Faculty of Engineering and Applied Science

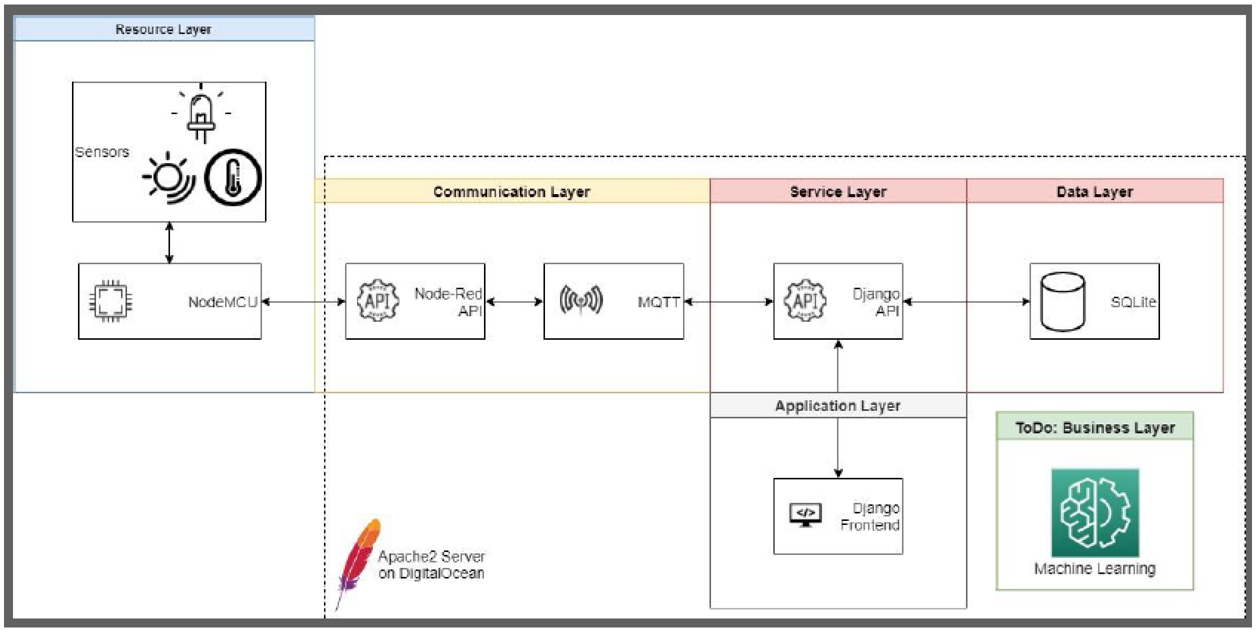
Internet of Things SOFE 4610U Final Project

Smart Thermostat

GitHub Repository:
https://github.com/danialasghar/SOFE4610Project

Danial Asghar - 100671850
Hamza Farhat Ali - 100657374
Yusuf Shaik - 100655451

Architecture Design Description



Resource:

DHT 11 Sensor:

The temperature and humidity sensor was able to gather all readings to display them as celsius degrees and percentages. This was always the real time readings for the user to set or automate the desired temperature.

Photoresistor:

The photoresistor was the independent variable used by the system to determine if the house was occupied or not. Under the assumption if the lights are turned off, the low value from the photoresistor resulted in a home status being not occupied. A higher value resulted in home status being occupied. This allowed the user to automate the desired temperature for when they return home.

LED lights:

The LEDs that should turn on depending on the current temperature. The blue LED should turn on to indicate that the house “is cooling”, and the red LED should turn on to indicate that the house is “warming up”. This should automatically turn on depending on the current temperature of the location, and whether or not a user is detected as home from the photoresistor. Lastly, Information from the temperature/humidity sensor, and photoresistor were combined into a URL encoded string to post this information to our node-red server.

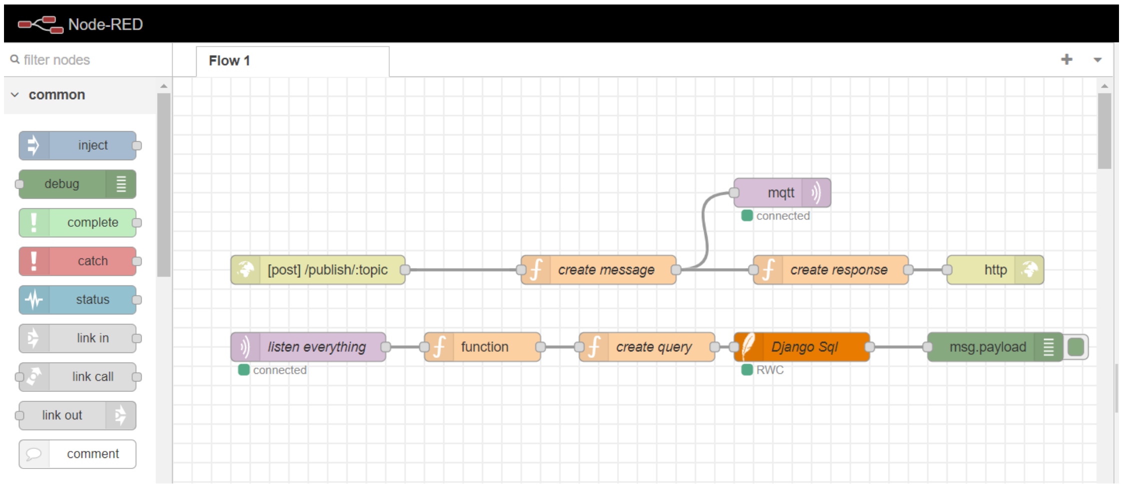
Node Microcontroller

This microcontroller has GPIO pins with WiFi adapter supporting a TCP/IP protocol stack to multiple client connections. Furthermore, it supports i2C with digital pins and 1 ADC channel, along with 4.5V-7V power supply via USB port. This was integrated with our project to all the sensors through a breadboard. This microcontroller was critical to read all sensor data from the photoresistor, DHT 11, and LED light status. This was posted as an HTTPS packet to post the information layer to the node red endpoint.

Communication:

Node Red API & MQTT

The information sent as a url-encoded message is received at the endpoint for the node-red api. The node red pipeline was configured to subscribe to this endpoint, and send an OK HTTP response back to the device that sent the message. Within the pipeline, the url would then be parsed, the useful information would be extracted, and the information would be sent to the backend endpoint for the django application.



Service:

Django API

The django API is an interface into the SQLite database used for this application. Any information received by this endpoint will be automatically inserted into the database. Once the information is received from the node-red server, the conditions (temperature, humidity, home) are entered into the database for permanent storage.

This API also provides a GET endpoint that will be used to retrieve information from the database. When the application interacts with this endpoint, they will be able to access all information in the database.

Data:

SQLite Database

Information received into the API endpoint is automatically inserted into the database. The SQLite database allows for a permanent structured storage facility that will allow for fast analytical queries in the future due to its structured nature. This component is used in this application. This is used to store conditions of the home in the following schema: id (int), temperature (float), humidity (float), home (boolean). The Django API will provide an interface to interact with this database which is how information is stored and retrieved.

Application:

Django front end connects to the Django API and performs a GET request to display information on the webpage. This component is used for providing an interface for the user to view the current and past conditions in their home, as well as graphs depicting the change of the conditions in their home over time.

Architectural Design - Design Decisions

Communication Protocols

One of the main design decisions for this project was the communication protocol that we decided to use for this project. This would handle the communication between our sensor, edge device, and our backend (digital ocean). There were three main options that we had: MQTT, CoAP, and REST. Our requirements for our application were to allow for two way communication between the edge device and cloud server. This would allow the temperature information to be conveyed, but also allow a user to change the thermostat to lower the current temperature.

With REST, our system would allow for the information to be uploaded in intervals, but it would require that the edge device constantly poll the endpoint to determine if the user has requested for the temperature to be updated. This method would cause a large amount of overhead on our system, and we therefore decided against it. Our system will need a communication protocol that follows the pub-sub architecture, which is why we looked into CoAP. Unfortunately, although CoAP follows the pub-sub architecture required for this project, we wanted to ensure that our system was reliable, and CoAP uses UDP rather than TCP. We wanted to ensure our system remains reliable even if it requires adding overhead due to the acknowledgement component.

Our final communication protocol was the first protocol we researched. MQTT follows the pub-sub architecture, while using TCP to ensure reliable communication. MQTT also uses SSL to ensure that all information transmitted is secure. Our application sends temperature data, as well as a boolean value indicating whether anyone is vacant in the home, which means that our application needs to be extremely secure to ensure that this information is not intercepted.

Security vs Development Time/Cost

Another important decision included implementing security throughout the entire application, as well as throughout the pipeline. Implementing security within the application included registering our domain, gaining an ssl certificate, securing node-red, securing our django backend, endpoints, and the implementation of connecting to our api over HTTPS. All of these steps have a very high development time and cost, but also greatly increase the security of our application. Leaving out the security component would have cut our development time in more than half. However, due to the nature of our application, it is of utmost importance that we ensure our end to end communication is as safe as possible. This includes all of the previously mentioned security enhancements, as well as integrating each of these components together to ensure we did not lose any functionality due to the added security.

Test Cases

Test Case 1: Position of navigation bar and other card views

This was mainly done through the CSS padding using bootstrap. This was to make sure the navigation bar was dynamically positioned to the left side while all card views are displayed to the center of screen. The card list view had constraints for top padding. This was tested through refresh upon HTML pages until correct alignment was achieved according to UI requirements. This was also tested on multiple devices to ensure the ui was consistent, and easy to use. This test case passed when all devices displayed the navigation bar at the top of the screen.

Test Case 2: On click to scroll

Each major component within the HTML code was distributed through div tags, and each id was called on each click. This should allow the user to click on each tag in the navigation bar, and be navigated to that major section. This was manually tested on multiple devices, with various browsers to ensure the ui worked consistently on all devices. This passed the test case when on click to scroll worked over 10 times on each device.

Test Case 3: Screen size for different screens, mobile and desktop
This was implemented through calling the bootstrap template and determining the content width and height automatically. This was tested by hosting the application locally on the network, and testing the application on mobile devices, as well as other computers. This passed the test case when the application dynamically resized on all screen sizes.

Test Case 4: Easy to use interface
This test case defines the application displaying the correct information to the user in each component that it defined: Temperature, temperature history, humidity, humidity history, home, home history. The containers in the application should display the information separately in each card-view as defined. This test case was passed when the information was displayed separately on the arduino editor, and compared to the information displayed in each card-view within the application

Test Case 5: PhotoResistor Reading
Test case results were achieved by manually isolating the arduino C code to check if the readings were accurate. The developers also placed their thumbs over the resistor to confirm a lower lumin reading was received. This meant that the photoresistor was reading the correct values. This was then integrated with the final C code. The test case was passed if the photoresistor readings lowered as the photoresistor was covered.
Test Case 6: DHT11 Sensor Reading
Test case results were achieved by manually isolating the arduino C code to check if the readings were accurate. The developer sent warm and cold air around the sensor by blowing to show increase or decrease in temperature and humidity readings. This was then integrated with the final C code.The test case was passed if the arduino serial monitor displayed the correct temperature that matched the approximate temperature within the house.

Test Case 7: Django Application
This was manually tested by checking if the port displayed the user interface within a local server through multiple devices. Each page of the application: home, api endpoint, and database ui, should be displayed at the correct pages: /index, /api, and /admin respectively. This test case was passed after the application was hosted on our digital ocean droplet, and each page routed to the page that displayed the correct functionality.

Test Case 8: Node-Red Server

This test case ensures that the node red server is functioning as required. This was tested by sending a new simulated reading using curl. The Node-Red queries should parse the string sent through url encoding, and display the information as a json object within the Node-Red debug terminal. This test case passed when information was sent to the endpoint using curl, and the correct information was displayed in the correct format on the Node-Red debug terminal.

Test Case 9: Django SQLite Database

The SQLite database was tested by checking if all condition statements were saved from the temperature, home, and humidity readings. Information should be uploaded to the endpoint using curl, and a combination of the Node-Red queries and functions should decode the string that was sent to the server, and enter the information into the correct column of the sqlite database. This test case was passed when information was sent to the application through the sensors, and the information was displayed on the /admin page which is an interface into the database. The administrator was able to login and see the exact values that we're sent from the Node MCU stored in the database under the correct columns.

Test Case 10: Correct LED Colour

This application contained 2 LEDs that should turn on depending on the current temperature. The blue LED should turn on to indicate that the house "is cooling", and the red LED should turn on to indicate that the house is "warming up". This should automatically turn on depending on the current temperature of the location, and whether or not a user is detected as home. This was tested by displaying the current temperature and home status, and turning on the appropriate light for the situation. Since there are 2 variables, these are the possible combinations:

Temperature	Home	Output (LED)
Below 25	True	Red
Below 25	False	Do Nothing
Equal to 25	True	Do Nothing
Equal to 25	False	Do Nothing
Above 25	True	Blue
Above 25	False	Do Nothing

To recap, we only want the temperature to automatically change if a user is home. Otherwise, it is a waste of energy.

The variable combinations were analysed, and each possible combination was tested for through simulating these specific values. This test case was passed. Additionally, the system was run normally with the sensors determining the values, and all conditions were tested manually with the sensors. The photoresistor was physically covered to simulate true and false, and the home temperature was around 25 degrees so blowing on the sensor allowed the user to easily change the temperature to test all conditions. This test case passed when all test conditions provided their expected output.