFA1"]));

  var subElements = { // elements to put in the device container
    name: {
      elementType: "div",
      content: device.name,
      label: "",
      className: "device_name"
    },

    owner: {
      elementType: "div",
      content: device.owner,
      label: "Owned by ",
      className: "owner_username"
    },

    type: {
      elementType: "div",
      content: device.type,
      label: "Type: ",
      className: "device_type"
    },
```

```
    button: {
      elementType: "button",
      content: "More",
      label: "",
      className: "normal_button device_button"
    }
  };

  if (VERBOSE){
    console.log("Device added:");
  };

  for (element in subElements){
    subElements[element].element =
document.createElement(subElements[element].elementType);
    subElements[element].element.className = subElements[element].className;

    var elementText = subElements[element].label + subElements[element].content;
    var tempTextNode = document.createTextNode(elementText);
    subElements[element].element.appendChild(tempTextNode);

    deviceElement.appendChild(subElements[element].element);

    if (VERBOSE){
      console.log(element + ": " + subElements[element].content);
    };
  };

  subElements.button.element.setAttribute("onclick", "goToDevicePage(this);");


document.getElementsByClassName("centered_container")[0].appendChild(deviceElement)
;
};

function updateListedDevices(){
  req = new ApiRequest("list_devices");

  req.setCalledOnResponse(
    function(response){
      var deviceElements =
document.getElementsByClassName("device_description_container");

      /*
      The code below removes all elements of class name
```

```
"device_description_container". It works
      and is acceptable because when document.getElements<etc> is called, a static
array is not returned
      instead a "live" iterable object is returned and so when an element is
removed from the actual document
      it is removed from the "list".
      */

      while(deviceElements[0]){
        deviceElements[0].parentNode.removeChild(deviceElements[0]);
      };

      for (device in response.devices){
        addDeviceToPage(response.devices[device]);
      };

      var refreshButton = document.getElementsByClassName("normal_button
refresh_button")[0];
      refreshButton.innerHTML = "Refresh Devices";
    }
  );

  req.addContent(getCookie("sessionKey"), "session_key");

  var refreshButton = document.getElementsByClassName("normal_button
refresh_button")[0];
  if (refreshButton != undefined){
    refreshButton.innerHTML = "Talking to API...";
  };

  req.send();
}

function goToDevicePage(buttonElement){ // function for the "more" button to send
the clicker to the device page of the container
  document.location.href = '/device.html?deviceID=' +
buttonElement.parentNode.getAttribute('deviceID');
}
```

create_user.js

```
function create_user(){
  var details = {
```

```
    "new_username": document.getElementById("username").value,
    "new_password": document.getElementById("password").value
  };


  var emailValue = document.getElementById("email").value;

  if (emailValue != ""){
        details["email"] = emailValue;
  }

  var req = new ApiRequest("create_user");

  req.setCalledOnResponse(
    function(response){
      if (response.status.code != 1){
        alert("Problem logging in: \n\n" + response.status.text);
      }

      else{
        alert("Successful user creation, redirecting to login page")
        document.location.href = "/login.html";
      };

      return;
    }
  );

  if (VERBOSE){
    console.log("Details added to ApiRequest object:");
  };

  for (detail in details){
    if (VERBOSE){
      console.log(detail + ": " + details[detail]);

    };
    req.addContent(details[detail], detail);
  }

  req.send();
}
```

device.js

```
function refreshTemperature(deviceID){
  req = new ApiRequest("call_command_on_device");

  req.setCalledOnResponse(
    function (response){
      if (response.status.code != 1){
        alert("Error, code " + response.status.code.toString() + ": " +
response.status.text)
        return;
      };

      console.log("Updated device temp!")
      console.log(response.device_response)
      var statusElement = document.getElementsByClassName("current_temp")[0]
      statusElement.innerHTML = response.device_response.temperature.toString()
    }
  );

  req.addContent(getCookie("sessionKey"), "session_key");
  req.addContent(getGETParameter("deviceID"), "device_id");
  req.addContent("gtm", "command_code");

  var temperatureElement = document.getElementsByClassName("current_temp")[0]
  temperatureElement.innerHTML = "Finding"

  req.send();
}

function applyPermission(){
  /*
  This function gathers up all of the permission data from the page
  and sends it over to the API to set as the new permission for this device.
  */

  req = new ApiRequest("set_device_permission");

  var applyButton = document.getElementsByClassName("normal_button
apply_permission")[0];
  applyButton.innerHTML = "Applying..."

  req.setCalledOnResponse(
    function (response){
      if (response.status.code != 1){
```

```
        alert("Error applying permission, code " + response.status.code.toString()
+ ": " + response.status.text)
        return;
      };

      var applyButton = document.getElementsByClassName("normal_button
apply_permission")[0];
      applyButton.innerHTML = "Apply Permission"
      updatePermission();
    }
  );

  var permissionDropdown = document.getElementById("permission_types_dropdown");
  var permissionType =
permissionDropdown.options[permissionDropdown.selectedIndex].text;

  var userContainerElementHTMLCollection =
document.getElementsByClassName("permission_username");
  var userContainersElementArray =
Array.prototype.slice.call(userContainerElementHTMLCollection);
  var usernamesToBeAdded = userContainersElementArray.map(function(element){return
element.innerHTML});

  if (VERBOSE){
    console.log("To be added: ")
    console.log(usernamesToBeAdded)
  }

  req.addContent(usernamesToBeAdded, "users");
  req.addContent(permissionType, "permission_type");
  req.addContent(getCookie("sessionKey"), "session_key");
  req.addContent(getGETParameter("deviceID"), "device_id");
  req.send();
}

function addSelectedUserToPermission(){
  var userDropdown = document.getElementById("users_dropdown");
  var username = userDropdown.options[userDropdown.selectedIndex].text;

  var userContainers = document.querySelectorAll("[user_in_permission=" + username
+ "]");
  console.log(userContainers);
  if (userContainers.length > 0){
    return;
  };
```

```
  addUser(username);
}

function addUser(username){
  // Creates a user container on the page with the username supplied.

  var permissionsBox = document.getElementById("permissions_box");

  var userContainer = document.createElement("div");
  userContainer.className = "permission_user_container";
  userContainer.setAttribute("user_in_permission", username);

  var usernameLabel = document.createElement("div");
  usernameLabel.className = "permission_username";
  var tempTextNode = document.createTextNode(username);
  usernameLabel.appendChild(tempTextNode);

  var removalButton = document.createElement("button");
  var tempTextNode = document.createTextNode("Remove");
  removalButton.appendChild(tempTextNode);
  removalButton.className = "normal_button remove_user_from_permission";
  removalButton.setAttribute("onclick", "removeElementByAttributeAndValue('" +
"user_in_permission" + "', '" + username + "')");

  userContainer.appendChild(removalButton);
  userContainer.appendChild(usernameLabel);

  permissionsBox.appendChild(userContainer);
}

function updatePermission(){
  // Request for list of device types and set them in the dropdown options menu

  req = new ApiRequest("get_permission_types");

  req.setCalledOnResponse(
    function (response){
      if (response.status.code != 1){
        alert("Error, code " + response.status.code.toString() + ": " +
response.status.text)
        return;
      };

      var dropdownElement = document.getElementById("permission_types_dropdown");
      dropdownElement.innerHTML = "";
```

```
      for (type in response.types){
        var tempElement = document.createElement("option");
        tempElement.setAttribute("id", response.types[type].name)
        var tempTextNode = document.createTextNode(response.types[type].name);
        tempElement.appendChild(tempTextNode);
        dropdownElement.appendChild(tempElement);
      };
    }
  );

  req.send();

  // Request for list of users and then set them as options in the dropdown options
menu

  req = new ApiRequest("get_users_list");

  req.setCalledOnResponse(
    function (response){
      if (response.status.code != 1){
        alert("Error, code " + response.status.code.toString() + ": " +
response.status.text)
        return;
      };

      var dropdownElement = document.getElementById("users_dropdown");
      dropdownElement.innerHTML = "";

      for (user in response.users){
        var tempElement = document.createElement("option");
        var tempTextNode = document.createTextNode(response.users[user]);
        tempElement.appendChild(tempTextNode);
        dropdownElement.appendChild(tempElement);
      };
    }
  );

  req.send()

  // Gets the users related to the current device permission and the permission
type and adds them to the page

  req = new ApiRequest("get_device_permission_info");

  req.setCalledOnResponse(
    function (response){
```

```
      if (response.status.code != 1){
        alert("Error, code " + response.status.code.toString() + ": " +
response.status.text)
        return;
      };

      var permissionTypesDropdown =
document.getElementById("permission_types_dropdown");
      var indexOfPermissionType =
Array.prototype.slice.call(permissionTypesDropdown.options).indexOf(document.getEle
mentById(response.permission.type))
      permissionTypesDropdown.selectedIndex = indexOfPermissionType;

      var currentUserElements =
document.getElementsByClassName("permission_user_container");

      while (currentUserElements[0]){
        currentUserElements[0].parentNode.removeChild(currentUserElements[0]);
      };

      console.log(response.permission);

      for (user in response.permission.users){
        addUser(response.permission.users[user]);
      };
    }
  );

  req.addContent(getCookie("sessionKey"), "session_key");
  req.addContent(getGETParameter("deviceID"), "device_id");
  req.send();
}

function buildDevicePermissions(device){
  // Creating the permissions box
  var permissionsBox = document.createElement("div");
  permissionsBox.className = "permissions_box";
  permissionsBox.setAttribute("id", "permissions_box");

  // Elements to put in the toggle box.
  var subElements = {
    title: {
      elementType: "div",
      content: "Device Permission",
      className: "permissions_title"
    },
```

```
    applyButton: {
      elementType: "button",
      content: "Apply Permission",
      className: "normal_button apply_permission"
    },

    description: {
      elementType: "div",
      content: "The info below shows the type of permission on the device I.E.
blacklist, whitelist etc. The users shown below the add user field (if there are
any) are ones relating to the permission hence we use a blacklist, they will be
blacklisted and the same for a whitelist. If using a permission type where having
users related to it is irrellevant, they are simply ignored.",
      className: "permission_explaination"
    }
  };

  for (element in subElements){
    subElements[element].element =
document.createElement(subElements[element].elementType);
    subElements[element].element.className = subElements[element].className;

    var tempTextNode = document.createTextNode(subElements[element].content);
    subElements[element].element.appendChild(tempTextNode);

    permissionsBox.appendChild(subElements[element].element);

    if (VERBOSE){
      console.log(element + ": " + subElements[element].content);
    };
  };

  subElements.applyButton.element.setAttribute("onClick", "applyPermission()");


document.getElementsByClassName("centered_container")[0].appendChild(permissionsBox
);

  // Making the Permission Type selector

  var permissionTypes = document.createElement("div");
  permissionTypes.className = "permission_header_container";

  var permissionTypesLabel = document.createElement("div");
  permissionTypesLabel.className = "permission_types_label";
```

```
  var tempTextNode = document.createTextNode("Permission Type: ");
  permissionTypesLabel.appendChild(tempTextNode);
  permissionTypes.appendChild(permissionTypesLabel);

  var permissionsDropdown = document.createElement("select");
  permissionsDropdown.className = "permissions_dropdown";
  permissionsDropdown.setAttribute("id", "permission_types_dropdown");
  permissionTypes.appendChild(permissionsDropdown);

  permissionsBox.appendChild(permissionTypes);

  // Making the User Adder

  var userSelect = document.createElement("div");
  userSelect.className = "permission_header_container";

  var userSelectLabel = document.createElement("div");
  userSelectLabel.className = "user_select_label";
  var tempTextNode = document.createTextNode("Add User: ");
  userSelectLabel.appendChild(tempTextNode);
  userSelect.appendChild(userSelectLabel);

  var userSelectDropdown = document.createElement("select");
  userSelectDropdown.className = "permissions_dropdown user_select";
  userSelectDropdown.setAttribute("id", "users_dropdown");
  userSelect.appendChild(userSelectDropdown);

  var addUserButton = document.createElement("button");
  addUserButton.className = "normal_button add_user_to_permission";
  var tempTextNode = document.createTextNode("Add");
  addUserButton.appendChild(tempTextNode);
  addUserButton.setAttribute("onClick", "addSelectedUserToPermission()");
  userSelect.appendChild(addUserButton);

  permissionsBox.appendChild(userSelect);

  // Function below updates the options with fresh data from the controller server

  updatePermission();
}

function updateStatus(){
  req = new ApiRequest("call_command_on_device");

  req.setCalledOnResponse(
    function (response){
```

```
      if (response.status.code != 1){
        alert("Error, code " + response.status.code.toString() + ": " +
response.status.text)
        return;
      };

      console.log("Updated device state!")
      console.log(response.device_response)
      var statusElement = document.getElementsByClassName("current_state")[0]
      statusElement.innerHTML = (response.device_response.state == "0") ? "Current
State: Off" : "Current State: On"
    }
  );

  req.addContent(getCookie("sessionKey"), "session_key");
  req.addContent(getGETParameter("deviceID"), "device_id");
  req.addContent("gcs", "command_code");

  var statusElement = document.getElementsByClassName("current_state")[0]
  statusElement.innerHTML = "Querying status..."

  req.send();
}

function toggleDevice(){
  req = new ApiRequest("call_command_on_device");

  req.setCalledOnResponse(
    function (response){
      if (response.status.code != 1){
        alert("Error, code " + response.status.code.toString() + ": " +
response.status.text)
        return;
      };

      updateStatus(getGETParameter("deviceID"));
    }
  );

  req.addContent(getCookie("sessionKey"), "session_key");
  req.addContent(getGETParameter("deviceID"), "device_id");
  req.addContent("tgl", "command_code");

  var statusElement = document.getElementsByClassName("current_state")[0]
  statusElement.innerHTML = "Talking to device..."
```

```
  req.send();
}

function buildPage(device){
  // Creates a device title at the top of page.
  var deviceTitle = document.createElement("div");
  deviceTitle.className = "big_title";
  var tempTextNode = document.createTextNode(device.name);
  deviceTitle.appendChild(tempTextNode);

document.getElementsByClassName("centered_container")[0].appendChild(deviceTitle);


  // If the device is a switch, a toggle box is made.
  if (device.type == "Switch"){
    // Creating the main div container for the toggle box.
    var toggleBox = document.createElement("div");
    toggleBox.className = "toggle_box";

    // Elements to put in the toggle box.
    var subElements = {
      toggle: {
        elementType: "button",
        content: "Toggle",
        className: "normal_button toggle_button"
      },

      state: {
        elementType: "div",
        content: "Querying State...",
        className: "current_state"
      },

      owner_and_type: {
        elementType: "div",
        content: device.type + " owned by " + device.owner,
        className: "owner_and_type"
      }
    };

    if (VERBOSE){
      console.log("Toggle Box added:");
    };

    for (element in subElements){
      subElements[element].element =
```

```
document.createElement(subElements[element].elementType);
      subElements[element].element.className = subElements[element].className;

      var tempTextNode = document.createTextNode(subElements[element].content);
      subElements[element].element.appendChild(tempTextNode);

      toggleBox.appendChild(subElements[element].element);

      if (VERBOSE){
        console.log(element + ": " + subElements[element].content);
      };
    };

    subElements.toggle.element.setAttribute("onClick", "toggleDevice('" +
device.id.toString() + "')");


document.getElementsByClassName("centered_container")[0].appendChild(toggleBox);

    updateStatus(device.id.toString())
  }

  else{
    // Creating the main div container for the thermometer box.
    var thermometerBox = document.createElement("div");
    thermometerBox.className = "thermometer_box";

    // Elements to put in the thermometer box.
    var subElements = {
      button: {
        elementType: "button",
        content: "Refesh Reading",
        className: "normal_button toggle_button"
      },

      temperature: {
        elementType: "div",
        content: "Querying Temp...",
        className: "current_temp"
      },

      owner_and_type: {
        elementType: "div",
        content: device.type + " owned by " + device.owner,
        className: "owner_and_type"
      }
```

```
    };

    if (VERBOSE){
      console.log("Thermometer Box added:");
    };

    for (element in subElements){
      subElements[element].element =
document.createElement(subElements[element].elementType);
      subElements[element].element.className = subElements[element].className;

      var tempTextNode = document.createTextNode(subElements[element].content);
      subElements[element].element.appendChild(tempTextNode);

      thermometerBox.appendChild(subElements[element].element);

      if (VERBOSE){
        console.log(element + ": " + subElements[element].content);
      };
    };

    subElements.button.element.setAttribute("onClick", "refreshTemperature('" +
device.id.toString() + "')");


document.getElementsByClassName("centered_container")[0].appendChild(thermometerBox
);

    refreshTemperature(device.id.toString())
  };

  buildDevicePermissions();
}

function loadDeviceInfo(){
  req = new ApiRequest("get_device_info");

  req.setCalledOnResponse(
    function(response){
      if (response.status.code == 9){
        alert("Sorry, you don't have permission to view this device.");
        window.location.href = "/list_devices.html";
        return;
      }

      else if (response.status.code != 1){
```

```
        alert("Error, code " + response.status.code.toString() + ": " +
response.status.text)
      }

      else {
        buildPage(response.device);
      };
         }
  );

  req.addContent(getCookie("sessionKey"), "session_key");
  req.addContent(getGETParameter("deviceID"), "device_id")

  req.send();
}
```

add_device.js

```
function addDevice(){
  var details = {
    "device_name": document.getElementById("device_name").value,
    "device_ip": document.getElementById("device_ip").value,
    "auth_key": document.getElementById("auth_key").value,
    "session_key": getCookie("sessionKey")
  };

  var req = new ApiRequest("add_device");

  req.setCalledOnResponse(
    function(response){
      if (response.status.code != 1){
        alert("Problem creatign device: \n\n" + response.status.text);
      }

      else{
        alert("Successful adding; redirecting to list devices!")
        document.location.href = "/list_devices.html";
      };

      var add_button = document.getElementById("add_device_button");
      add_button.innerHTML = "Add Device";

      return;
    }
```

```
  );

  if (VERBOSE){
    console.log("Details added to ApiRequest object:");
  };

  for (detail in details){
    if (VERBOSE){
      console.log(detail + ": " + details[detail]);

    };
    req.addContent(details[detail], detail);
  }

  var add_button = document.getElementById("add_device_button");
  add_button.innerHTML = "Adding...";

  req.send();
}
```

HTML and CSS Files

list_devices.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans"
rel="stylesheet">
    <link rel="stylesheet" type="text/css" href="/smart.css">

    <title>Smart Home - Devices</title>
    <script src="/js/main.js"></script>
    <script src="/js/list_devices.js"></script>
    <script>
      setApiHostnameAsDefaultIfNotSet();
      goToLoginIfSessionKeyNotValid();
      updateListedDevices();
    </script>
  </head>

  <body>
```

```
    <div class="centered_container">
      <div class="navbar">
        <button onclick="window.location.href = '/list_devices.html'"> List Devices
</button>
        <button onclick="openHostnameSetter();"> Change API Host </button>
        <button onclick="window.location.href = '/add_device.html'"> Add Device
</button>
        <button onclick="log_out()"> Log Out </button>
      </div>

      <div class="devices_title_container">
        <div class="big_title list_devices"> Devices </div>
        <button class="normal_button refresh_button"
onclick="updateListedDevices()"> Refresh Devices </button>
      </div>
    </div>
  </body>
</html>
```

device.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans"
rel="stylesheet">
    <link rel="stylesheet" type="text/css" href="/smart.css">

    <title>Smart Home - Device Manager</title>
    <script src="/js/main.js"></script>
    <script src="/js/device.js"></script>
    <script>
      setApiHostnameAsDefaultIfNotSet();
      goToLoginIfSessionKeyNotValid();
    </script>
  </head>

  <body>
    <div class="centered_container">
      <div class="navbar">
        <button onclick="window.location.href = '/list_devices.html'"> List Devices
</button>
        <button onclick="openHostnameSetter();"> Change API Host </button>
```

```
        <button onclick="window.location.href = '/add_device.html'"> Add Device
</button>
        <button onclick="log_out()"> Log Out </button>
      </div>

      <script>
        loadDeviceInfo();
      </script>

    </div>
  </body>
</html>
```

add_device.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans"
rel="stylesheet">
    <link rel="stylesheet" type="text/css" href="/smart.css">

    <title>Smart Home - Create User</title>
    <script src="/js/main.js"></script>
    <script src="/js/add_device.js"></script>
    <script>
      setApiHostnameAsDefaultIfNotSet();
    </script>
  </head>

  <body>
    <div class="centered_container">
      <div class="navbar">
        <button onclick="window.location.href = '/list_devices.html'"> List Devices
</button>
        <button onclick="openHostnameSetter();"> Change API Host </button>
        <button onclick="window.location.href = '/add_device.html'"> Add Device
</button>
        <button onclick="log_out()"> Log Out </button>
      </div>
    </div>

    <div class="create_centered_container">
      <div class="big_title"> Add Device </div>
```

```
      <input id="device_name" type="text" class="normal_text_entry
center_horizontally" placeholder="Device Name"></input>
      <br>
      <input id="device_ip" type="text" class="normal_text_entry
center_horizontally" placeholder="Device IP (E.G. 192.168.1.234)"></input>
      <br>
      <input id="auth_key" type="text" class="normal_text_entry
center_horizontally" placeholder="Device Authentication Key"></input>
      <br>
      <button id="add_device_button" class="normal_button center_horizontally
spaced" onclick="addDevice()" style="width: 120px"> Add Device </button>
    </div>
  </body>
</html>
```

create_user.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans"
rel="stylesheet">
    <link rel="stylesheet" type="text/css" href="/smart.css">

    <title>Smart Home - Create User</title>
    <script src="/js/main.js"></script>
    <script src="/js/create_user.js"></script>
    <script>
      setApiHostnameAsDefaultIfNotSet();
    </script>
  </head>

  <body>
    <div class="create_centered_container">
      <div class="big_title"> Create User</div>
      <input id="username" type="text" class="normal_text_entry
center_horizontally" placeholder="Username"></input>
      <br>
      <input id="password" type="text" class="normal_text_entry
center_horizontally" placeholder="Password"></input>
      <br>
      <input id="email" type="text" class="normal_text_entry center_horizontally"
placeholder="Email (Optional)"></input>
```

```
      <br>
      <button class="normal_button center_horizontally spaced"
onclick="create_user()" style="width: 120px"> Create User </button>
      <button class="normal_button center_horizontally spaced"
onclick="window.location.href = '/login.html'" style="width: 80px"> Back </button>
    </div>
  </body>
</html>
```

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      document.location.href = "/login.html";
    </script>
  </head>
</html>
```

smart.css

```
body{
  text-align: center;
}

* {
    font-family: 'Open Sans',
Helvetica, Arial, sans-serif;
}

a{
      text-decoration: none;
}

div.login_centered_container{
  width: 400px;
  height: 635px;
}

div.big_title.list_devices{
  position: absolute;
  width: 350px;
  margin: 0;
  padding: 0;
  bottom: 12px;
  left: 20px;
}

button.normal_button.refresh_button{
  width: 200px;
  height: 80px;
  position: absolute;
  right: 10px;
  top: 16px;
}
```

```
  position: absolute;
  top: 0;
  bottom: 0;
  left: 0;
  right: 0;
  margin: auto;
}

div.create_centered_container{
  width: 400px;
  height: 450px;

  position: absolute;
  top: 0;
  bottom: 0;
  left: 0;
  right: 0;
  margin: auto;
}

.center_horizontally{
  display: block;
  margin: 0 auto;
}

input.normal_text_entry{
  box-shadow:0 0 5px 1px #C0C0C0;
  border-radius: 4px;
  border-width: 2px;
  border-color: black;
  border-style: solid;
  height: 50px;
  padding-left: 20px;
  padding-right: 20px;
  -webkit-box-sizing: border-box; /*
These stop the box coming out of the
parent */
  -moz-box-sizing: border-box;
  box-sizing: border-box;
  width: 250px;
}

div.toggle_box, div.thermometer_box{
  height: 160px;
  border-style: solid;
  border-width: 2px;
  border-color: black;
  border-radius: 15px;
  margin-top: 20px;
  margin-bottom: 20px;
  box-shadow:0 0 5px 0.1px black;
  position: relative;
}

button.normal_button.toggle_button{
  border-radius: 15px;
  position: absolute;
  width: 290px;
  height: 70px;
  left: 15px;
  bottom: 15px;
}

div.current_state, div.current_temp{
  font-size: 30px;
  overflow: hidden;
  text-align: right;

  white-space: nowrap;
  width: 300px;
  height: 50px;
  color: black;
  margin: 0px;
  position: absolute;
  bottom: 20px;
  right: 18px;
}

div.current_temp{
  font-size: 30px;
  text-align: left;

  white-space: nowrap;
```

```
input[type=password]{
  text-security: disc;
  -webkit-text-security: disc;
  -moz-text-security: disc;
}

button.normal_button{
  outline: none !important;
  font-weight: bold;
  color: white;
  border-radius: 25px;
  border-color: black;
  border-width: 0;
  border-style: solid;
  background-color: black;
  width: 100px;
  height: 35px;
}

button.normal_button.spaced{
  margin-top: 10px;
  margin-bottom: 10px;
}

button.normal_button:hover{
  background-color: #61cc7e;
}

div.centered_container{
    width: 600px;
    text-align: left;
    position: absolute;
    top: 0;
    bottom: 0;
    left: 0;
    right: 0;
    margin: auto;
    font-size: 0;
    vertical-align: middle;
  }
  padding: 10px 60px;
  color: black;
  margin: 0px;
  position: absolute;
  bottom: 20px;
  right: 18px;
}

div.owner_and_type{
  font-size: 30px;
  overflow: hidden;

  white-space: nowrap;
  width: 600px;
  height: 50px;
  color: black;
  margin: 0px;
  margin-left: 30px;
  position: absolute;
  top: 19px;
}

div.temperature_box{
  height: 160px;
  border-style: solid;
  border-width: 2px;
  border-color: black;
  border-radius: 15px;
  margin-top: 20px;
  margin-bottom: 20px;
  box-shadow:0 0 5px 0.1px black;
  position: relative;
}

div.current_temp_label{
  font-size: 30px;
  overflow: hidden;

  white-space: nowrap;
  width: 300px;
  height: 50px;
```

```
div.big_title{
  font-size: 50px;
  font-weight: bold;
  color: black;
  padding: 10px;
  padding-left: 15px;
  margin-top: 20px;
  margin-bottom: 20px;
}

div.device_description_container{
  height: 100px;
  border-style: solid;
  border-width: 2px;
  border-color: #728AFF;
  border-radius: 15px;
  margin-top: 20px;
  margin-bottom: 20px;
  box-shadow:0 0 5px 0.1px black;
  position: relative;
}

div.device_name{
  font-size: 30px;
  overflow-y: hidden
  overflow-x: scroll;

  white-space: nowrap;
  width: 225px;
  height: 50px;
  color: black;
  margin: 0px;
  margin-left: 17px;
  position: absolute;
  top: 11px;
}

div.owner_username{
  font-size: 18px;
  overflow: hidden;
  color: black;
  margin: 0px;
  position: absolute;
  bottom: 15px;
  left: 30px;
}

div.current_temp{
  font-size: 30px;
  text-align: center;
  overflow: hidden;
  height: 45px;
  width: 100px;
  border-style: solid;
  border-width: 2px;
  border-color: black;
  border-radius: 15px;
  margin: 0px;
  color: black;
  box-shadow:0 0 5px 0.1px black;
  position: absolute;
  bottom: 15px;
  right: 30px;
}

div.permissions_box{
  border-style: solid;
  border-width: 2px;
  border-color: black;
  border-radius: 15px;
  margin-top: 20px;
  margin-bottom: 20px;
  box-shadow:0 0 5px 0.1px black;
  position: relative;
}

div.permissions_title{
  display: inline-block;
  font-weight: bold;
  font-size: 40px;
  margin: 20px;
```

```
  white-space: nowrap;
  width: 225px;
  height: 30px;
  color: black;
  margin: 0px;
  margin-left: 17px;
  position: absolute;
  bottom: 10px;
}

div.device_type{
  font-size: 25px;
  overflow: hidden;
  white-space: nowrap;
  width: 250px;
  height: 40px;
  color: black;
  margin: 0px;
  margin-left: 17px;
  position: absolute;
  top: 32px;
  left: 240px;
}


button.normal_button.device_button{
  border-radius: 15px;
  position: absolute;
  width: 70px;
  height: 70px;
  right: 15px;
  top: 15px;
}

div.navbar{
  top: 0;
  background-color: grey;
  width: 100%;
  height: 30px;
}
}

button.normal_button.apply_permissio
n{
  border-radius: 15px;
  display: inline-block;
  height: 65px;
  width: 180px;
  padding: 5px;
  font-size: 20px;
}

div.permission_header_container{
  border-style: solid;
  border-width: 2px;
  border-color: black;
  border-radius: 15px;
  margin: 20px;
  box-shadow:0 0 5px 0.1px black;
  display: block;
  height: 100px;
  width: 550px;
  position: relative;
}

div.permission_types_label,
div.user_select_label{
  position: absolute;
  font-weight: bold;
  font-size: 30px;
  margin: 20px;
  top: 10px;
}

select.permissions_dropdown{
  box-shadow:0 0 5px 1px #C0C0C0;
  border-radius: 4px;
  border-width: 2px;
  border-color: black;
  border-style: solid;
  height: 40px;
```

```
div.navbar button{
  border-style: none;
  color: white;
  background-color: transparent;
  height: 100%;
}

div.navbar button:hover{
  background-color: black;
}

div.hostname_setter_container{
  text-align: center;
  height: 350px;
  width: 250px;
  z-index: 1;
  border-style: solid;
  border-width: 2px;
  border-color: black;
  border-radius: 15px;
  margin-bottom: 20px;
  background-color: white;
  box-shadow: 0 0 5px 0.1px black;
  display: inline-block;
  position: fixed;
  top: 50%;
  margin: -175px -125px;
}

button.normal_button.hostname_setter_
exit{
  text-align: center;
  border-radius: 5px;
  position: absolute;
  width: 30px;
  height: 30px;
  right: 15px;
  top: 15px;
}

input.normal_text_entry.hostname_sett
  padding-left: 20px;
  padding-right: 20px;
  -webkit-box-sizing: border-box; /*
These stop the box coming out of the
parent */
  -moz-box-sizing: border-box;
  box-sizing: border-box;
  width: 250px;
  right: 15px;
  top: 32px;
  position: absolute;
}

select.permissions_dropdown.user_sel
ect{
  right: 120px;
}

button.normal_button.add_user_to_per
mission{
  position: absolute;
  right: 10px;
  border-radius: 15px;
  height: 80px;
  width: 80px;
  padding: 5px;
  font-size: 20px;
  top: 10px;
}

div.permission_user_container{
  border-style: solid;
  border-width: 2px;
  border-color: black;
  border-radius: 15px;
  margin: 20px;
  box-shadow:0 0 5px 0.1px black;
  display: block;
  height: 50px;
  width: 550px;
  position: relative;
```

```
er_text_entry{
  width: 200px;
  position: absolute;
  bottom: 100px;
  right: 25px;
}

button.normal_button.hostname_setter_
apply{
  width: 100px;
  height: 30px;
  position: absolute;
  bottom: 50px;
  right: 75px;
}

div.hostname_setter_label{
  display: inline-block;
  font-size: 14px;
  margin-top: 55px;
  width: 200px;
  height: 300px;
}


div.devices_title_container{
  position: relative;
  width: 100%;
  height: 100px;
}
}

div.permission_username{
  position: absolute;
  font-size: 25px;
  top: 7px;
  left: 18px;
}

button.normal_button.remove_user_fro
m_permission{
  position: absolute;
  right: 10px;
  border-radius: 15px;
  height: 30px;
  width: 80px;
  padding: 5px;
  font-size: 10px;
  top: 10px;
}

div.permission_explaination{
  font-size: 15px;
  margin: 20px;
}
```

Serial Communication Wifi Configuration Desktop Program

device_configurator.py (base functions)

```
import serial
import time

SERIAL_CONFIGURATION = {
    "baudrate": 115200,
    "bytesize": serial.EIGHTBITS,
    "stopbits": serial.STOPBITS_ONE,
    "timeout": 0.5
}

CARRIAGE_RETURN = "\r\n" # This needs to be sent after each command to execute the
command.

CONNECTION_WAIT_TIME = 12

def get_response(recieved, line_breaks=1):
        """
        Command is echoed back, then the return value on next line
        then the prompt on the next and so splitting on the carrige return
        and selecting the second value gives us what we want.
        """

        if CARRIAGE_RETURN in recieved:
            recieved = recieved.split(CARRIAGE_RETURN)[line_breaks]

            return recieved

        else:
            return ""


def get_iot_device_ports():
    ports = []

    """
    Windows offers up ports COM[1-256] as serial communication ports so
    what we're doing here is trying to connect to each one, and if we do,
    we are sending the ping() command which is programmed into my devices
    which simply returns "Okay", if it does so, we know that it is a device we
    want to program.
    """

    for i in range(1, 257):
```

```
        potenial_port = "COM" + str(i)

        try:
            with serial.Serial(**SERIAL_CONFIGURATION, port=potenial_port) as conn:
                conn.write("\x03".encode()) # effectively a CTRL + C break the
server listening loop so we can talk to the device
                conn.write(("ping()" + CARRIAGE_RETURN).encode())

                recieved = conn.read(256).decode()

        except serial.SerialException:
            continue

        recieved = get_response(recieved, 2)

        if "Okay" in recieved:
            ports.append(potenial_port)

    return ports

def connect_to_network(ssid, password, port):
    with serial.Serial(**SERIAL_CONFIGURATION, port=port) as conn:
        conn.write(("connect_to_network('{}', '{}')".format(ssid, password) +
CARRIAGE_RETURN).encode())

        time.sleep(CONNECTION_WAIT_TIME)

        recieved = conn.read(256).decode()

    recieved = get_response(recieved)

    if "True" in recieved:
        return True
    elif "False" in recieved:
        return False


def get_ip_and_auth_key(port):
    with serial.Serial(**SERIAL_CONFIGURATION, port=port) as conn:
        conn.write(("get_ip_and_auth_key()" + CARRIAGE_RETURN).encode())

        recieved = conn.read(256).decode()

    recieved = get_response(recieved)

    return recieved.split(":")
```

device_configurator_gui.py

```
from device_configurator import *
from tkinter import *
from tkinter.filedialog import *
import threading

class ConverterGUI:
    def __init__(self, master):
        self.master = master
        master.title("IoT Device Configurator")

        self.device_port = StringVar()

        self.refresh_devices_button = Button(self.master, text="Refresh Device
Ports", command=self.start_refresh_connected_iot_devices)
        self.refresh_devices_button.grid(row=0, column=2, padx=10, pady=10)

        self.device_selection_dropdown = OptionMenu(self.master, self.device_port,
"None")
        self.device_selection_dropdown.grid(row=0, column=1, padx=10, pady=10)

        self.device_selection_label = Label(self.master, text="Select a port: ",
bg="white")
        self.device_selection_label.grid(row=0, column=0, sticky=W, padx=10,
pady=10)

        self.ssid = StringVar()

        self.ssid_label = Label(self.master, text="WiFi SSID: ", bg="white")
        self.ssid_label.grid(row=1, column=0, sticky=W, padx=10, pady=10)

        self.ssid = Entry(self.master, textvariable=self.ssid)
        self.ssid.grid(row=1, column=1, columnspan=2, padx=10, pady=10)

        self.wifi_password_label = Label(self.master, text="WiFi Password: ",
bg="white")
        self.wifi_password_label.grid(row=2, column=0, sticky=W, padx=10, pady=10)

        self.wifi_password = StringVar()

        self.wifi_password = Entry(self.master, textvariable=self.wifi_password)
```

```
        self.wifi_password.grid(row=2, column=1, columnspan=2, padx=10, pady=10)

        self.status_text = StringVar()

        self.status = Label(master, textvariable=self.status_text, bg="gray", bd=5)
        self.status_text.set("Status: Idle")
        self.status.grid(row=3, column=0, columnspan=2, padx=10, pady=10)

        self.apply_button = Button(self.master, text="Apply WiFi Config",
command=self.start_apply_configuration_to_device)
        self.apply_button.grid(row=3, column=2, padx=10, pady=10)

        self.ip_label_label = Label(self.master, text="Device IP: ", bg="white")
        self.ip_label_label.grid(row=4, column=0, sticky=W, padx=10, pady=10)

        self.ip = StringVar()

        self.ip_label = Label(self.master, textvariable=self.ip, bg="white")
        self.ip_label.grid(row=4, column=1, columnspan=2, sticky=W, padx=10,
pady=10)

        self.auth_key_label_label = Label(self.master, text="Auth Key: ",
bg="white")
        self.auth_key_label_label.grid(row=5, column=0, sticky=W, padx=10, pady=10)

        self.auth_key = StringVar()

        self.auth_key_label = Label(self.master, textvariable=self.auth_key,
bg="white")
        self.auth_key_label.grid(row=5, column=1, columnspan=2, sticky=W, padx=10,
pady=10)

        self.refresh_details_button = Button(self.master, text="Refresh Selected
Device's IP and Auth Key", command=self.start_refresh_details)
        self.refresh_details_button.grid(row=6, column=0, columnspan=3, padx=10,
pady=10)

        self.start_refresh_connected_iot_devices()

    def refresh_connected_iot_devices(self):
        try:
            self.devices_connected = get_iot_device_ports()

        except: # Errors caused by device being plugged in during search
            self.status_text.set("Status: Error scanning, try again")
            self.status.config(bg="red")
```

```
            return

        self.refresh_devices_connected_dropdown()

        self.status_text.set("Status: Refreshed")
        self.status.config(bg="green")


    def start_refresh_connected_iot_devices(self):
        threading.Thread(target=self.refresh_connected_iot_devices).start()
        self.status_text.set("Status: Looking for devices")
        self.status.config(bg="yellow")

    def refresh_devices_connected_dropdown(self):
        self.device_selection_dropdown.destroy()

        if not self.devices_connected:
            self.devices_connected = ["None"]

        self.device_selection_dropdown = OptionMenu(self.master, self.device_port,
*self.devices_connected)
        self.device_selection_dropdown.grid(row=0, column=1, padx=10, pady=10)

    def refresh_details(self):
        if not self.device_port.get():
            self.status_text.set("Status: No device selected")
            self.status.config(bg="red")
            return

        try:
            ip, auth_key = get_ip_and_auth_key(self.device_port.get())

        except:
            self.status_text.set("Status: Error talking to device")
            self.status.config(bg="red")
            return

        if ip == "0.0.0.0":
            self.ip.set("Not Connected")
        else:
            self.ip.set(ip)

        self.auth_key.set(auth_key)

        self.status_text.set("Status: Refreshed")
        self.status.config(bg="green")
```

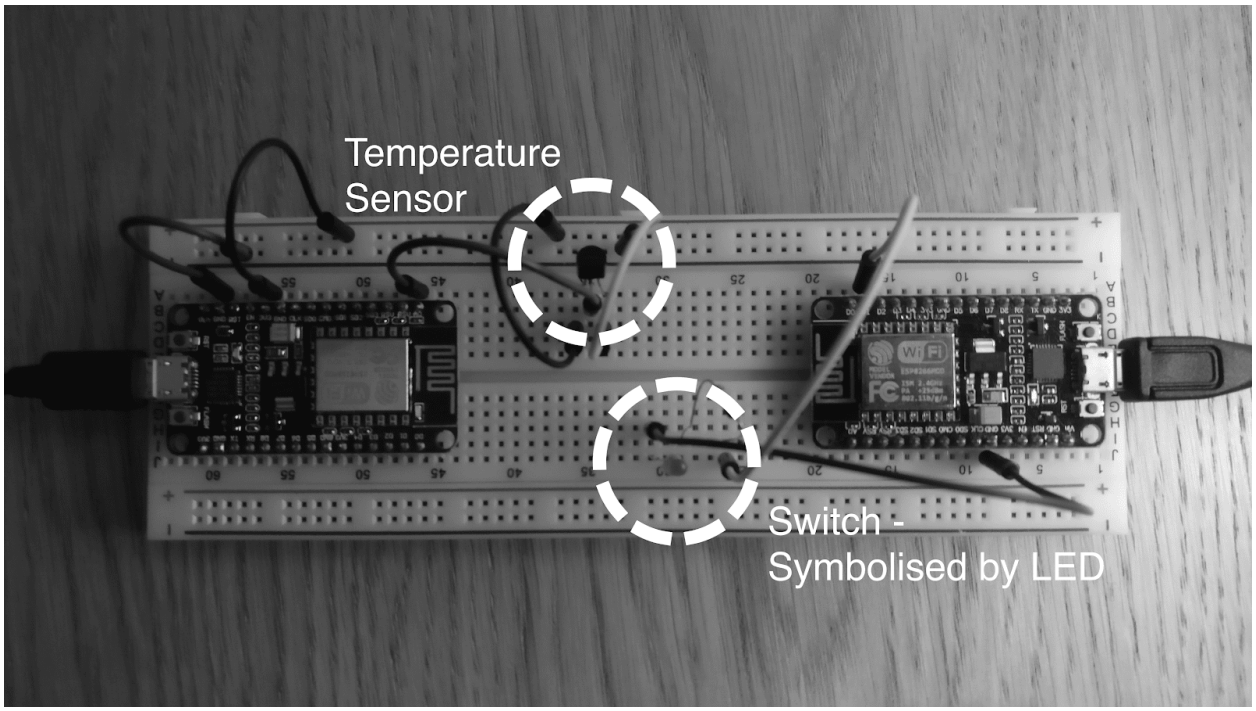
```
    def start_refresh_details(self):
        threading.Thread(target=self.refresh_details).start()
        self.status_text.set("Status: Refreshing Details")
        self.status.config(bg="yellow")

    def apply_configuration_to_device(self):
        ssid = self.ssid.get()
        password = self.wifi_password.get()
        port = self.device_port.get()

        if not(ssid and password and port):
            self.status_text.set("Status: Parameters not set")
            self.status.config(bg="red")
            return

        try:
            is_connected = connect_to_network(ssid, password, port)

        except:
            self.status_text.set("Status: Error applying")
            self.status.config(bg="red")
            return

        if is_connected:
            self.status_text.set("Status: Connected")
            self.status.config(bg="green")

        elif not is_connected:
            self.status_text.set("Status: Incorrect SSID or Password")
            self.status.config(bg="red")

        elif not is_connected:
            self.status_text.set("Status: Error applying")
            self.status.config(bg="red")


    def start_apply_configuration_to_device(self):
        threading.Thread(target=self.apply_configuration_to_device).start()
        self.status_text.set("Status: Applying to device")
        self.status.config(bg="yellow")

root = Tk()
gui = ConverterGUI(root)
root.geometry("355x320")
```

```
root.configure(background="white")
root.resizable(0, 0)

icon_directory = __file__.split("/")
icon_directory[-1] = "logo.ico"
icon_directory = "/".join(icon_directory)
root.iconbitmap(icon_directory)

root.mainloop()
```

Hardware

Being a computer science/ programming project rather than an electronics project, I have focused more on the software side of the project; however, these devices still function very well.

See these annotated images of the devices on an electronics breadboard:



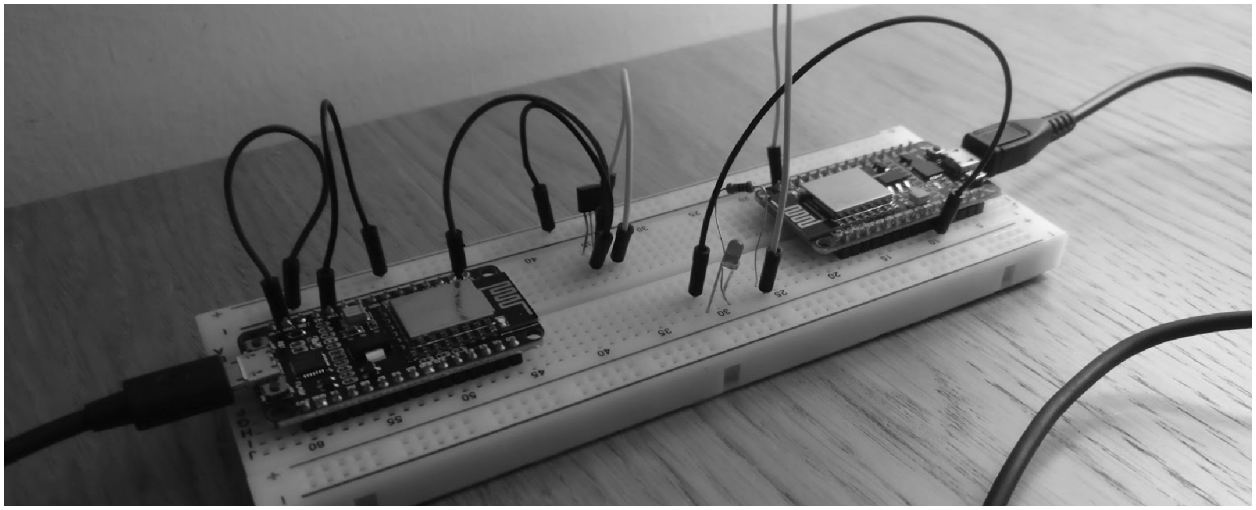
The device on the left hand side of the breadboard is the thermometer, it pulls data via a general purpose input output (GPIO) pin connected to a temperature sensing module which is the higher of the circled components.

The thermometer's lone readings tend to be a bit chaotic, however if, for example, 500 readings are taken and a mean of those 500 is calculated then it becomes very good. I have tested this against the thermostat currently in our home and the results match up very well.

The switch (on the right of the diagram) works as a proof-of-concept, it can currently switch on and off an LED - the lower of the circled components. This is simply to represent it outputting it's signal to a component such as a relay which is a component much like a transistor in that it opens and closes a switch based on input. A relay can switch on and off

mains power and would be what I would use if I were to implement my control system into a fully fledged electronical system.

The devices can currently be powered via a small USB plug, or battery bank, they draw very little power.



Once configured using the serial communication WiFi configurator, power can be lost reconnected and they still start up again, remembering which network they were connected to and will jump straight back into listening for socket requests.

Testing

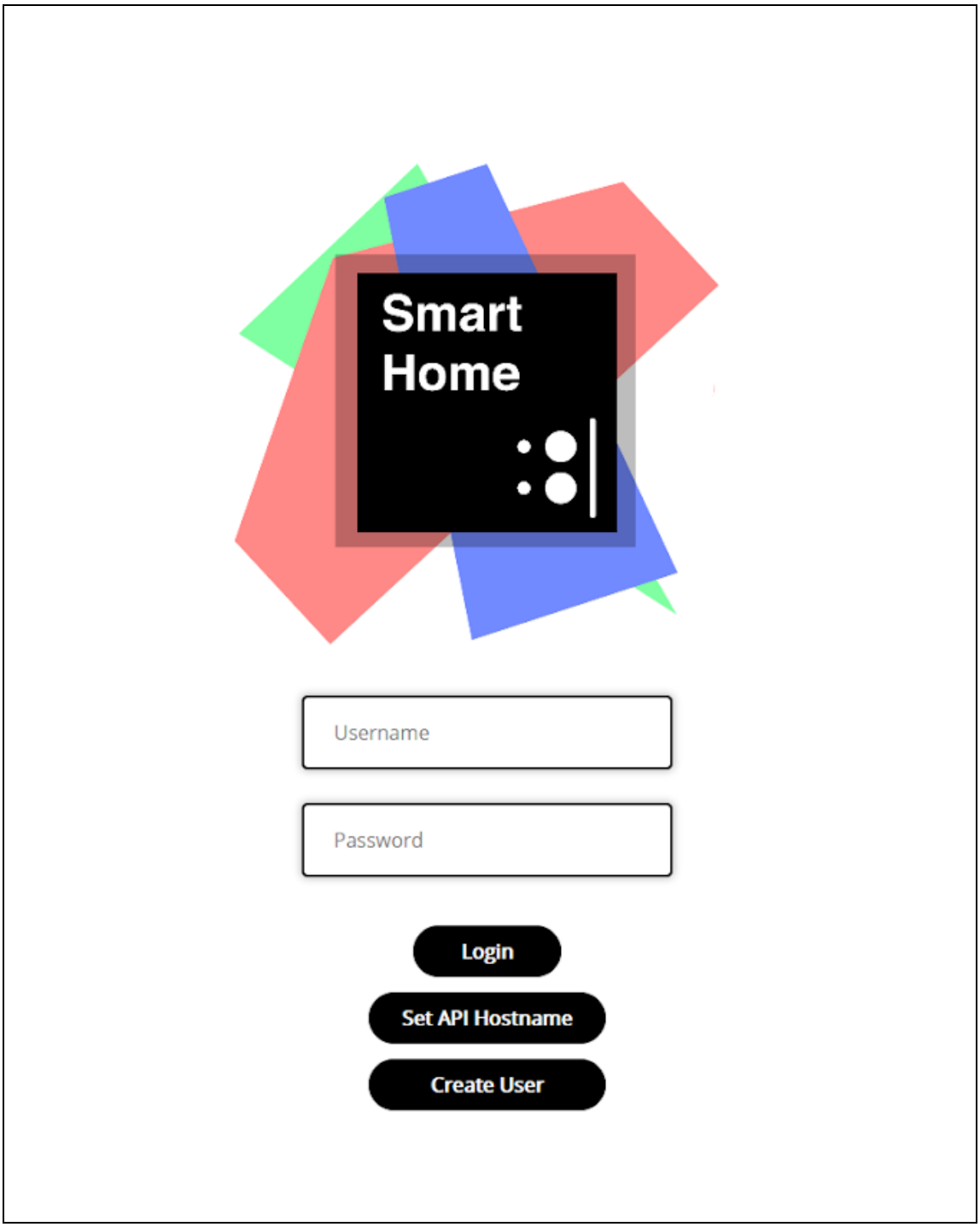
Normal System Walk Through

For this section of the testing, I will simply run through what a normal user would do without any intentionally erroneous data showing how and proving that the system functions.

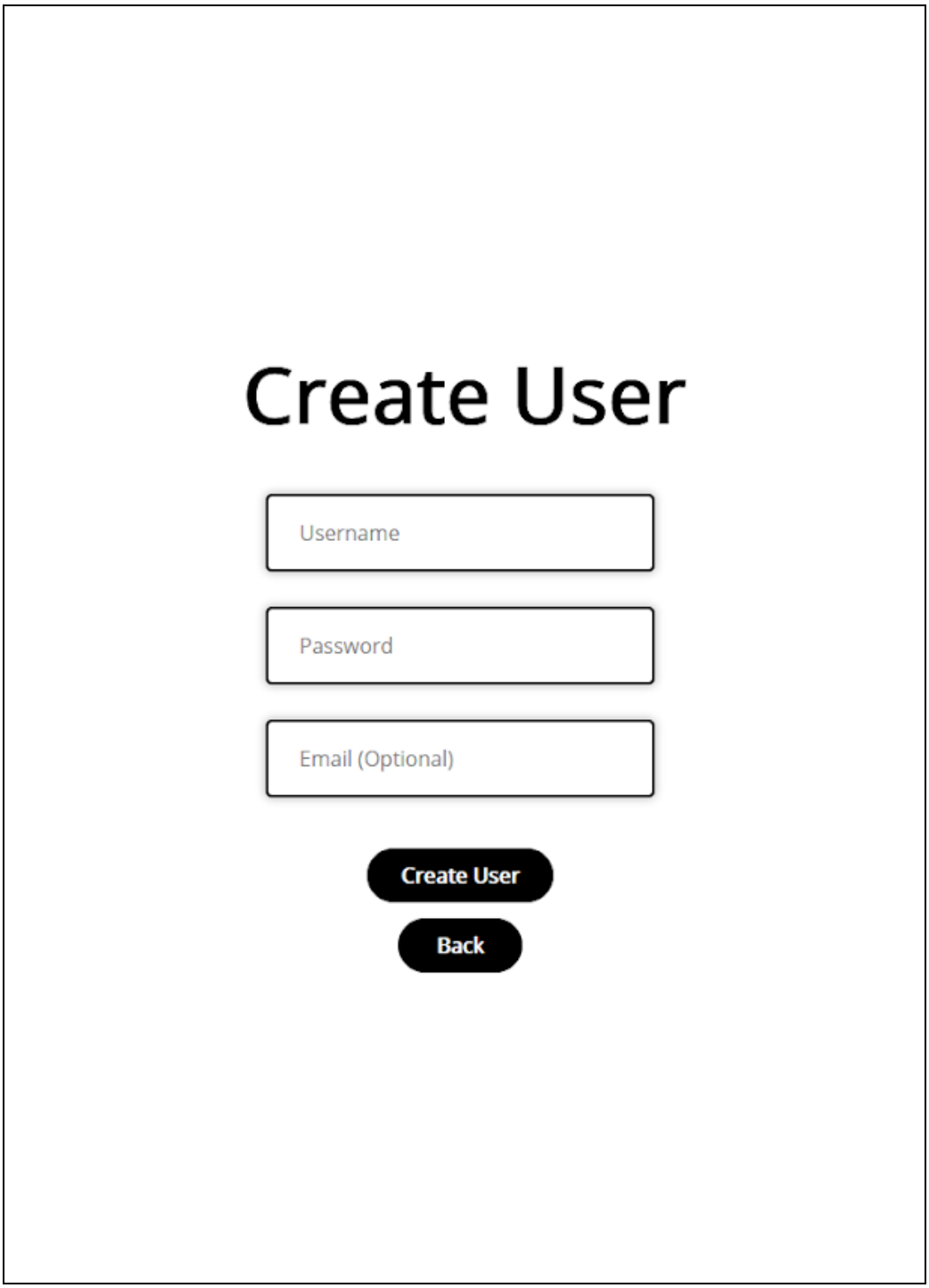
The interface for the main website during this test is from firefox on a 1920x1080 monitor on a PC; it has been cropped for visibility.

The serial communication WiFi configuration program is also running on that PC.

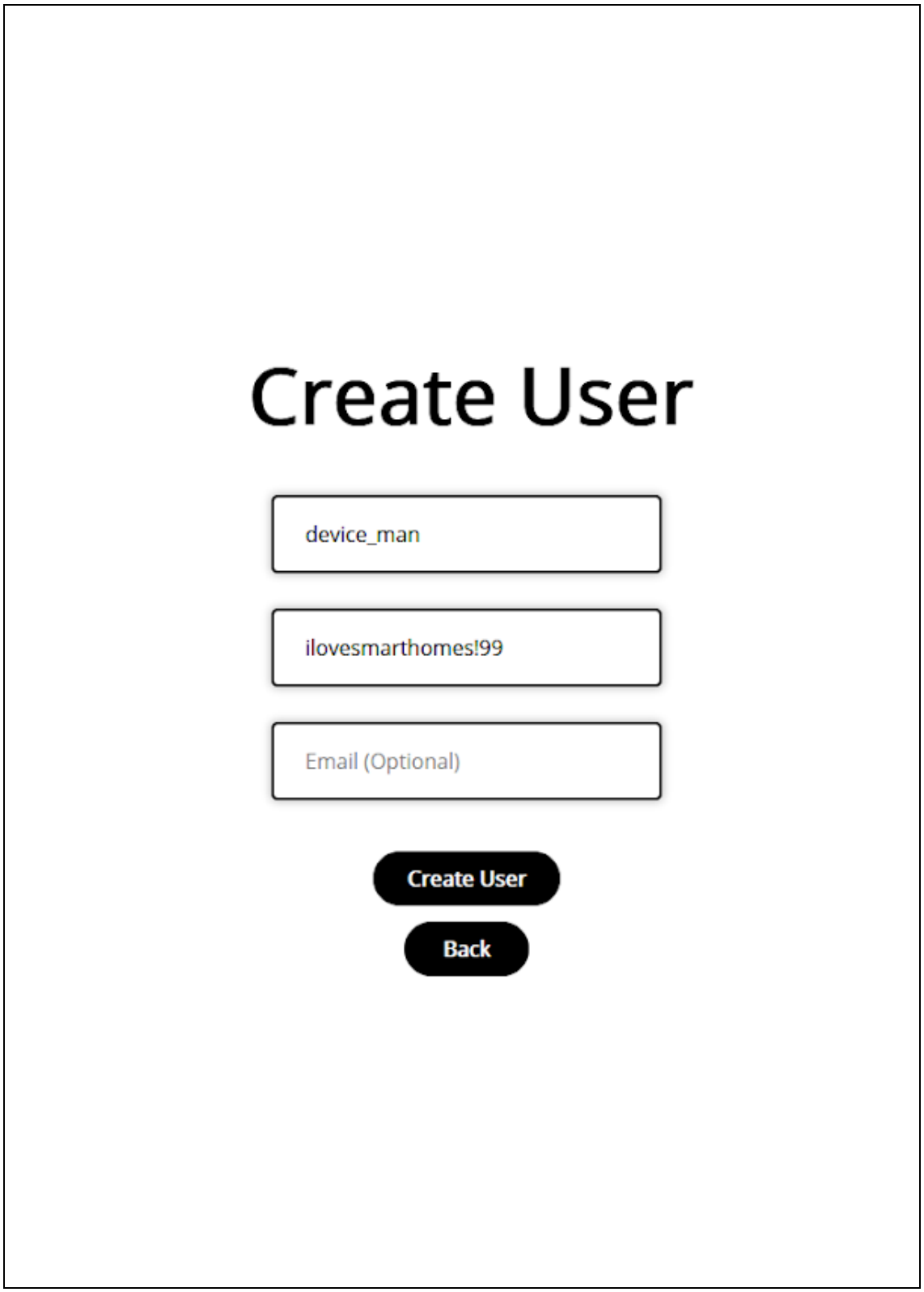
1. Opening up the website at the IP of my controller; it is on 192.168.141:8000.



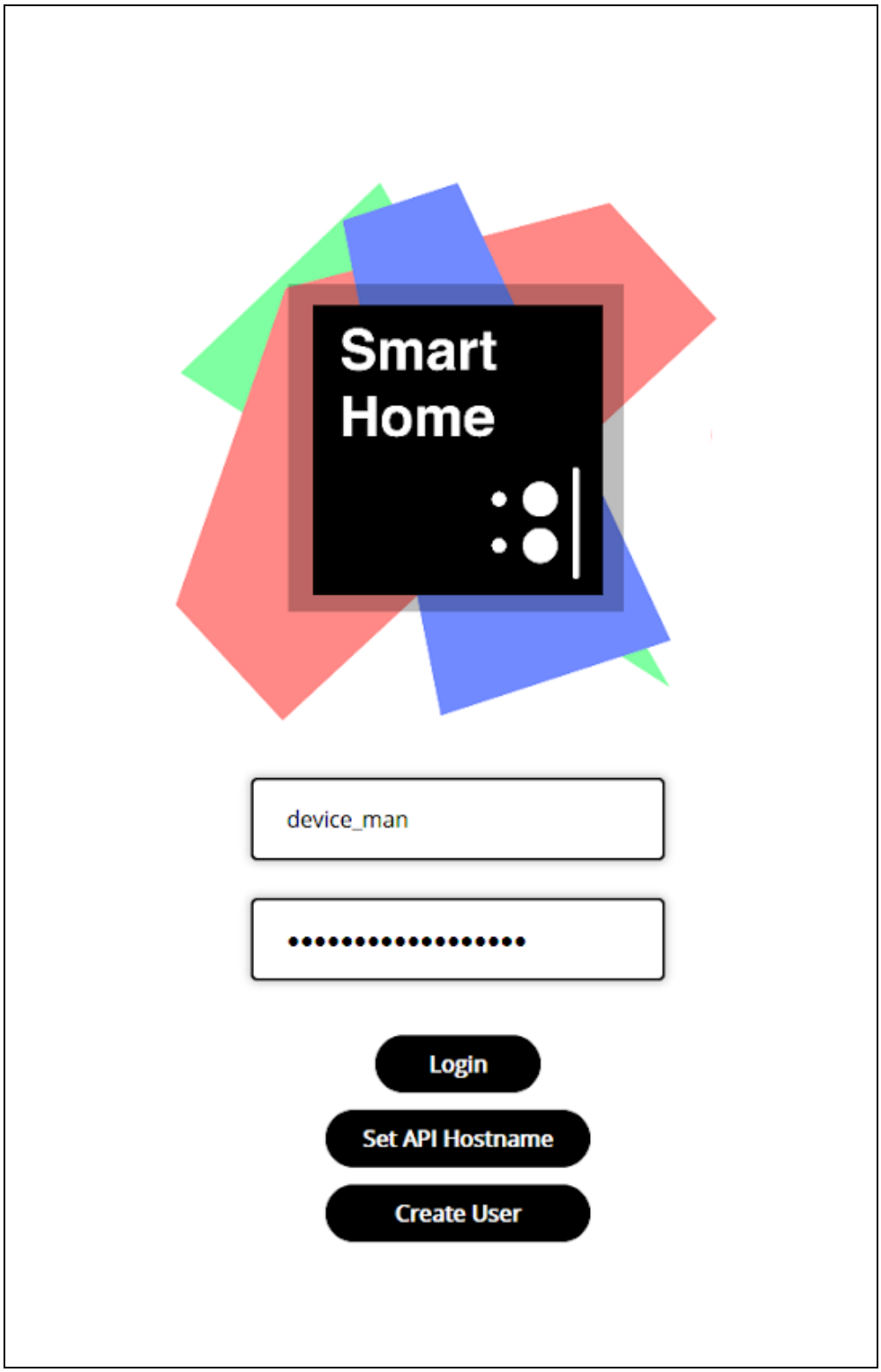
2. Opening create user.



3. Entering a username and password.

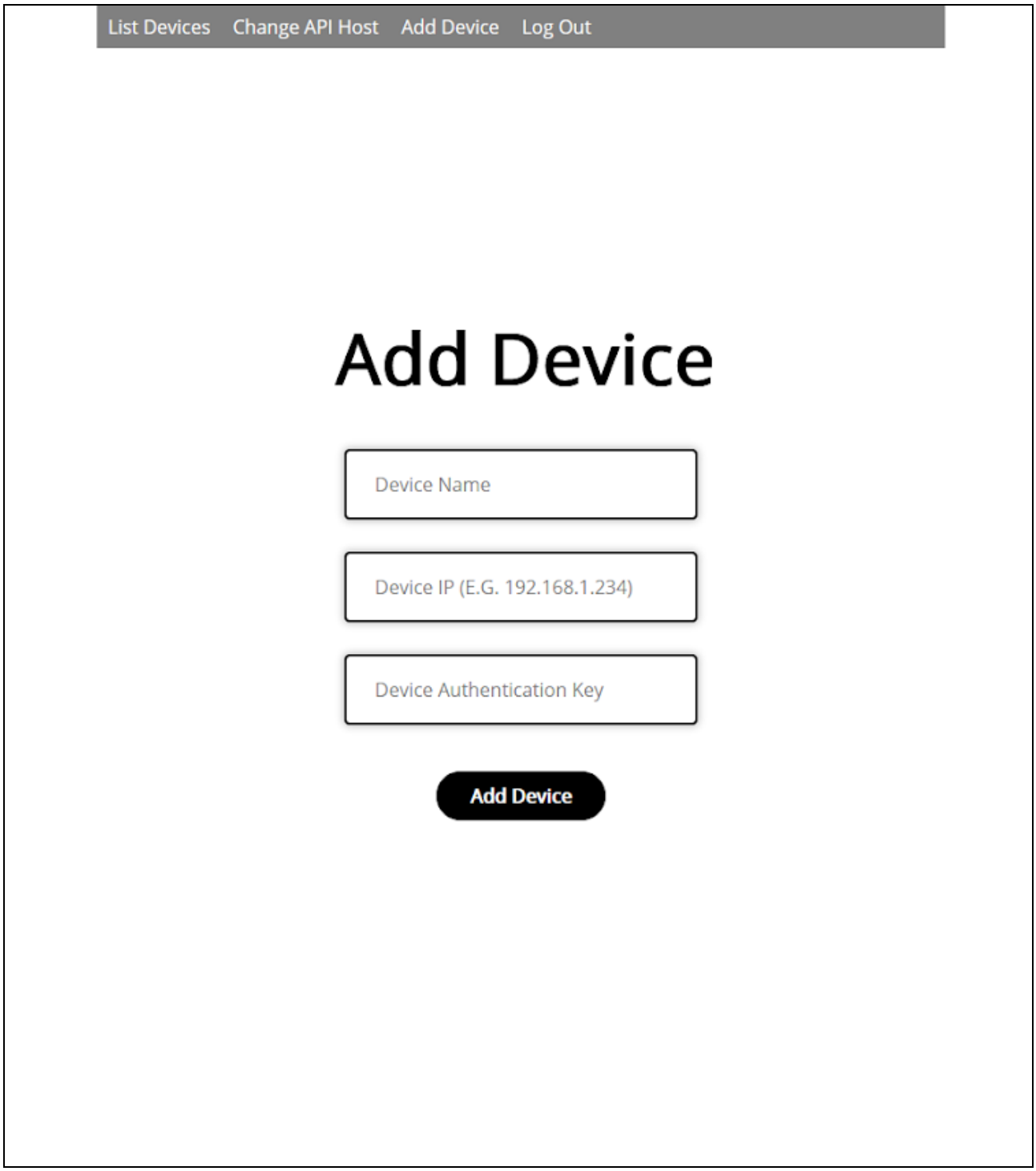


4. It creates the user successfully, we are redirected to the login page.

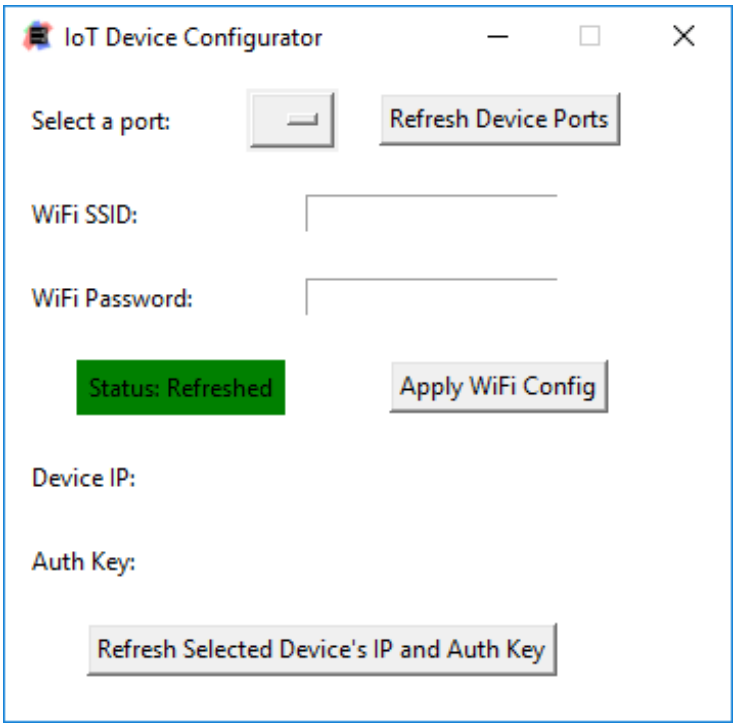


5. There are no devices on the system yet.

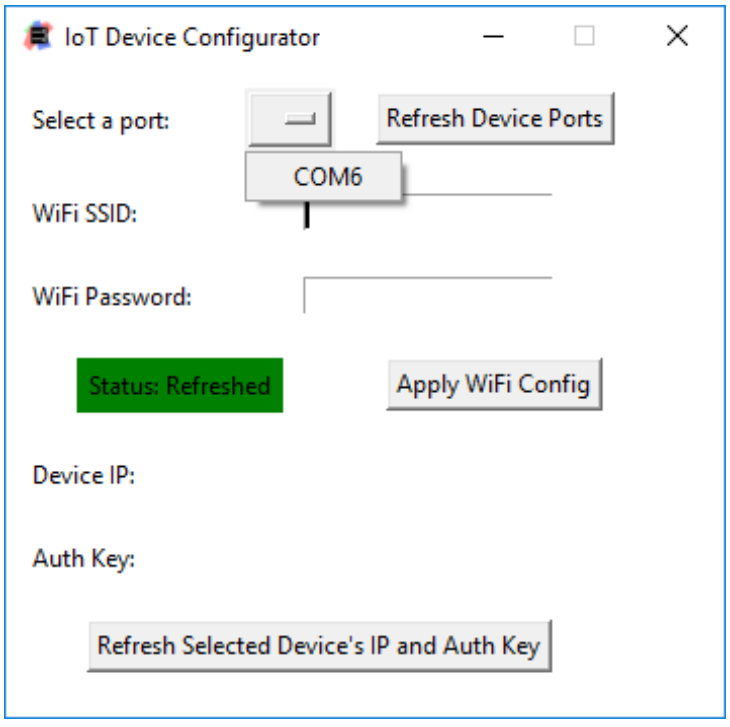
6. Opening add devices page.



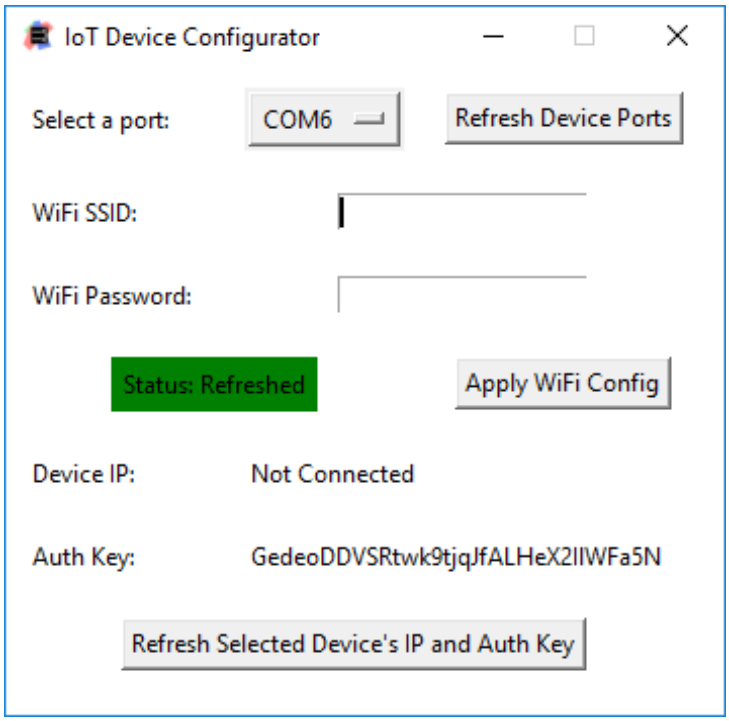
7. Open up the serial communication device configuration program, plug in device via USB.



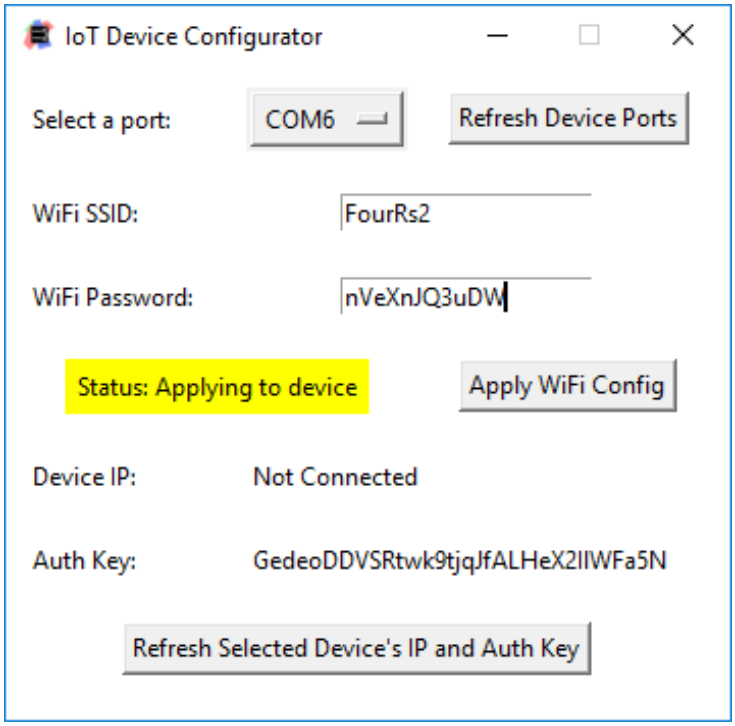
8. Scanning for devices, found a device.



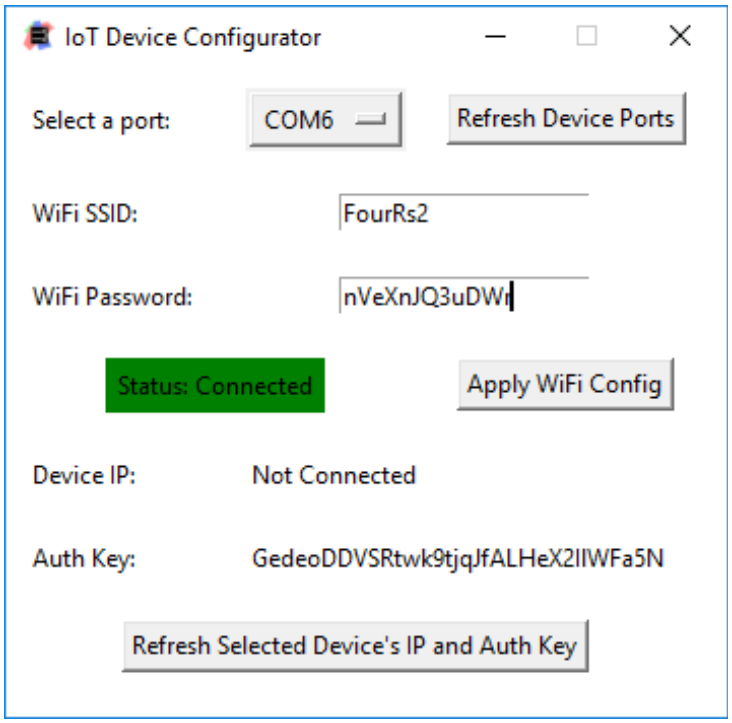
9. Select a device and refresh IP and Auth Key, we can see we are not connected.



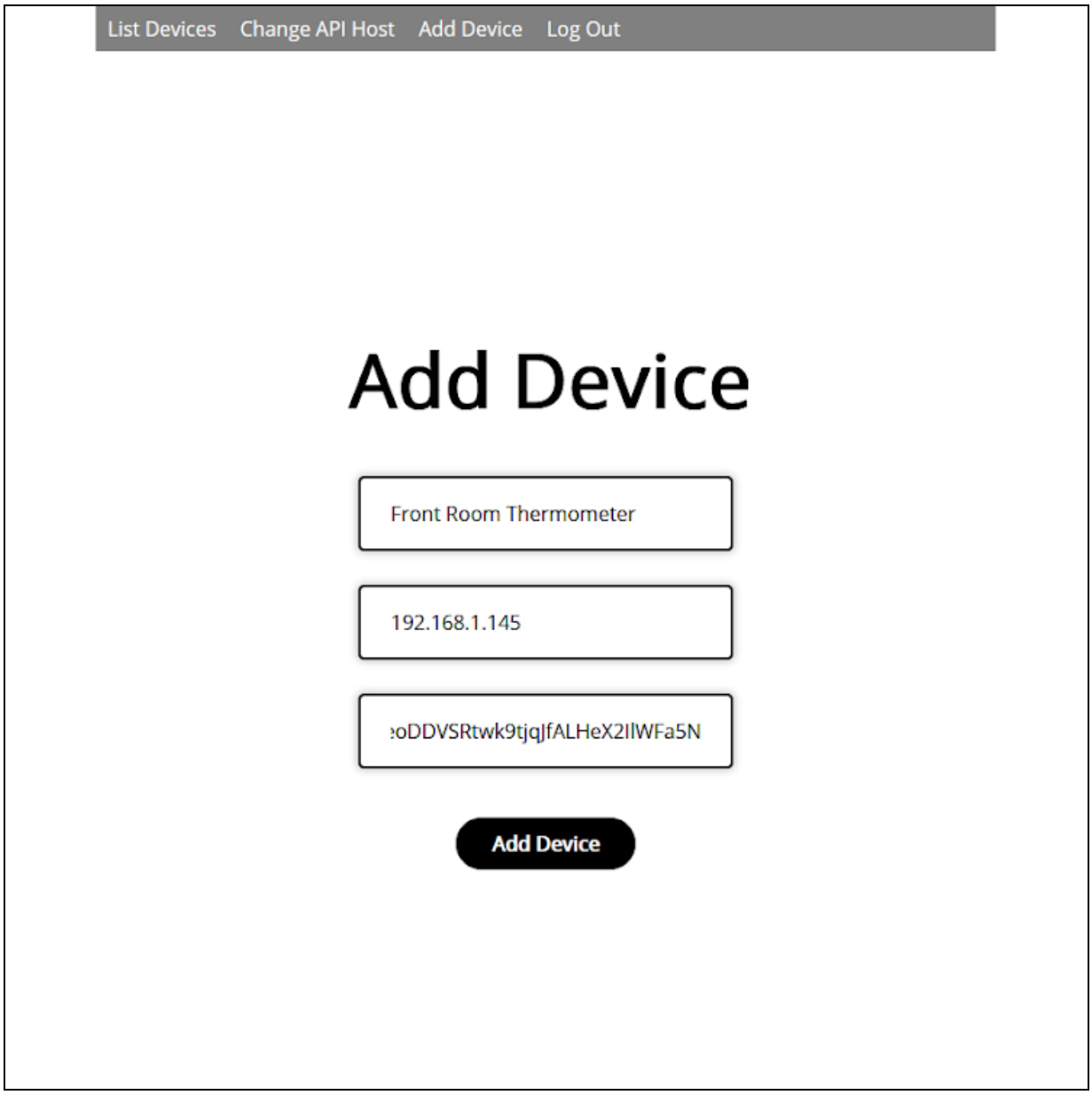
10. Enter WiFi details and attempt a connection.



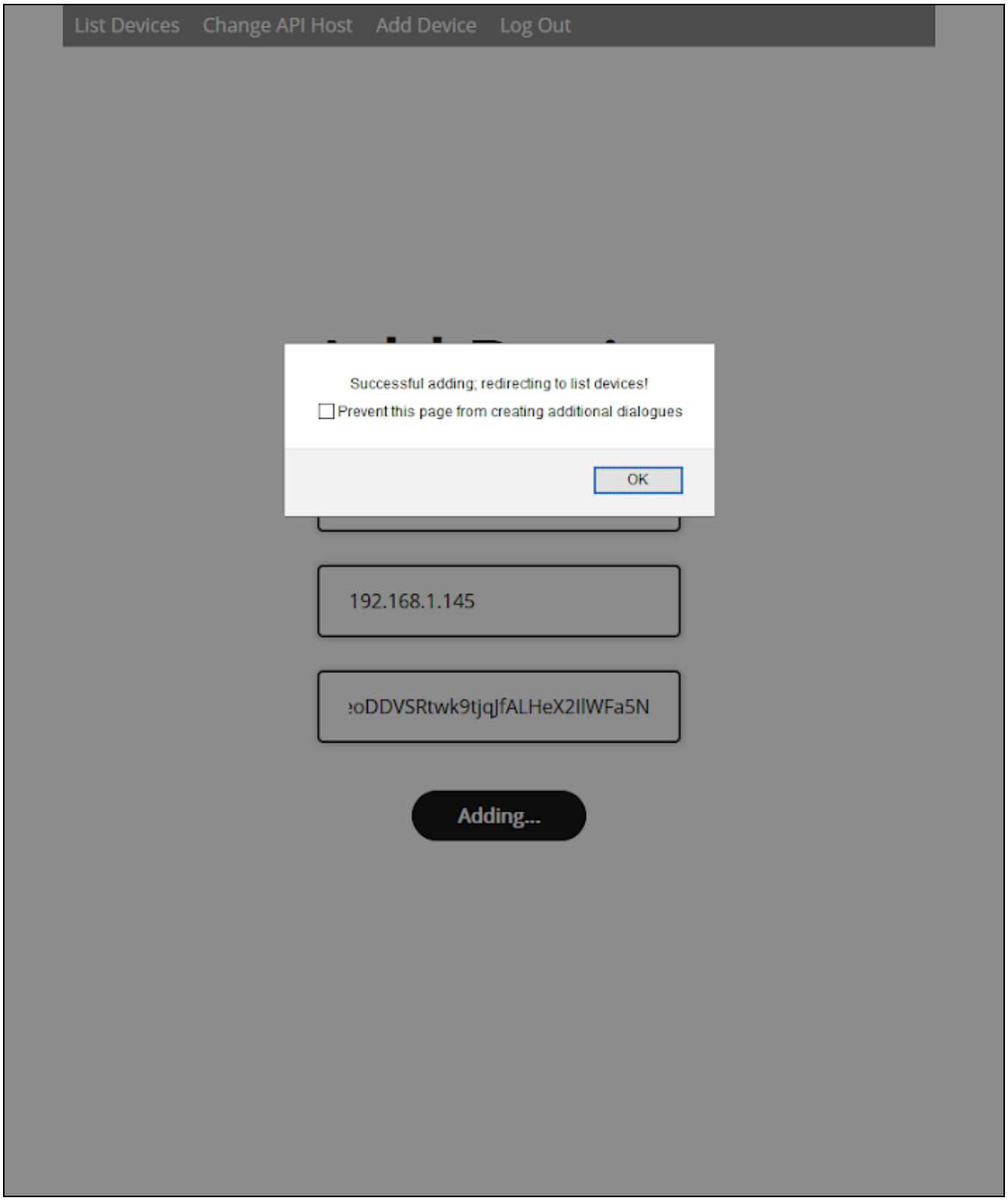
11. Correct details entered, connection succeeds and IP and auth key are refreshed again successfully this time.



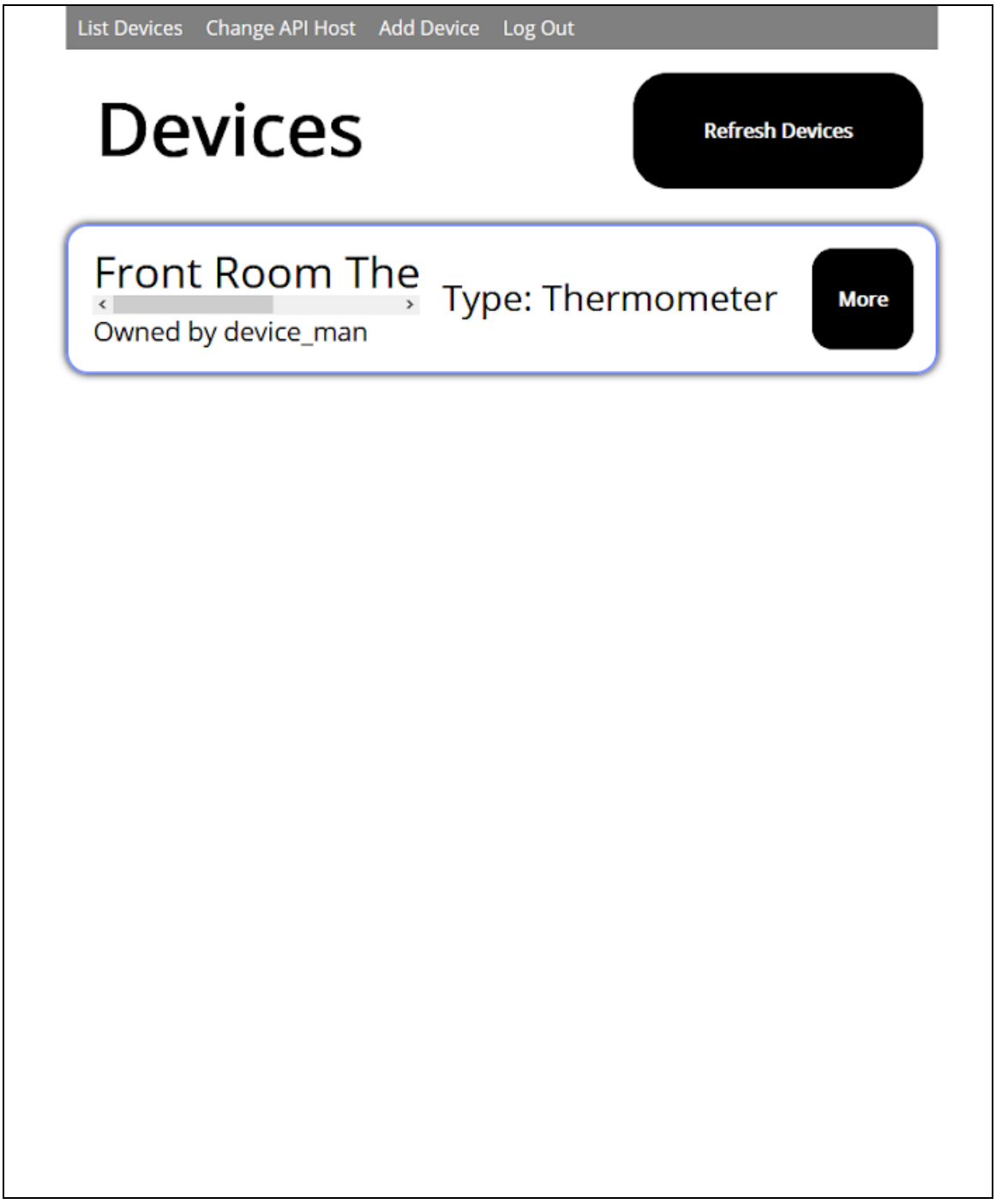
12. Returned back to add devices page and enter the details.



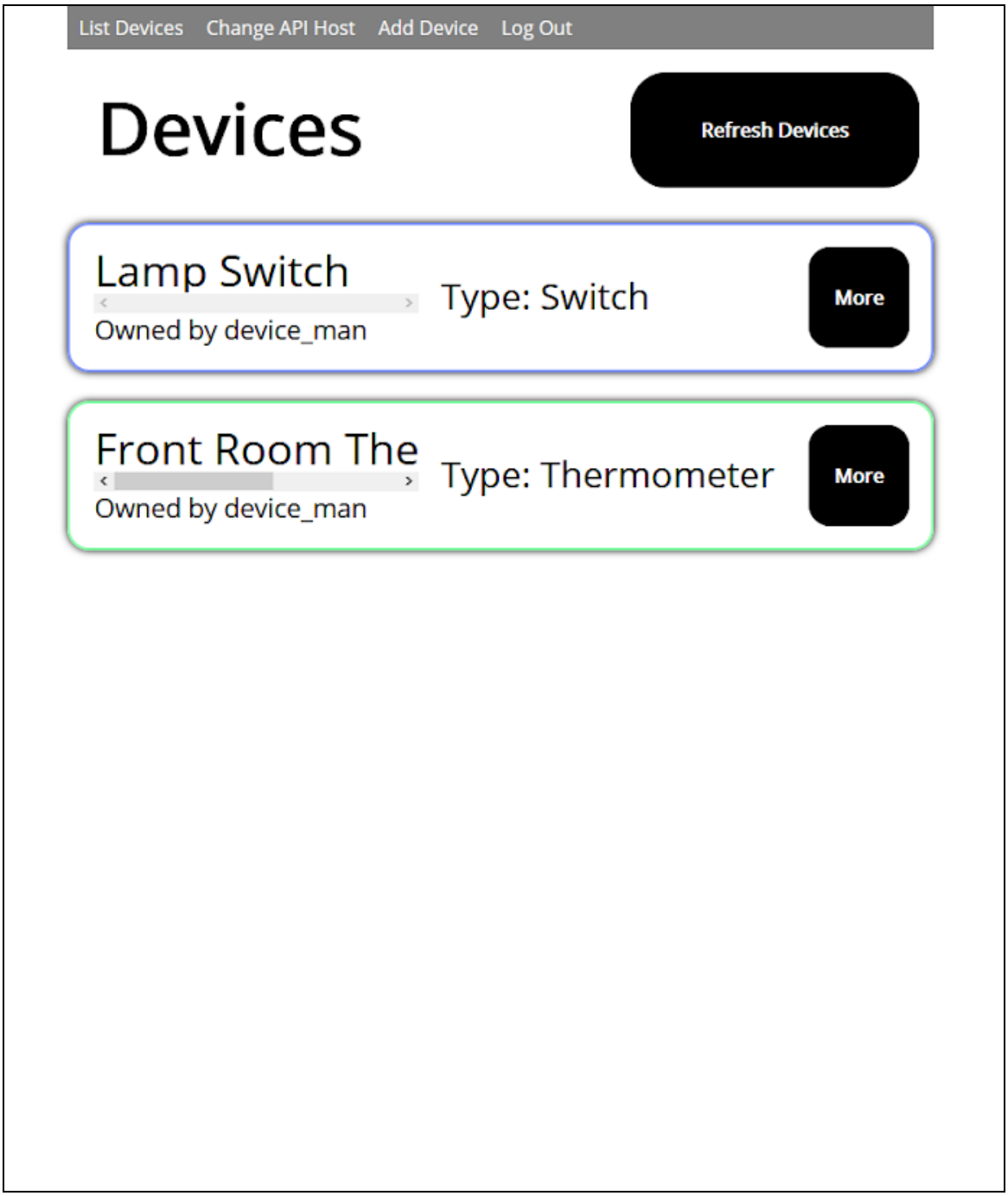
13. Device was successfully communicated with and hence was added.



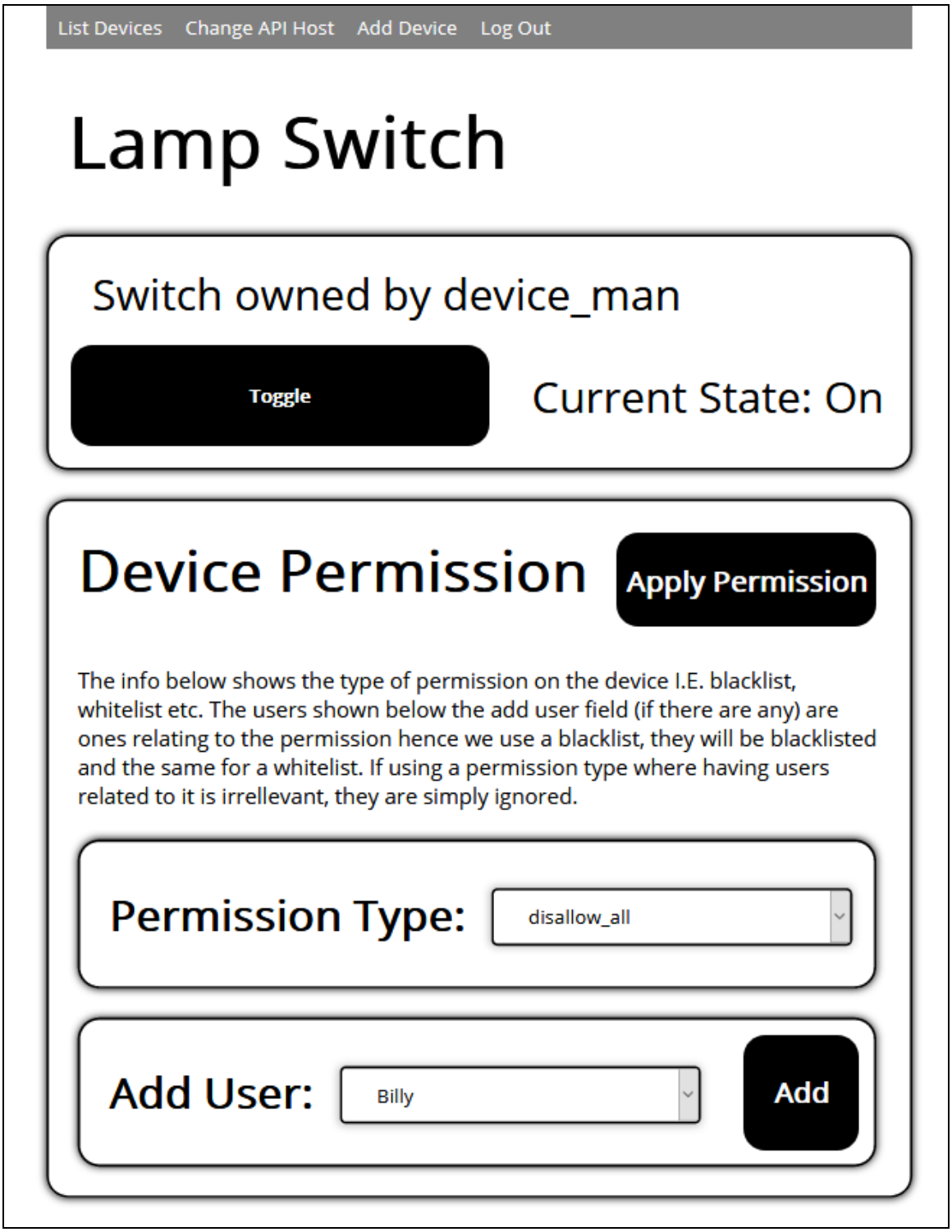
14. The device now appears on the list_devices page.



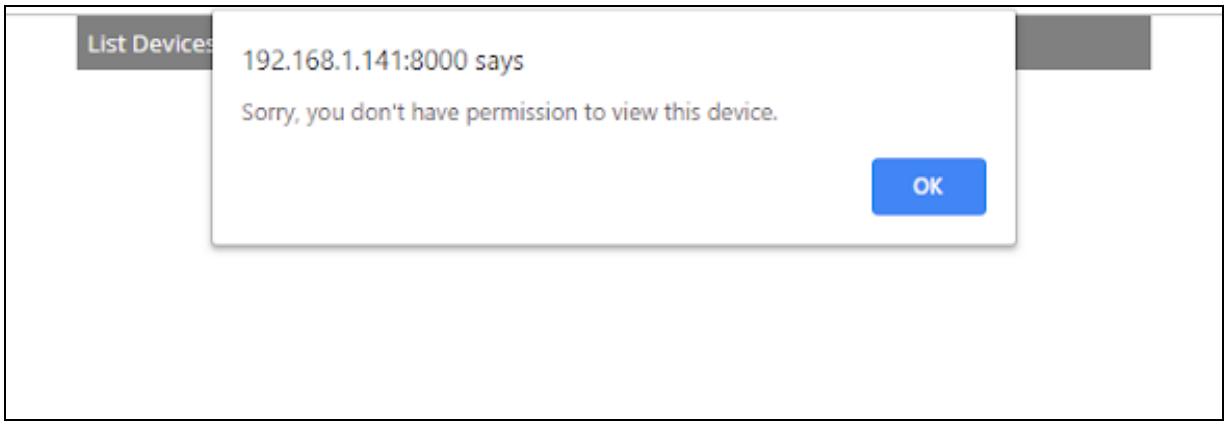
15. I repeated the process but with another device, a switch this time. A **bug was found** during this process, it will be explained after this section.



16. Clicking “More” on one of the devices (the switch) takes us to the /device page with the GET parameter of the deviceID which we clicked on.

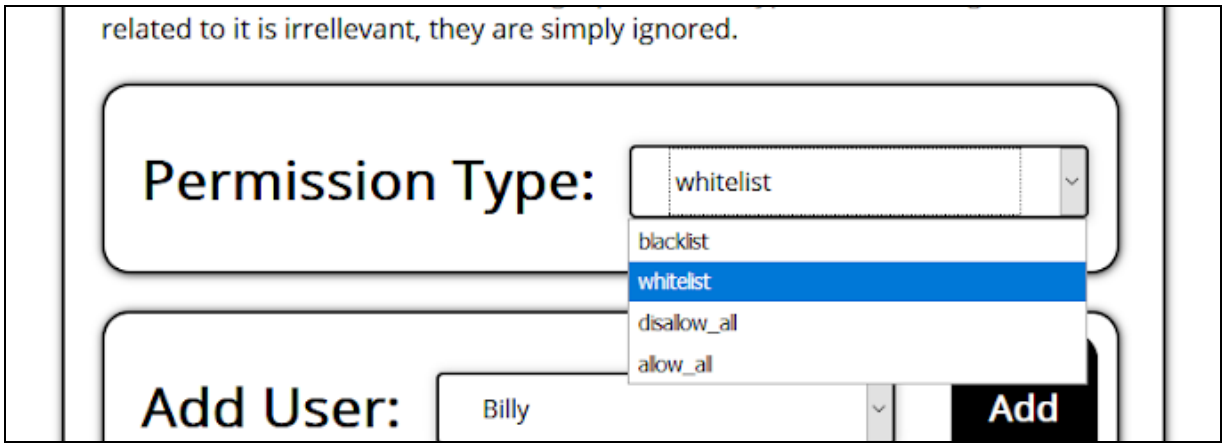


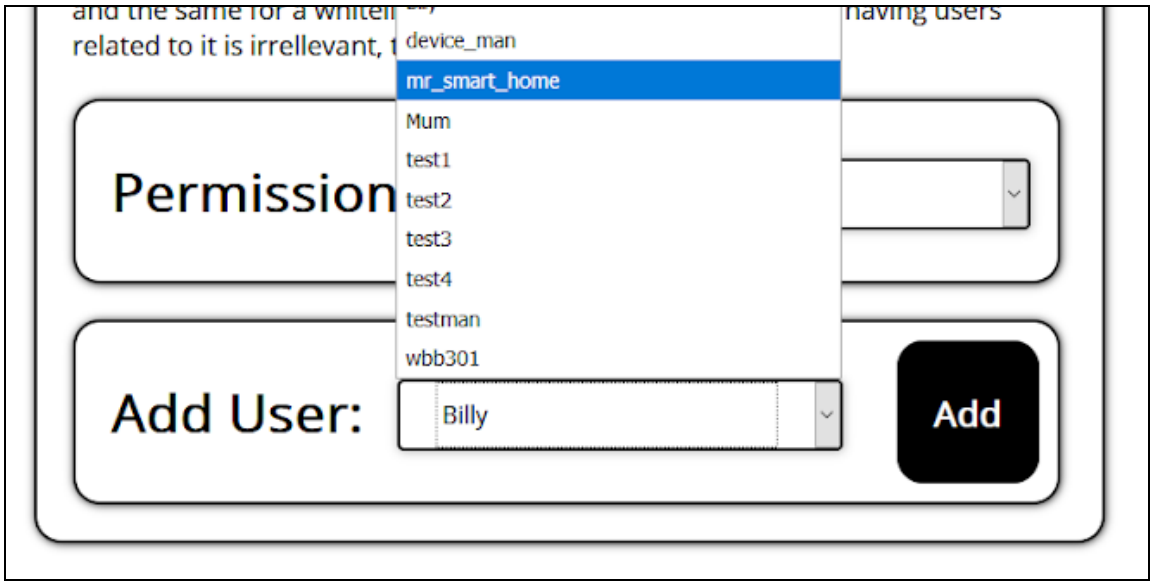
17. As we can see in the previous image, the permission type is currently disallow all. However as we are the owner we can still access it. See below when I try to access the device as a different user: “mr_smart_home”.

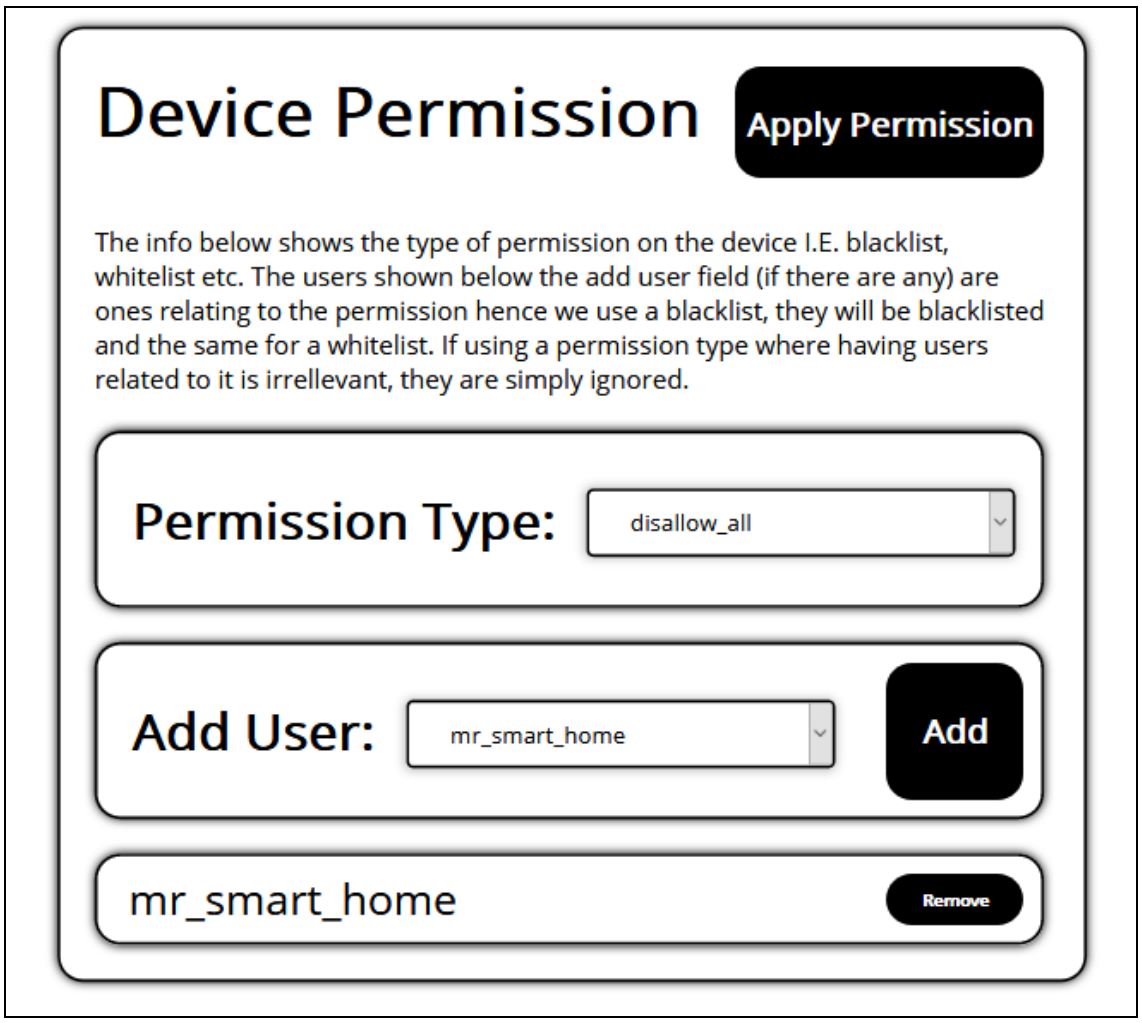


18. Next (back as the user “device_man”) I can set the permission type to whitelist and add “mr_smart_home” to it. The following images show this process.

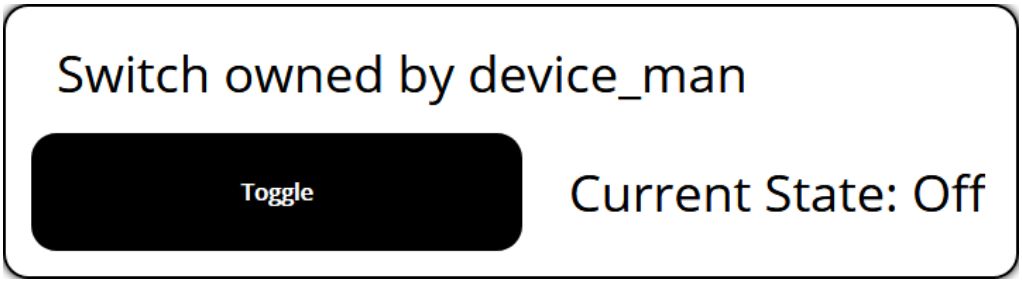
    Once the permission has been applied by clicking the top right button, “mr_smart_home” can now access this device.

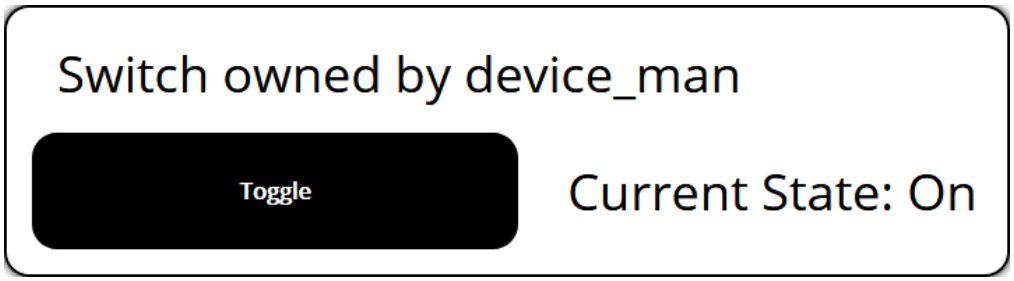


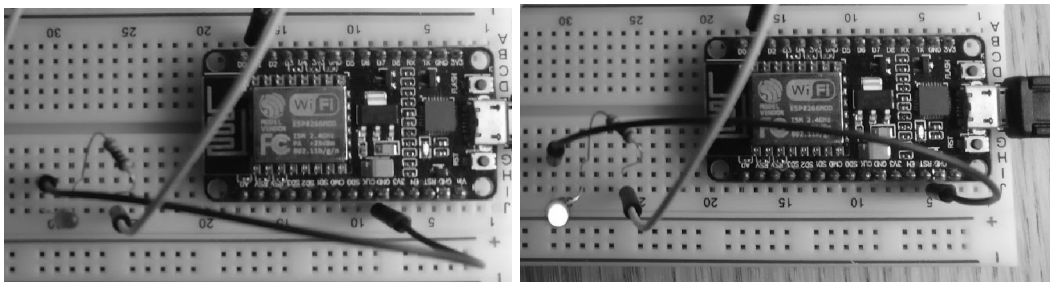

related to it is irrelevant,
having users
device_man
mr_smart_home
Mum
test1
test2
test3
test4
testman
wbb301
Permission
Add User:
Billy
Add


Device Permission
Apply Permission
The info below shows the type of permission on the device I.E. blacklist, whitelist etc. The users shown below the add user field (if there are any) are ones relating to the permission hence we use a blacklist, they will be blacklisted and the same for a whitelist. If using a permission type where having users related to it is irrelevant, they are simply ignored.
Permission Type:
disallow_all
Add User:
mr_smart_home
Add
mr_smart_home
Remove

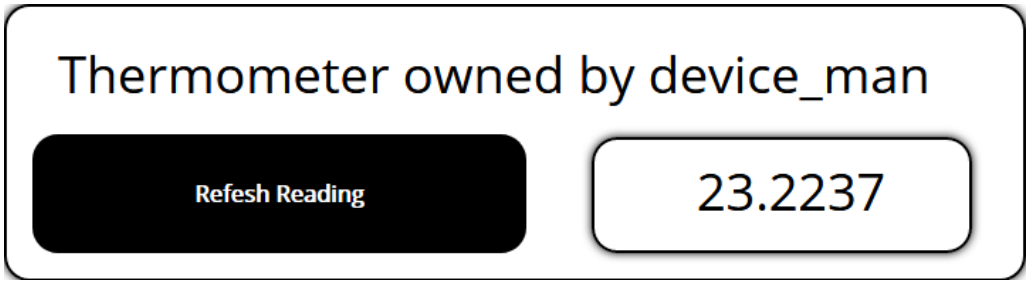
18. Finally, the toggle and refresh temperature buttons work well when pressed: the thermometer took accurate readings when compared with an existing thermometer and the switch toggled correctly.

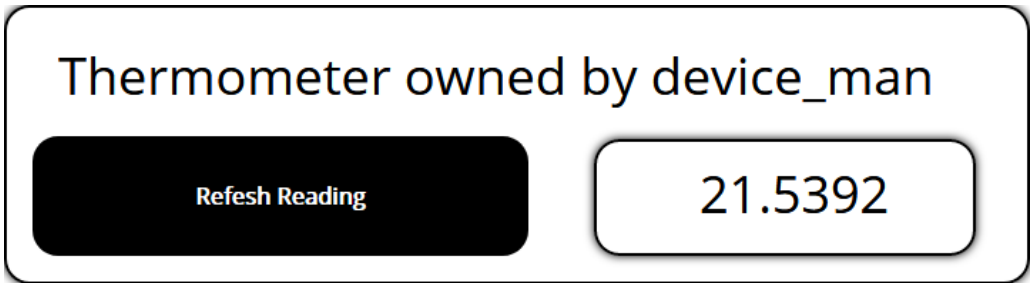






In hot area vs in cooler area.

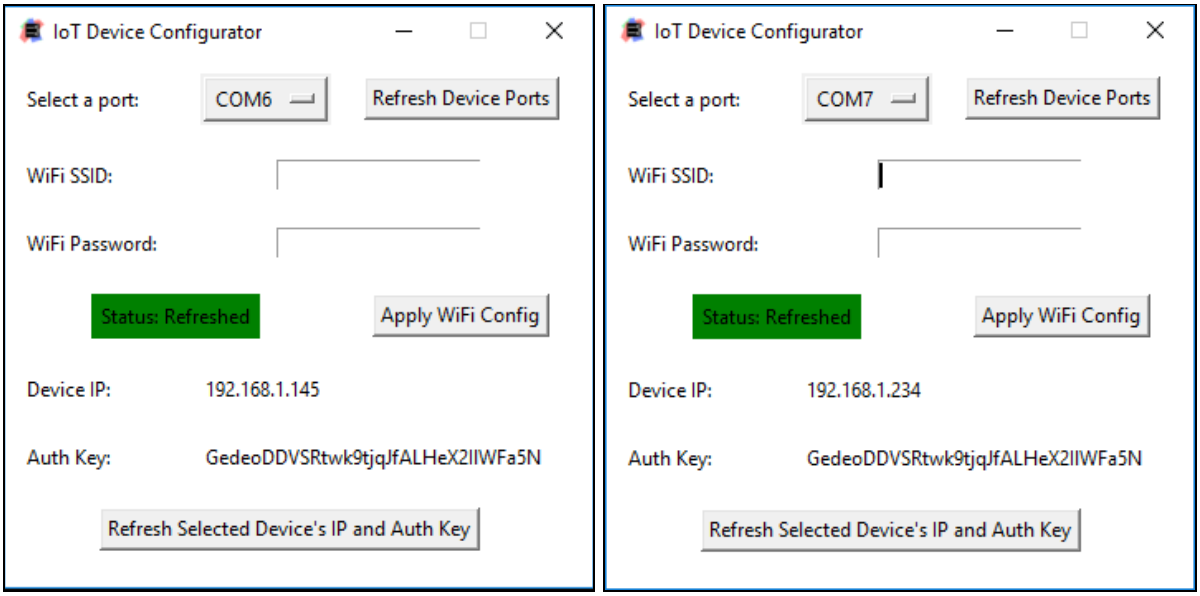




Bug Found In Walkthrough

The bug I found in walkthrough was when adding the second device; the authentication keys provided were the same.

The way that the keys are generated is by selecting a list of 32 random characters from a list of characters. However the ESP8266, being a simple programmable microcontroller, doesn't have the most sophisticated random function and I believe this could be the cause of this problem: the function used to generate a random auth key is using the same "random" values.



The identical "random" keys.

To test this hypothesis I - knowing that the devices have both generated the same amount of random numbers, generated 5 more; they were identical.

```
>>> randint(0, 100)
94
>>> randint(0, 100)
29
>>> randint(0, 100)
94
>>> randint(0, 100)
3
>>> randint(0, 100)
3
>>> sta_if = network.WLAN(network.STA_IF)
>>> sta_if.ifconfig()
('192.168.1.234', '255.255.255.0', '192.168.1.254', '192.168.1.254')
>>>
```

```
>>> randint(0, 100)
94
>>> randint(0, 100)
29
>>> randint(0, 100)
94
>>> randint(0, 100)
3
>>> randint(0, 100)
3
>>> sta_if = network.WLAN(network.STA_IF)
>>> sta_if.ifconfig()
('192.168.1.145', '255.255.255.0', '192.168.1.254', '192.168.1.254')
>>>
```

The identical “random” values; the lines of code below show that they are generated on 2 different devices as the IP (first in the tuple) is different.

To fix this I need to apply a seed to the random module before calling it. The seed must be reasonably random as well so a source of entropy is needed; I used a function from the time module[15], time.ticks_cpu().

This function returns how many CPU ticks past an arbitrary start point the system is, it rolls over periodically and could be used for timing.

I feed this in as a seed every time a number is generated now, and it works, no more identical keys.

Testing On Different Devices

Below are screenshots of the website on devices such as those my clients will be using; I have done this to ensure the mobile and easy access features in my analysis are fulfilled.

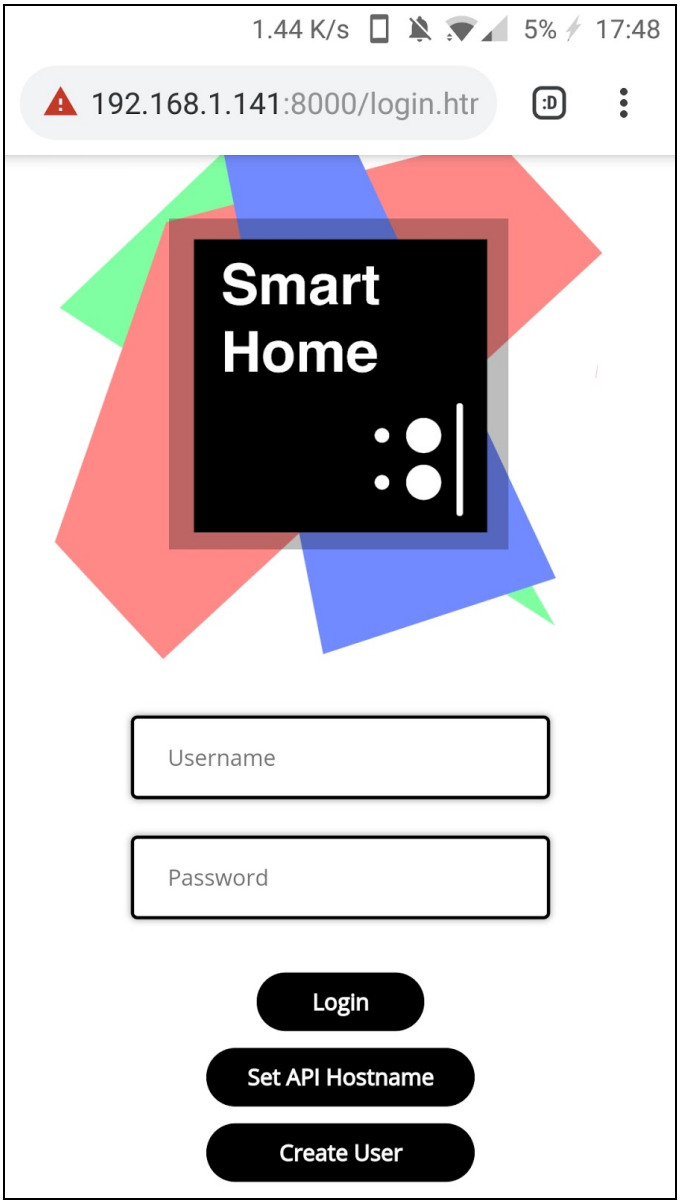
[15] https://docs.micropython.org/en/latest/library/utime.html

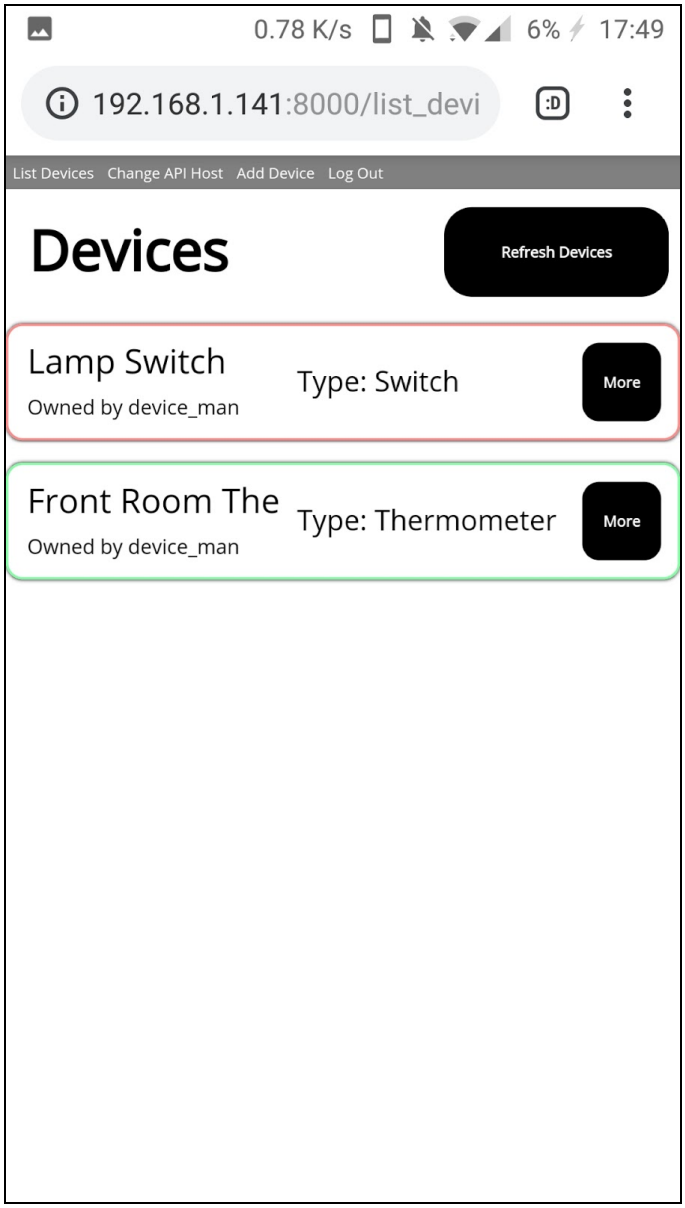
Android Smartphone

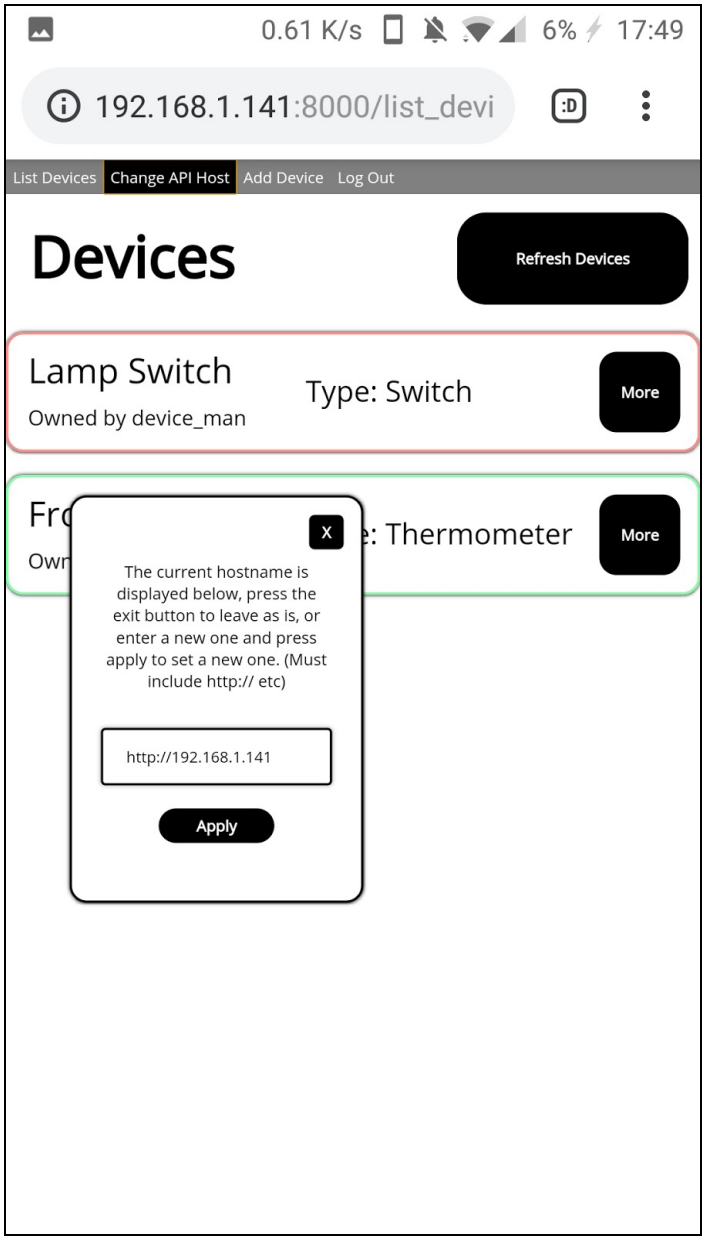
I found that, unfortunately, the UI was a bit squashed and warped on smaller devices such as smartphones.

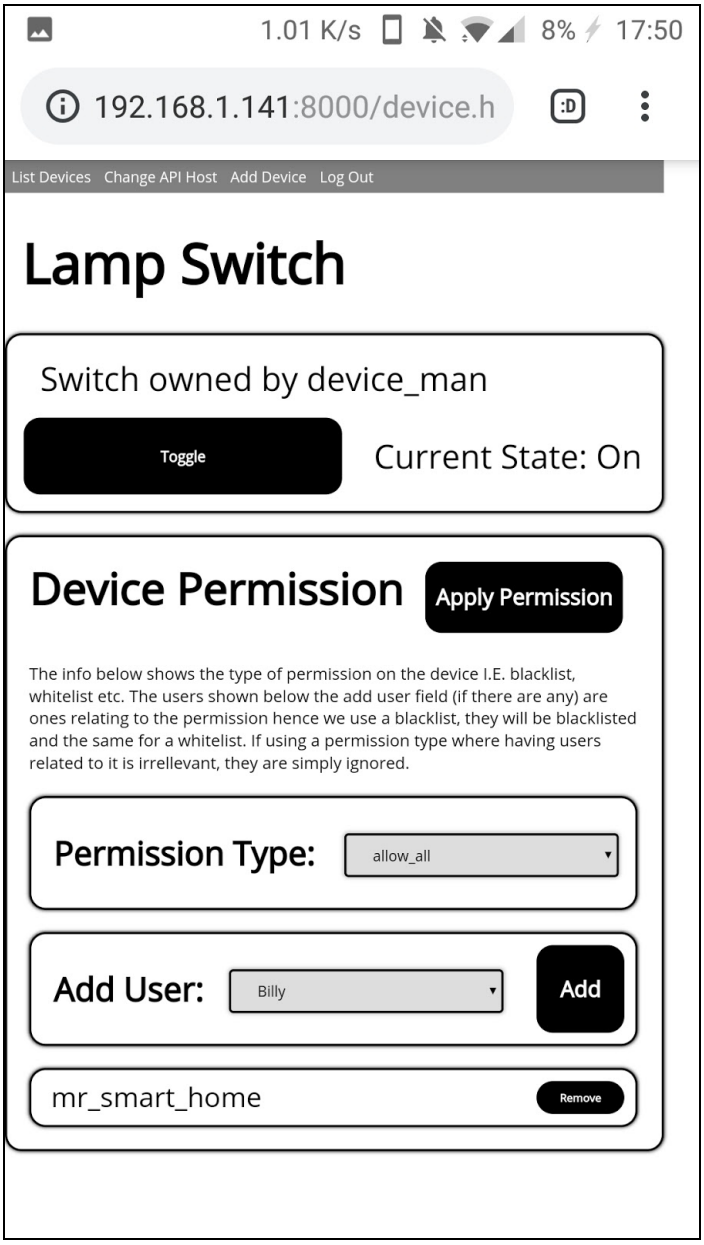
I think this is because I used fixed lengths for my central containers' CSS.

However, the system is still perfectly usable, all the functionality remains but with just a more of a cramped and warped GUI.



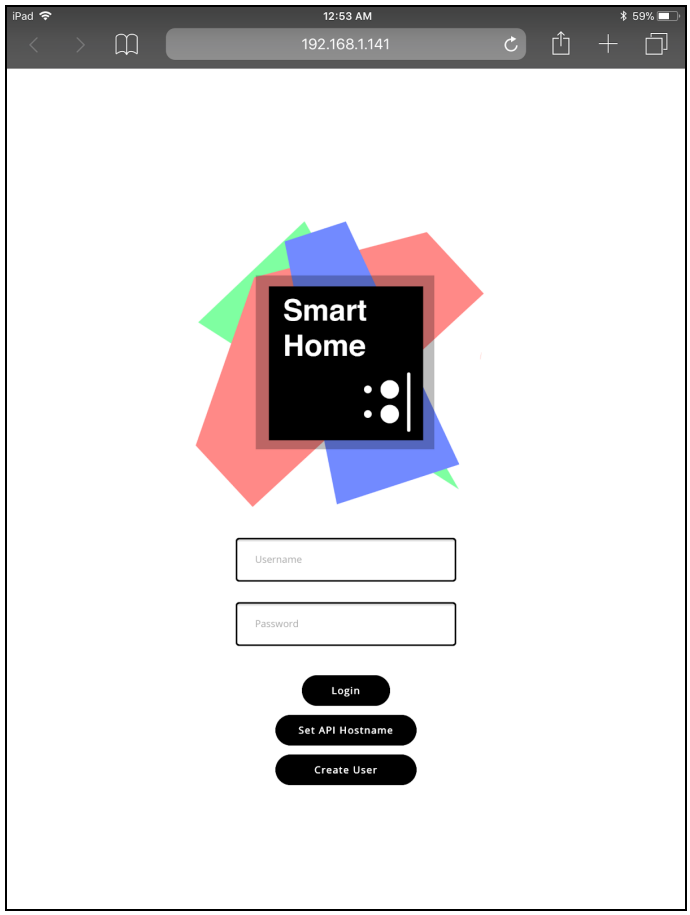


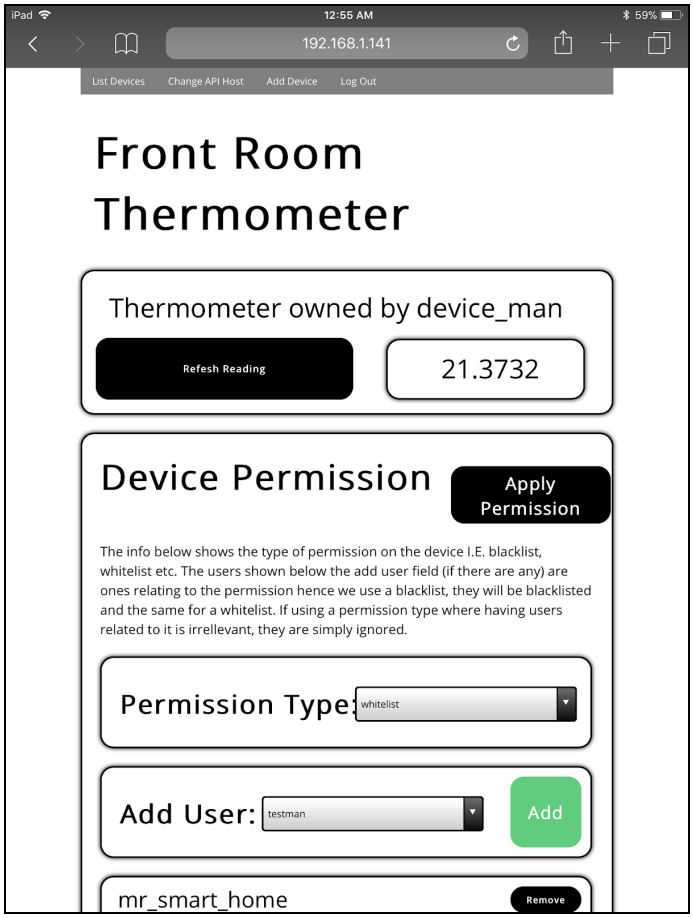


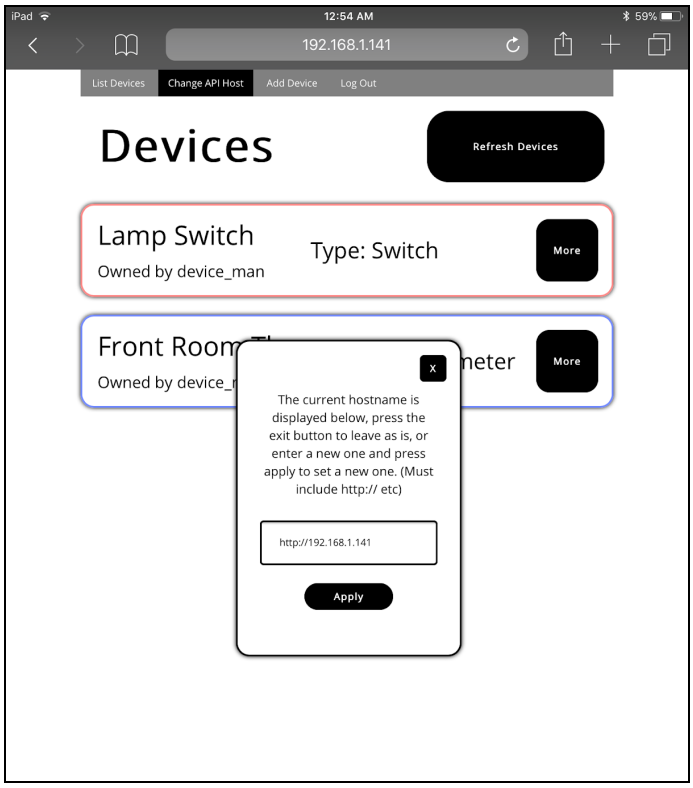


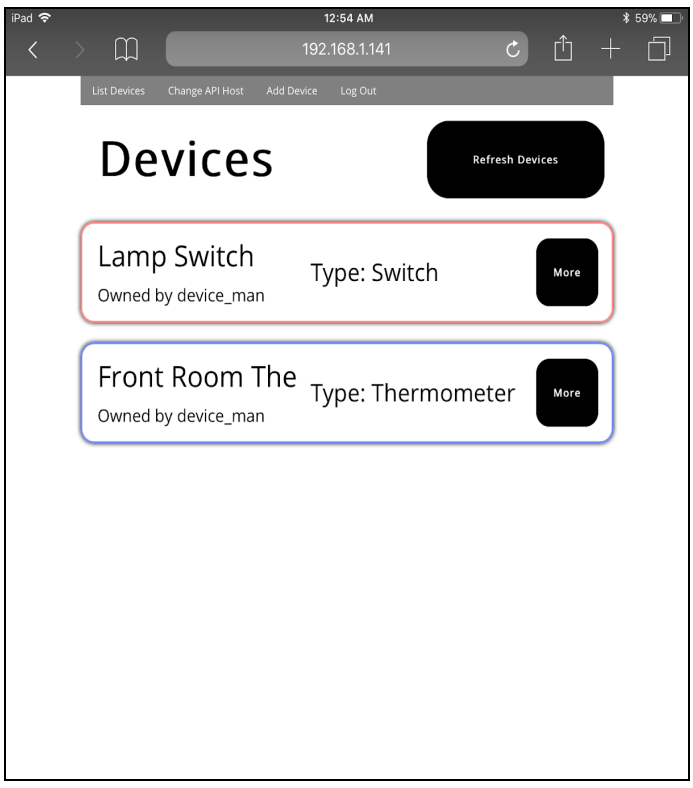
Apple iPad

Due to its large screen size, the GUI works beautifully on this device, apart from some tiny text rendering issues. This will be the device I recommend my clients use the system with.


iPad
12:53 AM
59%
192.168.1.141
Smart Home
Username
Password
Login
Set API Hostname
Create User


iPad
12:55 AM
59%
192.168.1.141
List Devices
Change API Host
Add Device
Log Out
Front Room Thermometer
Thermometer owned by device_man
Refesh Reading
21.3732
Device Permission
Apply Permission
The info below shows the type of permission on the device I.E. blacklist, whitelist etc. The users shown below the add user field (if there are any) are ones relating to the permission hence we use a blacklist, they will be blacklisted and the same for a whitelist. If using a permission type where having users related to it is irrellevant, they are simply ignored.
Permission Type:
whitelist
Add User:
testman
Add
mr_smart_home
Remove


iPad
12:54 AM
59%
192.168.1.141
List Devices
Change API Host
Add Device
Log Out
Devices
Refresh Devices
Lamp Switch
Owned by device_man
Type: Switch
More
Front Room
Owned by device_
More
X
The current hostname is displayed below, press the exit button to leave as is, or enter a new one and press apply to set a new one. (Must include http:// etc)
http://192.168.1.141
Apply


iPad
12:54 AM
59%
192.168.1.141
List Devices
Change API Host
Add Device
Log Out
Devices
Refresh Devices
Lamp Switch
Owned by device_man
Type: Switch
More
Front Room The
Owned by device_man
Type: Thermometer
More

Testing With Erroneous Data and Strange Circumstances

Note: all validation checks are done server side and so there is no way to slip around a validation check by simply calling an API function yourself with invalid data.

All RegEx strings have been validated using an online RegEx checker[16].

Response Colour Code:

- Okay
- Could be better
- Serious Problem

If an intended response is labeled “Improvement” it is not a bug or flaw in my current code but an improvement that could be made to it.

This section is simply to try and break the program by entering strange abnormal data to uncover errors.

#	Page	Action	Data	Response	Intended Response
1	/login	SQL Injection Attempt	In the users field: '); DROP TABLE user; --	Problem logging in: Parameters invalid: username	
2	All	Setting the API hostname erroneously.	http://google.com	Sorry, there was a problem connecting to the controller server, check the hostname is	

[16] https://regexr.com/

				correct.	
3	/device	User tries to look at a device without logging in by going straight to a /device?deviceID=x directory.	/device?deviceID=5	User is redirected back to login. Even if they weren't, they couldn't look as they have no session key.	
4	/add_device	User goes directly to this page without logging in and tried to add a device.	N/A	Problem creating device: Session key missing.	Improvement: Page should redirect to login automatically.
5	/device	Spamming the toggle button.	N/A	Error, code 12: Command call timed out. Then most of the request eventually all filter through and the system returns to normal.	Improvement: JS should stop the ability to send another request until the current one is done.
6	/device	Adding the owner of a device to the blacklist.	N/A	The controller API doesn't allow the owner to be added to the blacklist.	
7	/device	Spamming the apply permissions button.	N/A	Sorry, there was a problem connecting to the controller server, check	Improvement: JS should stop the ability to send another request until the current one is done.

				the hostname is correct. Takes a fair amount of time however all requests eventually filter through.	
8	/add_device	Trying to add a device on an IP that is not a device.	N/A	Problem creating device: Could not talk to the device, perhaps the IP is wrong.	
9	/device	Trying to access a device that isn't powered on.	N/A	Error, code 11: General problem connecting to device. Ensure the device is powered and connected to the network.	
10	/login	Not entering any values and logging in.	N/A	Problem logging in: Parameters invalid: password, username	

Evaluation

Presentation to and discussion with main client

Once I had finished the system, I walked my father (my main client) through the system and how it works discussed it with him as we went along.

My father, having a general interest in technology, has a basic knowledge of how a system such as this would function and so picked up quickly the more technological concepts such how to and why we need need to set the API's hostname. However, as he mentioned, typing in IP addresses may prove confusing to a completely tech unsavvy user; I agree.

This issue also arose in the adding a device section where you must type in the IP of the device to add: it could be confusing to a non technical user.

After further discussion I thought of the idea of encoding the device IP into QR code and allowing the user to upload a picture of said code to the website which we both agreed was a good solution.

This would also solve another minor issue he had: typing in the authentication key was tedious because it was too long. In retrospect 32 random characters is a pain for a user to type in with no mistakes; this could be encoded into a QR code in the configurator program.

Other user-friendliness problems included:

- Not telling the user why their password is invalid when rejecting it for being too weak (not matching the ReGex).
- No way to tell which user you are logged in as easily.

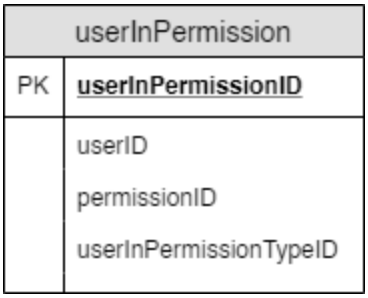
Theses would be easily fixable in a new iteration of this system.

A suggestion he had for the permissions system was that instead of my current system - in which when a user is added to a device's permission and gains full control over said device

and its permission - it would nice if you could add a user to a device's permission as a user or admin.

For example, a user would only be able to use the device I.E. switch it on and off whereas an admin could change permission as well as having the power of a user.

This could be implemented with a third field in the user In Permission linking table, it would go to this:



userInPermissionTypeID would store types of way a user could be in an permission E.G. user or admin or child.

Regardless of the criticism, my client still thought that this project was a success as it proved a system could be made which is cheaper than other alternatives, more private and very functional and was well within the bounds of my acceptable limitations.

Analysis of Objectives

Key: Well achieved, Reasonably Achieved, Failed to achieve

Objective	**Analysis**
Must have a controller API which controls IoT devices on the client's network.	Success.
Will store data about users and devices in an SQL database.	Success.
Has a user and permissions system so only certain people can use certain devices.	Works well however could be expanded to have different types of user E.G. user and admin.
Many users can access the system at once.	Success, tested with no problems.

Allow clients to send and receive parameters.	Success.
Will allow devices to be interacted with; data can be sent to, and be received from, the devices.	Success.
A website interface to access and control the system	Success.
Will call call the API functions and generate and modify the webpage based. upon the responses.	Success.
Accessible from mobile devices.	Success.
IoT devices must be easy to use.	Proves complex for non tech savvy users.
Website will call call the API functions and generate and modify the webpage based.	Success.
Website must be easy to access and use.	Reasonably achieved however as mentioned in previous section some technical knowledge may be required.
Website must be accessible from mobile devices.	Success.
IoT devices must take commands from a controller server via sockets.	Success.
IoT devices must be able to be connected to the user's WiFi network.	Success.
Iot devices should only take commands from the controller, not other devices; perhaps by the user of an authentication token.	Success.

Self identified problems with the system

One problem I have identified is that all of my communications across the network are unencrypted; this effectively renders my authentication key system useless if someone were packet sniffing on said network. They could get the key and send commands to a device maliciously. The same thing could occur for passwords, as they are sent across to the controller server unencrypted via http then hashed for comparison with the stored one.

Some solutions to these problems could be to, instead of an authentication key, use a digital signature which uses the same mathematical principles as a public-private key exchange to prove that a message was sent by a specific device.

And for the other problem, we could simply use https which would solve the packet sniffing problem very simply.

A second problem is add features: adding a feature to the program takes a long time to do as - effectively - 2 functions have to be written, one in JS for the website and one in python for the controller. I still stand by my API-Client system as it works well however it is simply something to take into consideration if I were to make a similar system in the future.