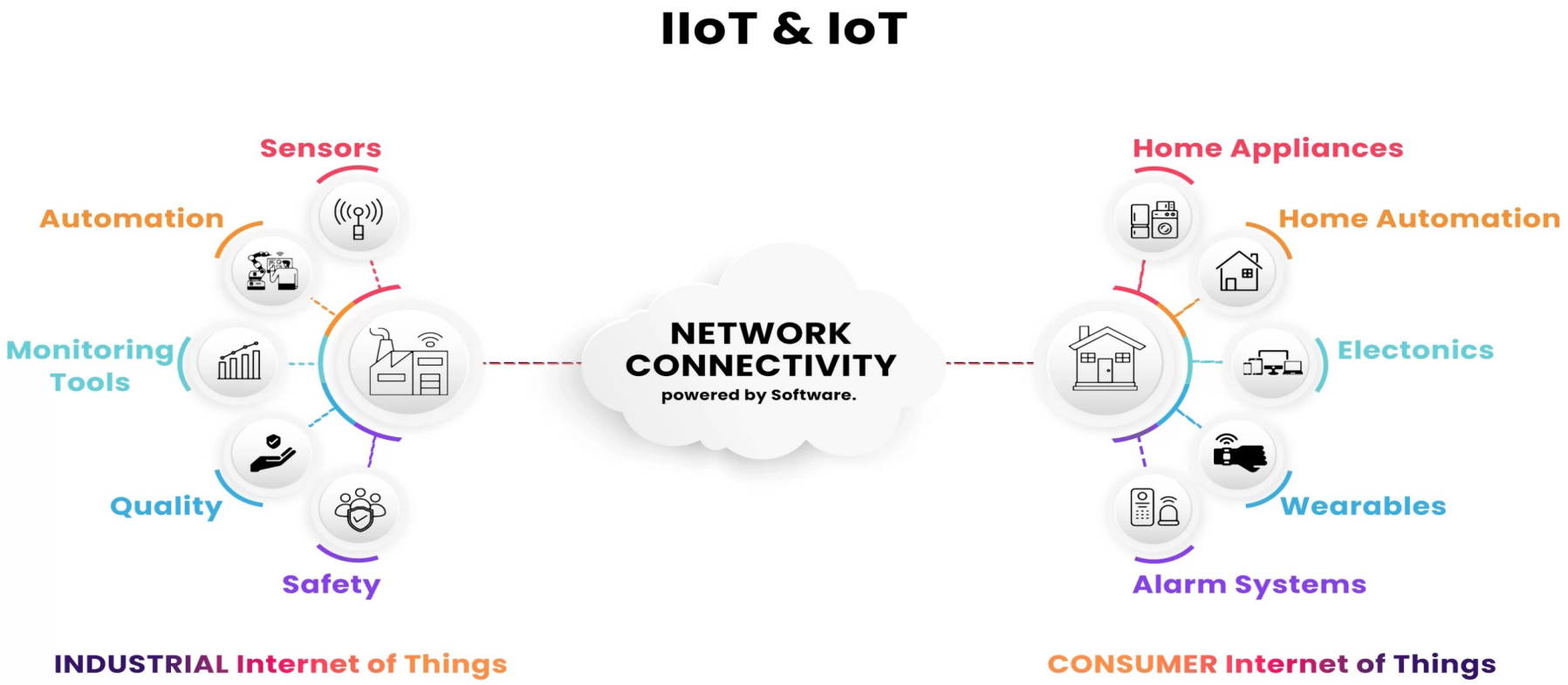
MQTT

Publishing and Subscribing protocol for IoT & IIoT

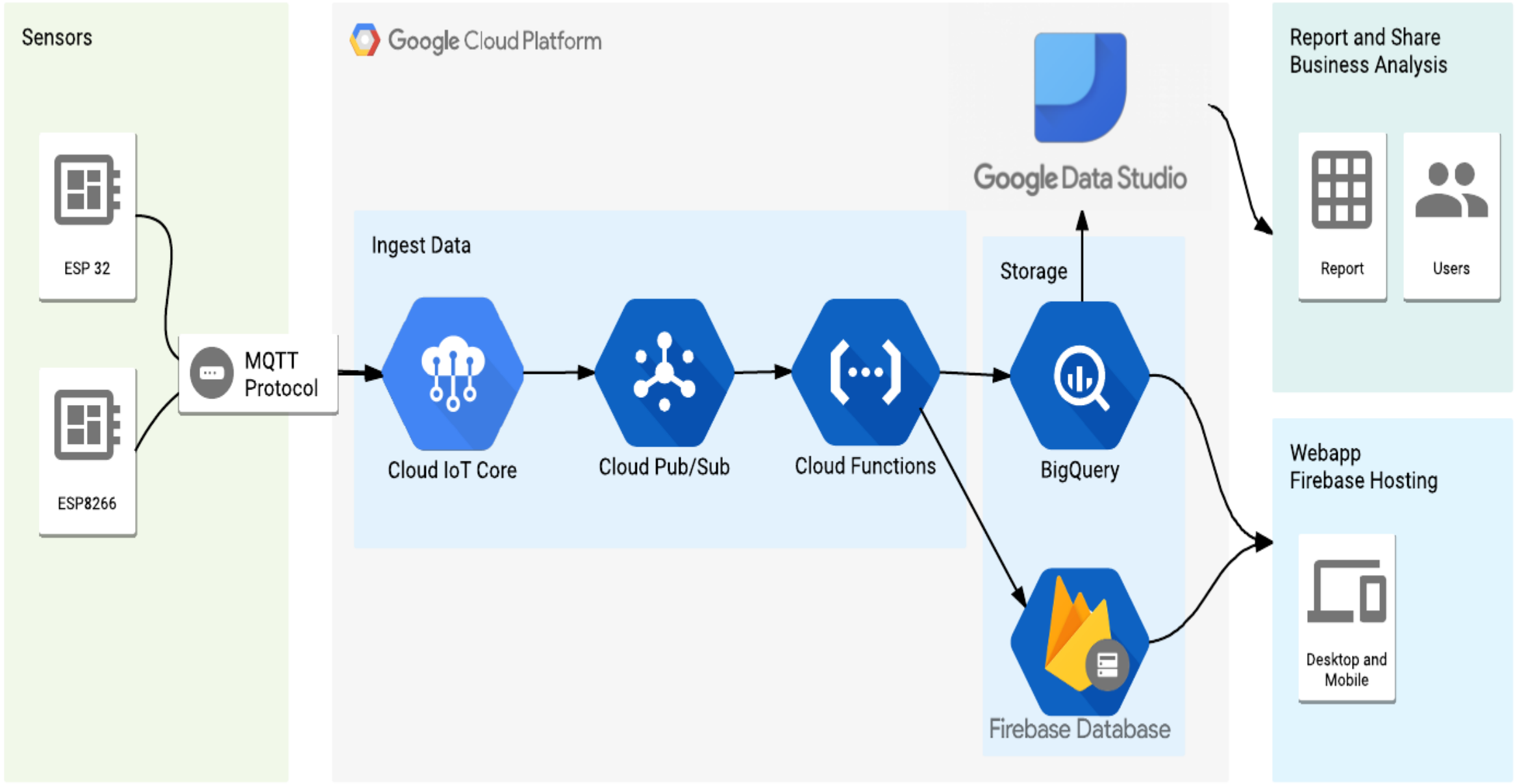
-Bilal El Barbir

An Example: Amazon IoT

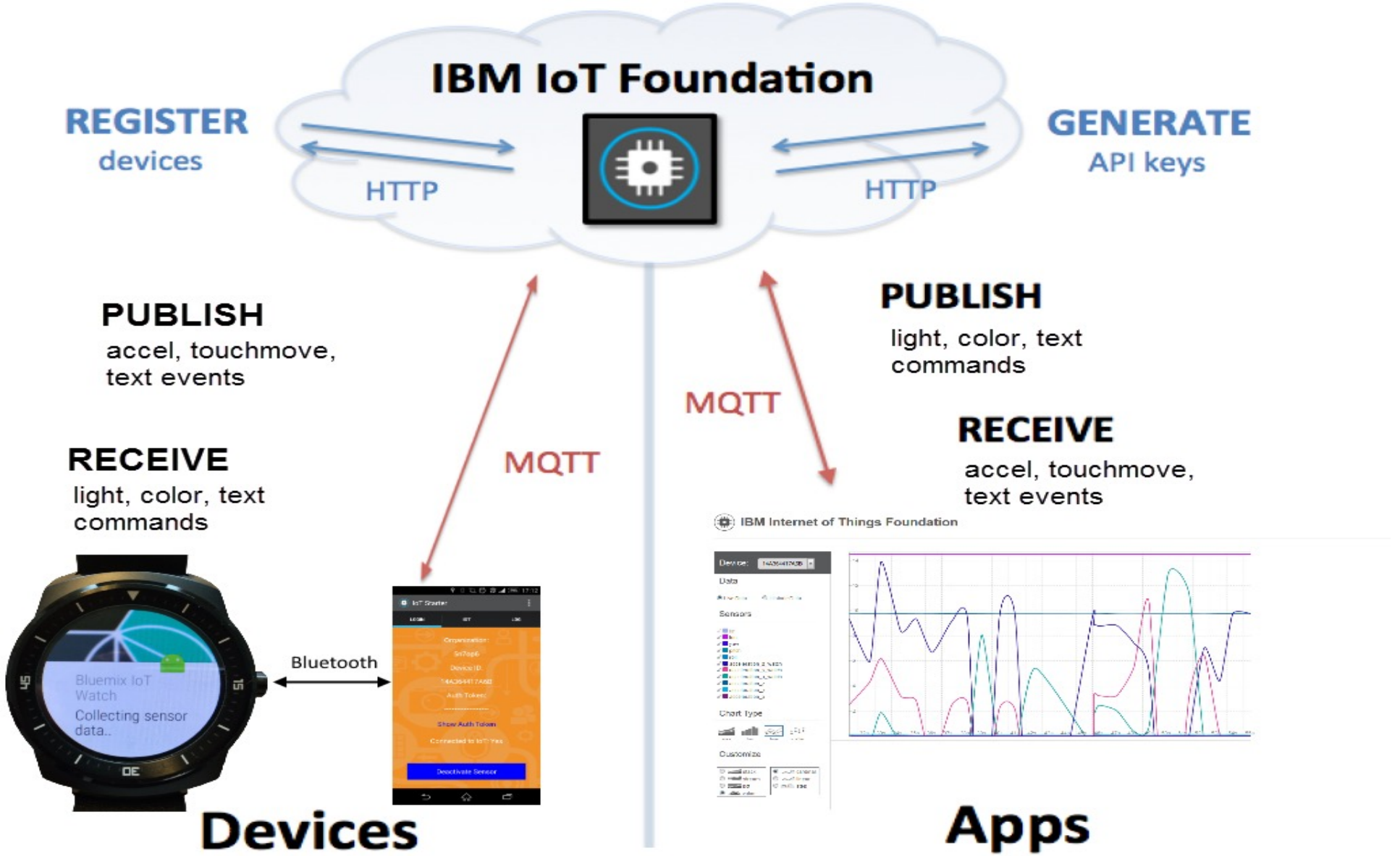
- Ask Amazon Alexa to turn on the car and heat until 25 degrees.



Another Example: Google and MQTT



Another Example: IBM and MQTT



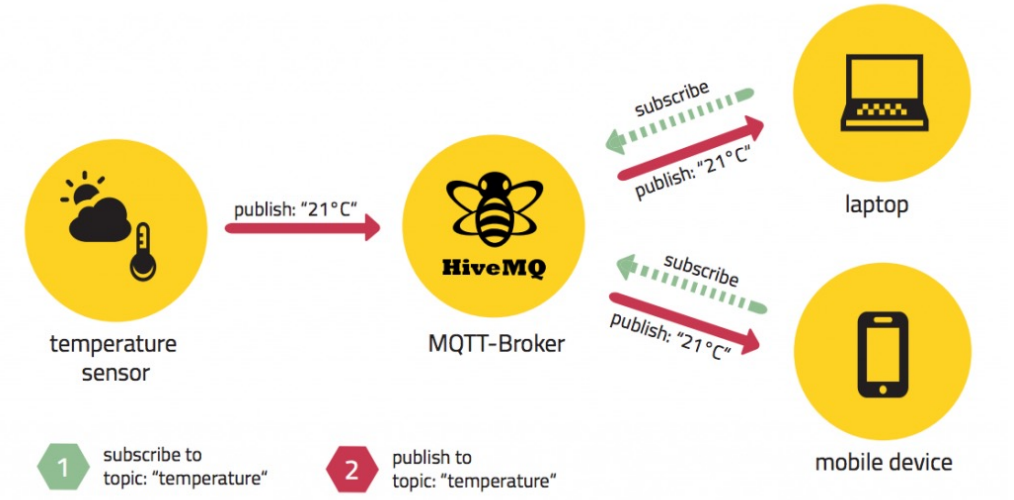
MQTT: Sensor readings with pub/sub

Decoupled in space and time.
The clients do not need each other's IP address and port (space) and They do not need to be running at the same time (time).

The broker's IP and port must be known by clients.

Namespace hierarchy used for topic filtering.

It may be the case that a published message is never consumed by any subscriber.



MQTT

- Message Queuing Telemetry Transport (old acronym) since the 1990's
- "Telemetry" is from the Greek remote measure
- Created by Andy Stanford-Clark (IBM) and Alan Nipper - now part of OASIS
- Originally built for oil pipeline monitoring over satellite connections.
- Satellites appropriate because pipelines are remote

MQTT

- Built for a proprietary imbedded system now shifting to IoT
- You can send anything as a message. Up to 256 MB.
- Built for unreliable networks
- Enterprise scale implementations
- Message have a topic, quality of service and retain status associated with them.

MQTT Clients

- A client may publish or subscribe or do both.
- An MQTT client is any device from a micro controller up to a full fledged server, that has an MQTT library running and is connecting to an MQTT broker over any kind of network. (from HiveMQ MQTT Essentials)
- A client is any device that has a TCP/IP stack and speaks MQTT. MQTT-SN does not require TCP.
- Client libraries widely available (Android, Arduino, iOS, Java, Javascript, etc.)
- No client is connected directly to any other client

MQTT Broker

- The broker is primarily responsible for receiving all messages, filtering them, decide who is interested in it and then sending the message to all subscribed clients. (From HiveMQ MQTT Essentials)
- May authenticate and authorize clients.
- Maintains sessions and missed messages
- Maintains a hierarchical namespace for topics and allows subscribers (but not publishers) to use wildcards (+ and #).

Topics are organized into a hierarchical namespace

- Suppose a client publishes to mm6House/Kitchen/Sensor/Temperature
- Another client might subscribe to:

  mm6House/Kitchen/Sensor/Temperature
- Or, using a single level wildcard (+)

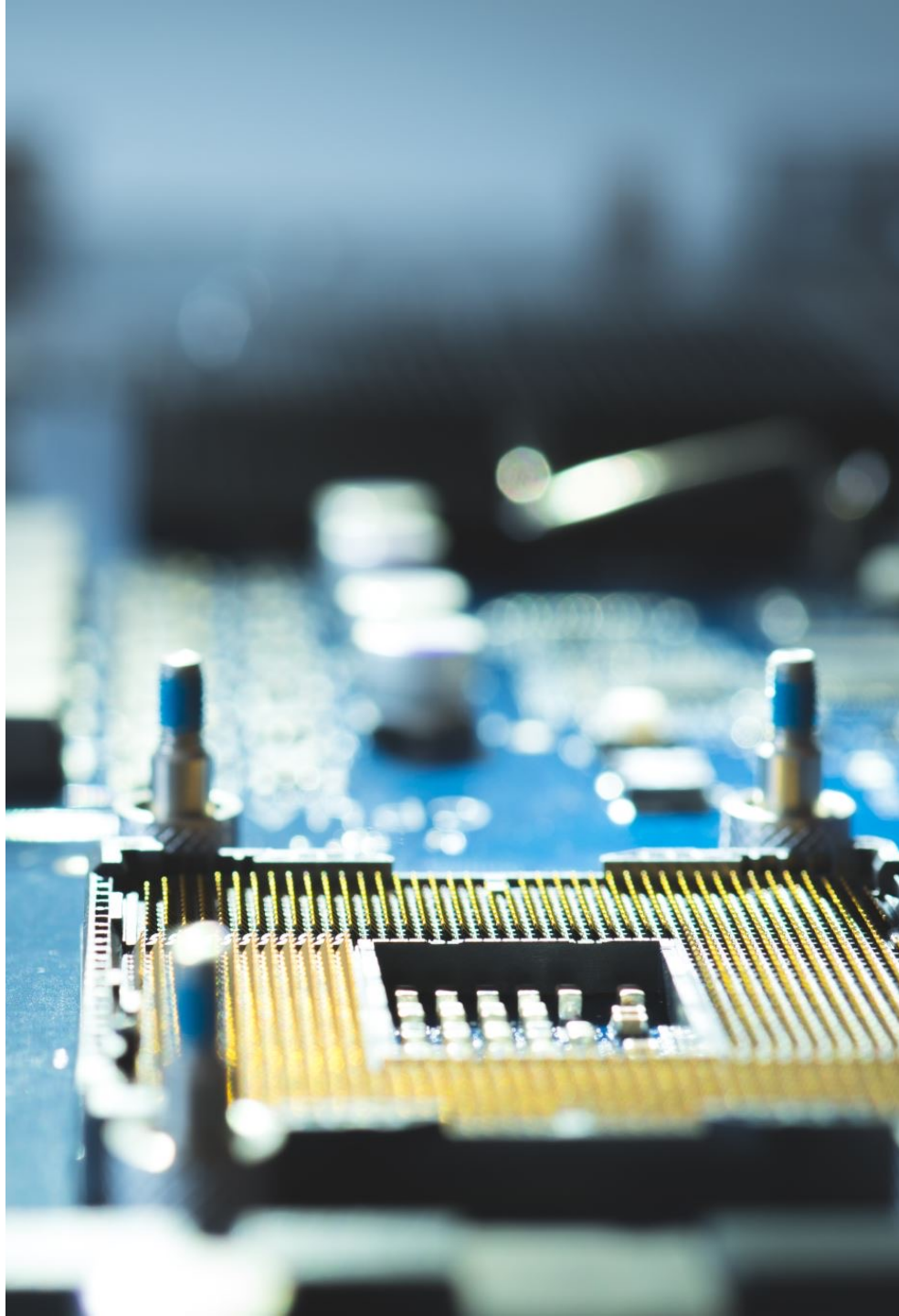
  mm6House/Kitchen/+/Temperature // All children of Kitchen that

  // have a child called Temperature
- Or, using a multi level wildcard (#)

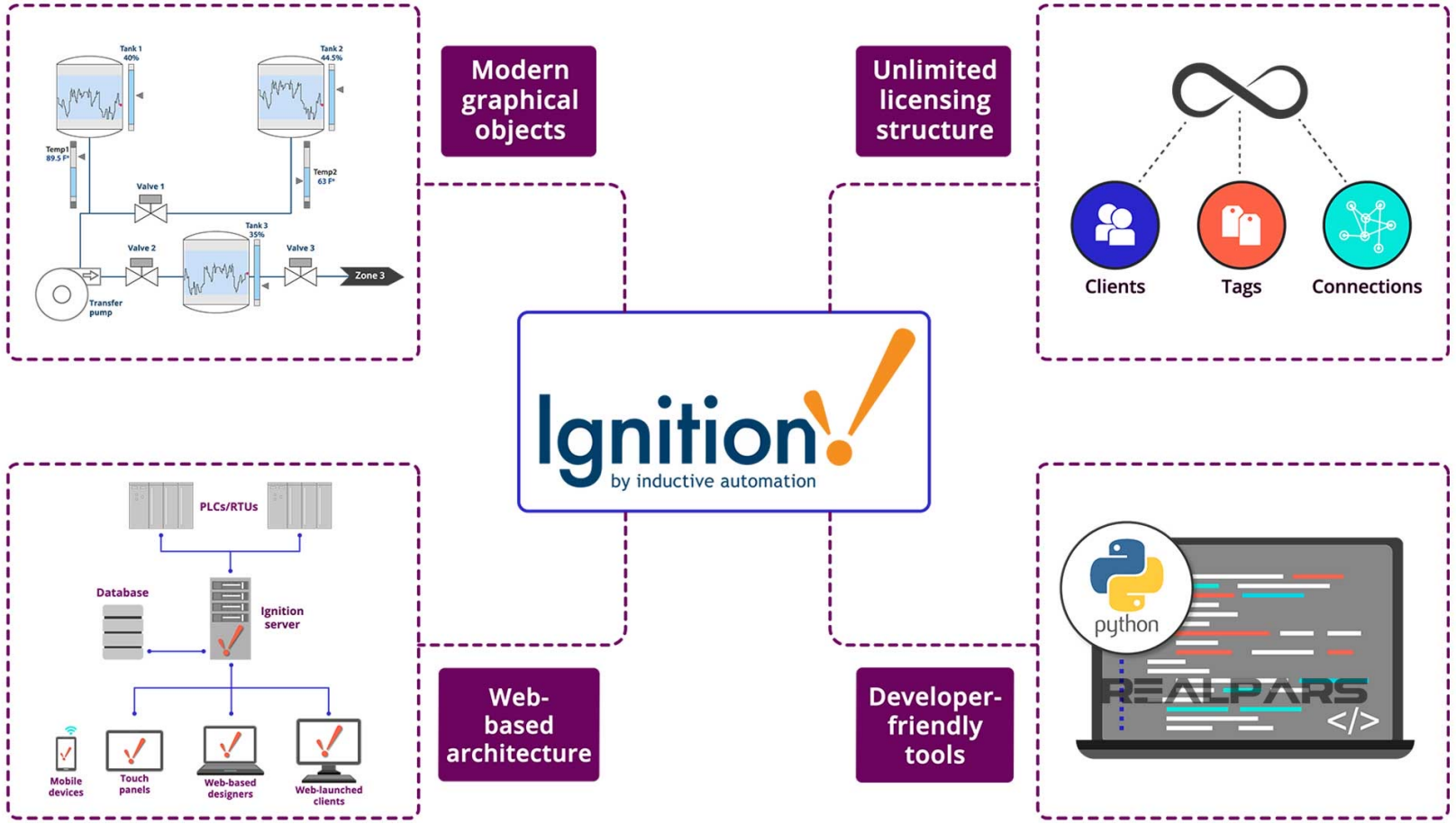
  mm6House/Kitchen/# // Goes deep
- The octothorp (#) must be the last character.
- To see every message, subscribe to #.

Ignition SCADA

- Ignition SCADA is a powerful software platform that is designed to help businesses and organizations monitor and control their industrial processes in real-time. SCADA stands for "Supervisory Control and Data Acquisition" and refers to the process of collecting data from various sensors and devices, and then using that data to control and monitor industrial processes.
- The Ignition SCADA platform is built on a modern architecture that makes it easy to scale and customize, and it includes a variety of tools and features that can be used to build powerful industrial control applications.



Ignition SCADA



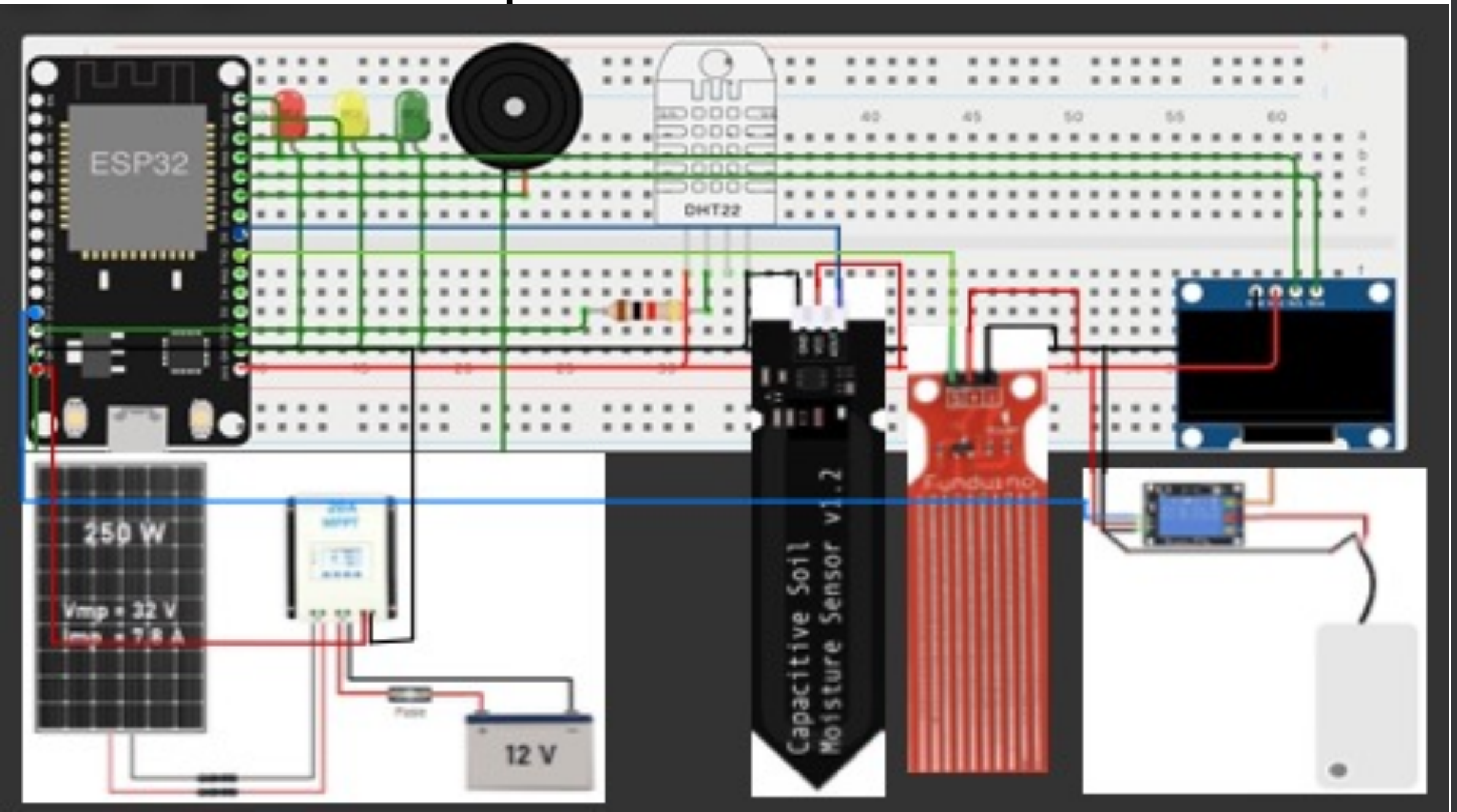
Some of the key features of Ignition SCADA include:

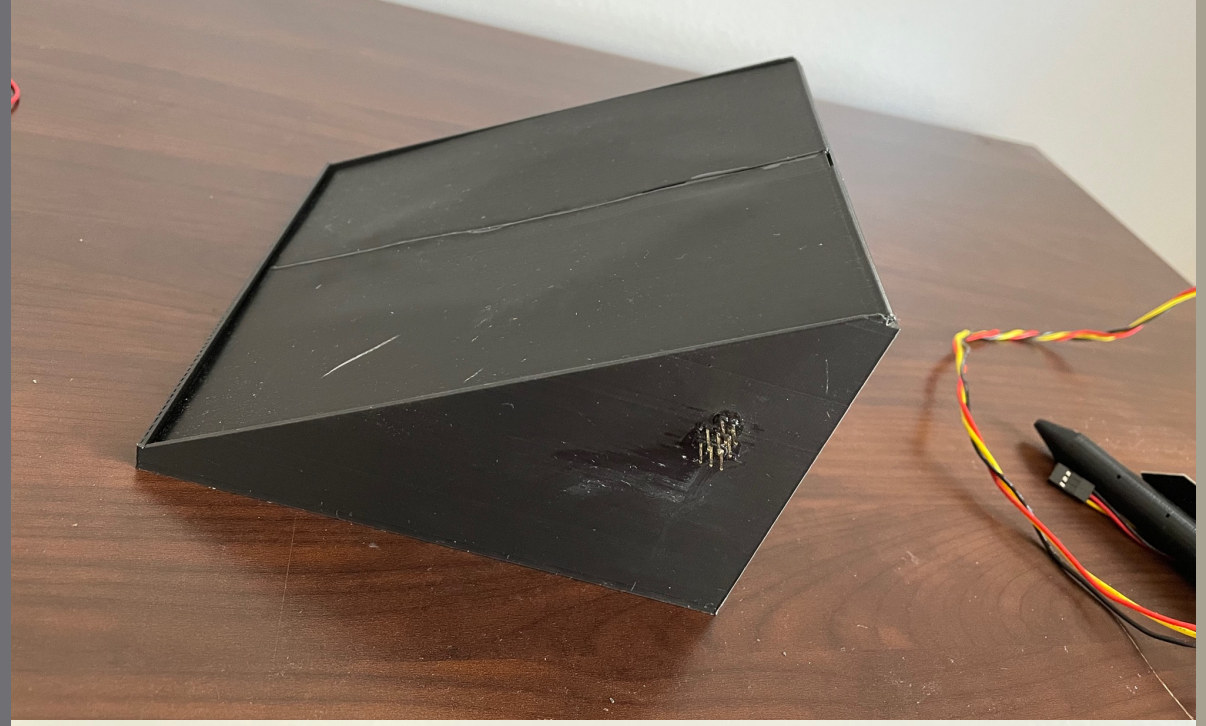
- Real-time monitoring
- Data logging: Ignition SCADA can log data from a wide range of sensors and devices, and can store this data in a variety of formats for later analysis.
- Customizable dashboards
- Mobile support

The following slides will cover my personal experience with MQTT, IIoT and Ignition SCADA…

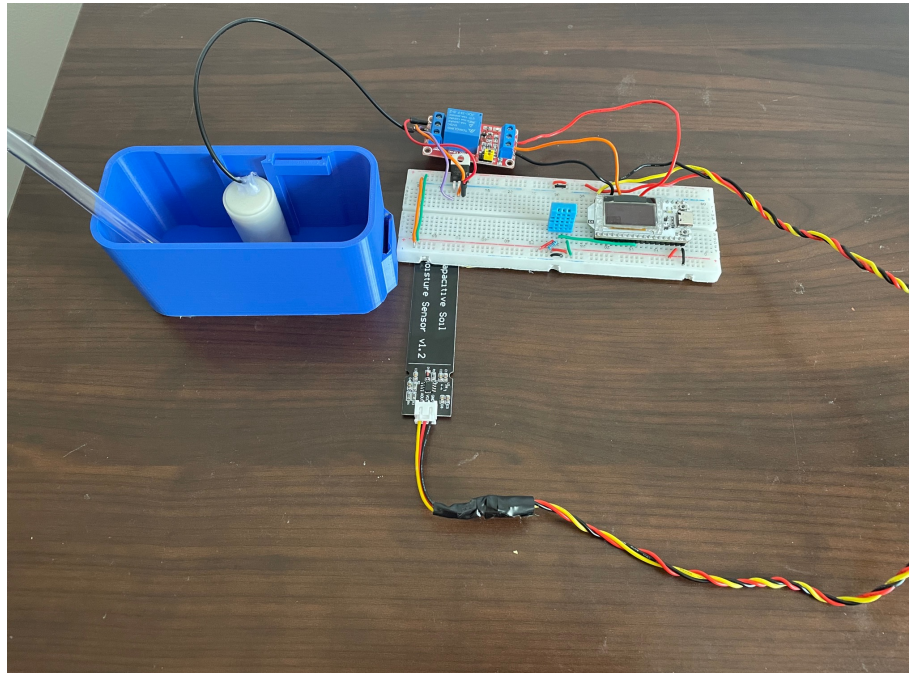
I present to you my project The Smart Automatic Irrigation Garden Monitor

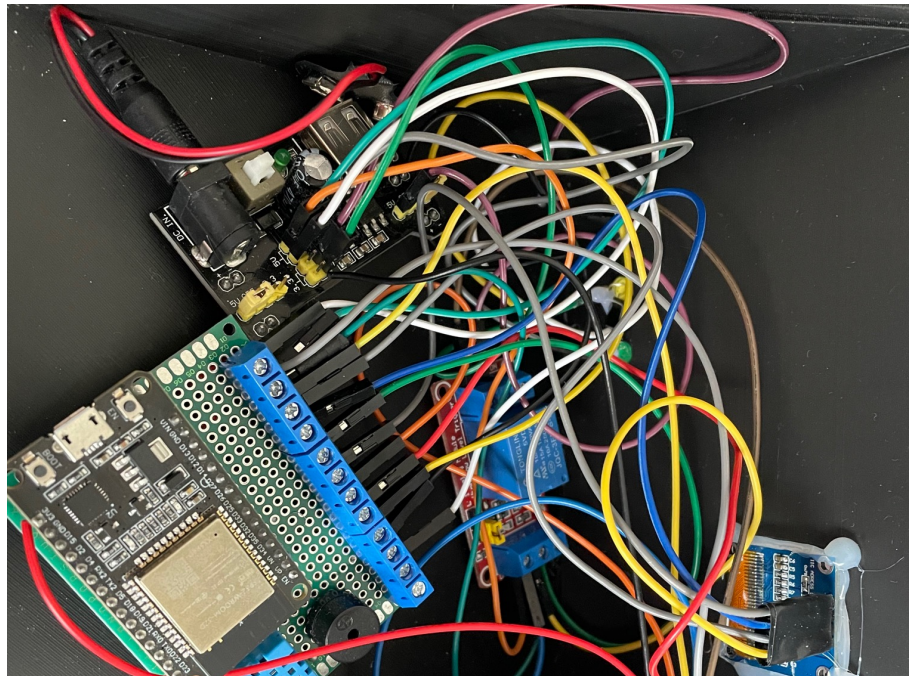
Hardware setup:





An enclosure to house all of the electronics (lights and a display in front to indicate sensor values and water tank level):

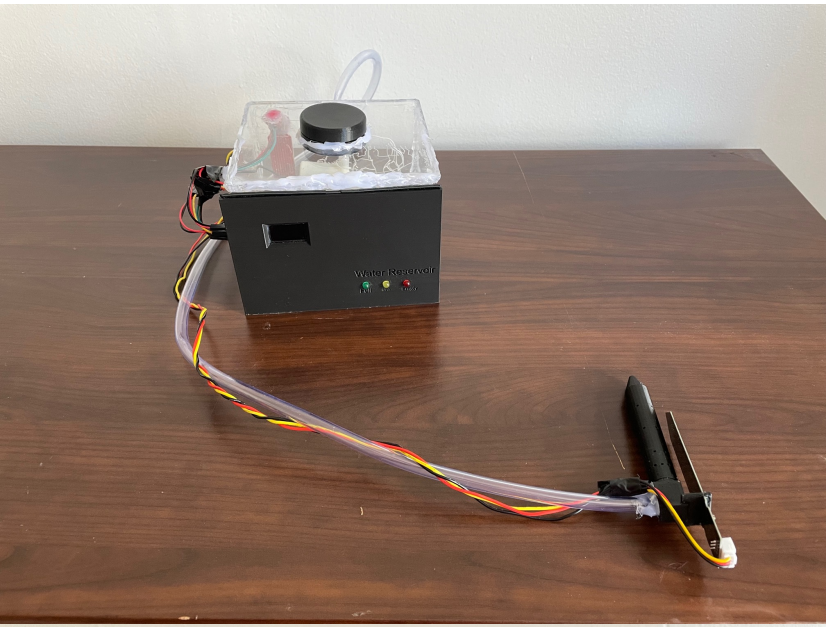


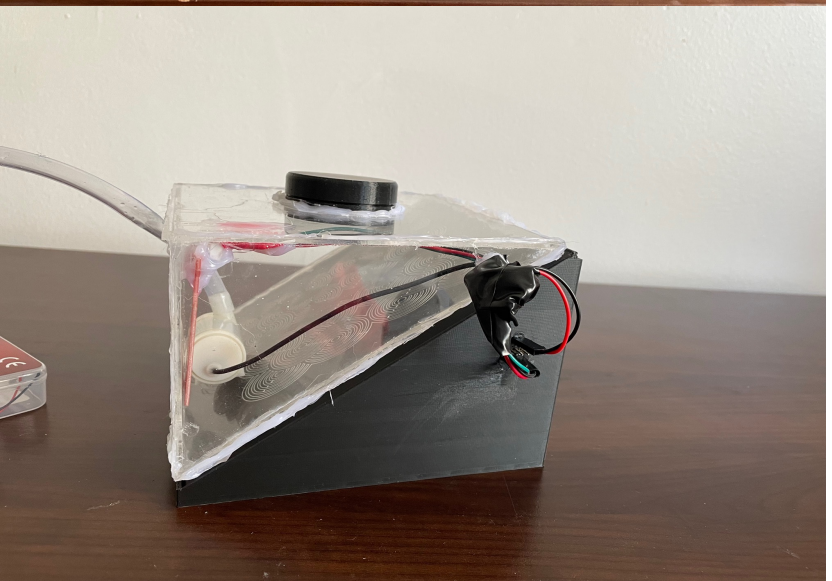


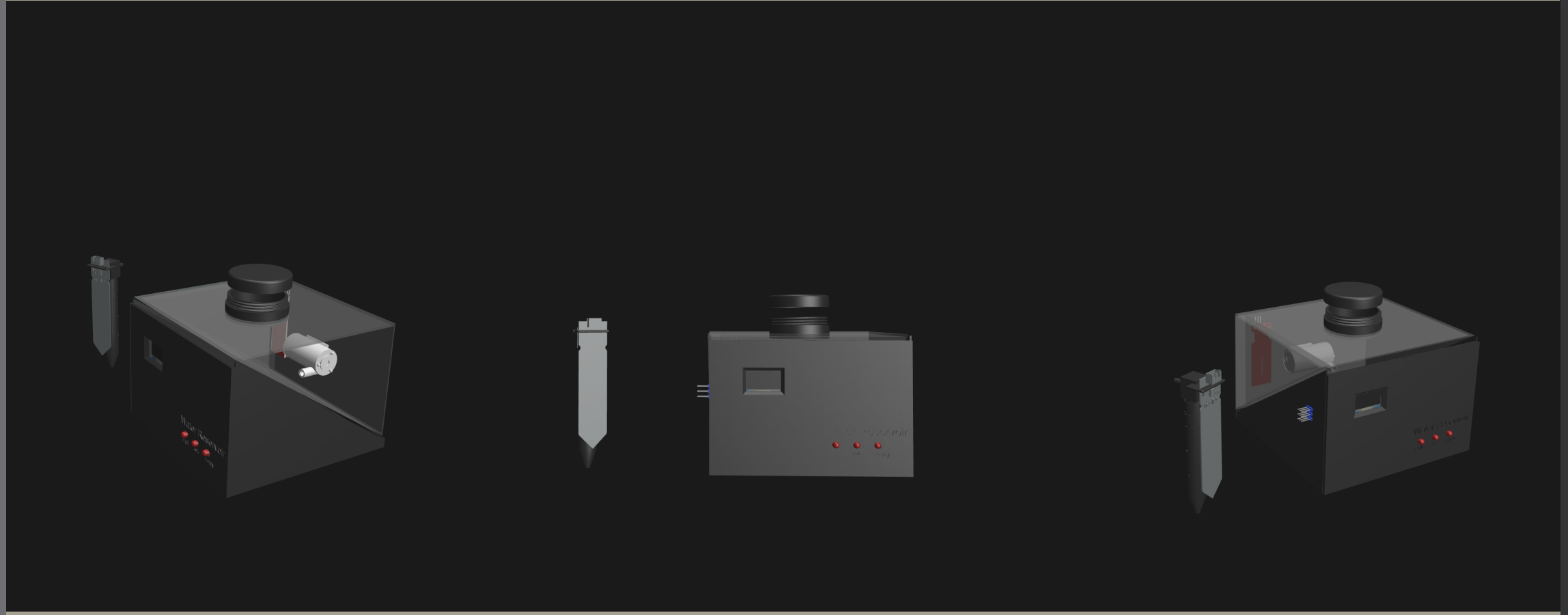
The inside of my main unit that's holding my ESPclient1 and another miniture unit which is my ESPclient2. The beauty of this, it's scalable!. You can keep adding clients and make up to topics!

This is the final look of my project.

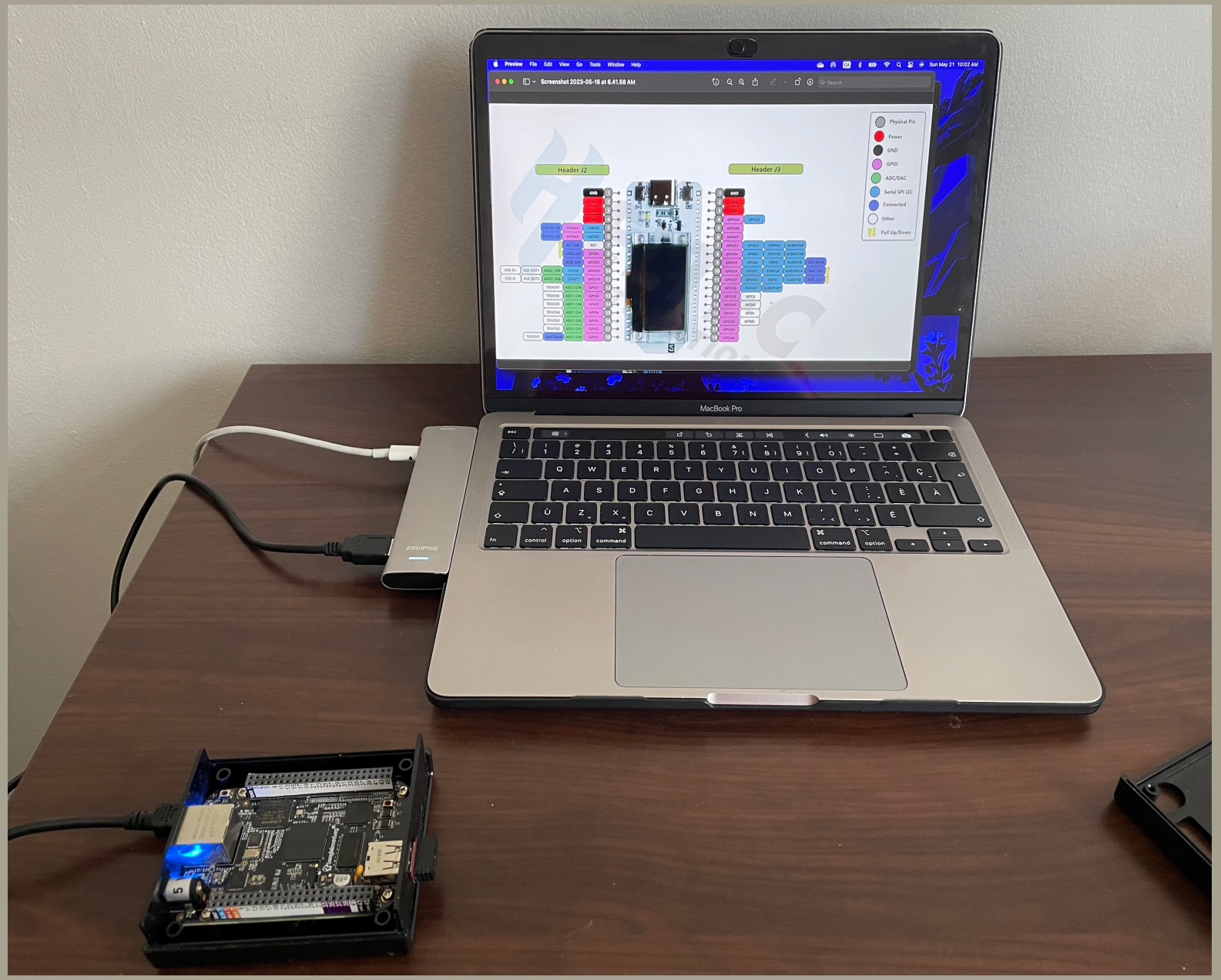
- It has a water tank on the other half forming a cube.
- A tube comes out of the back and waters the soil with a drip attachment.
- Next to the attachment there's a soil moisture sensor.





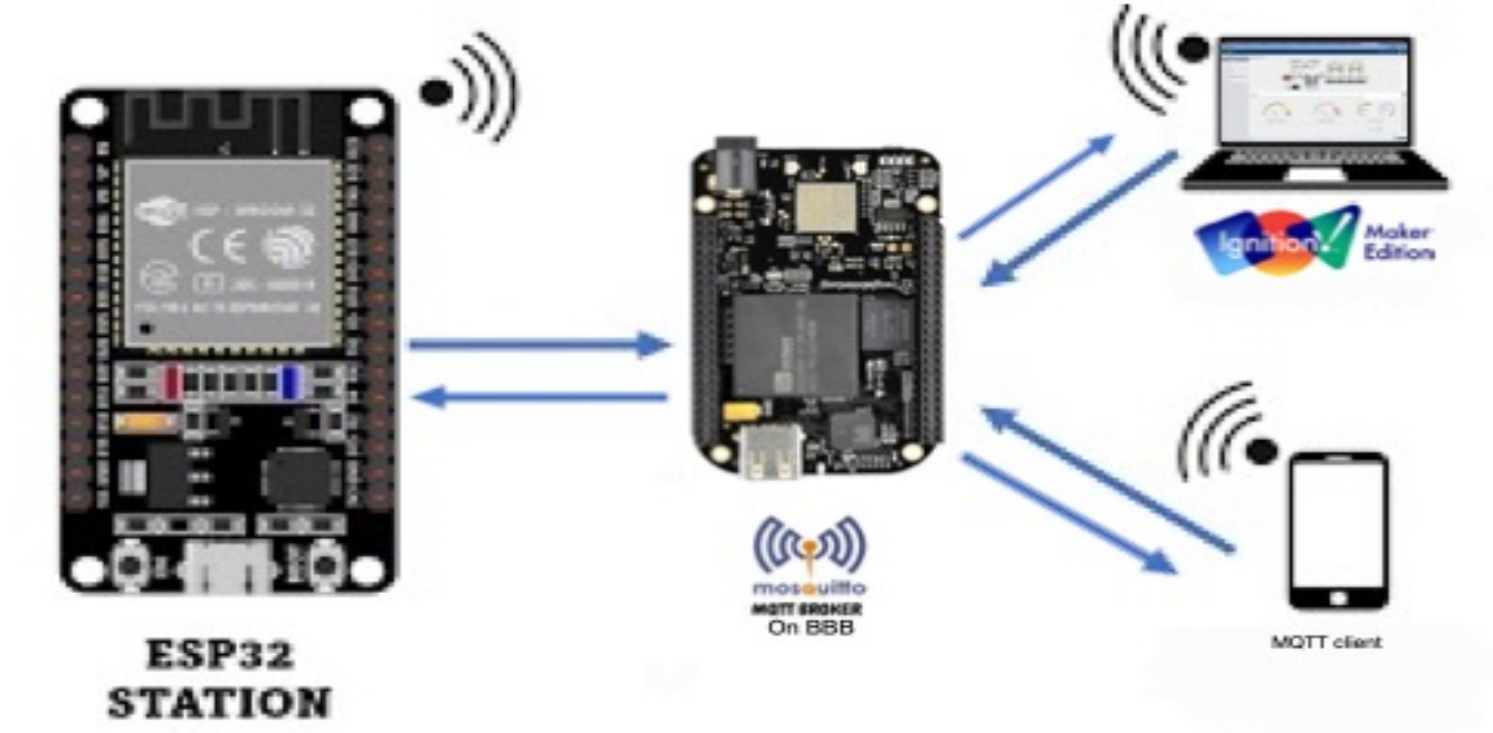


3D model:

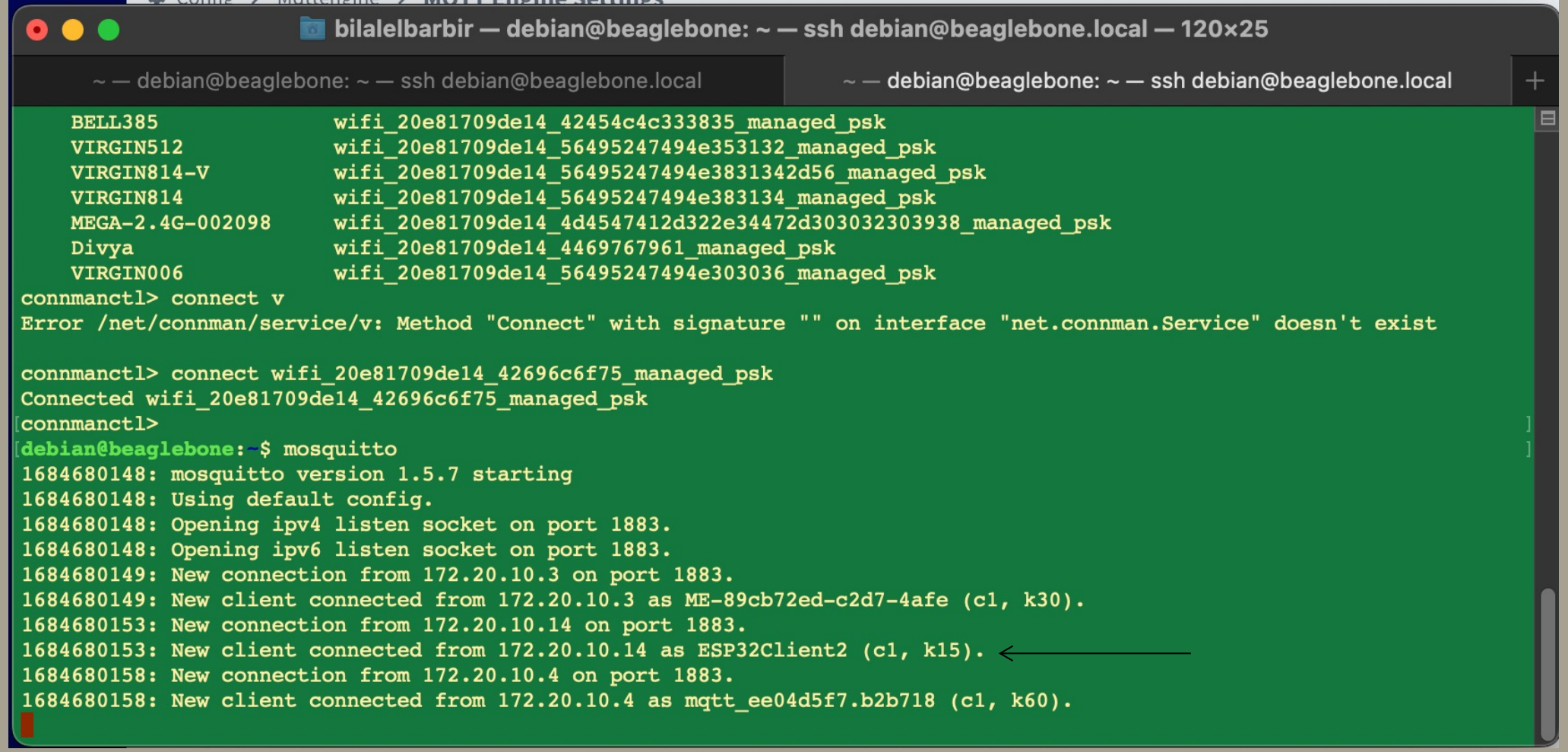


Next comes the software and IoT setup.

-For that I'm using my personal computer to run ignition designer and a beaglebone black to hold my mosquitto broker.


ESP32
STATION
mosquitto
MQTT BROKER
On BBB
Ignition
Maker
Edition
MQTT client

IOT setup



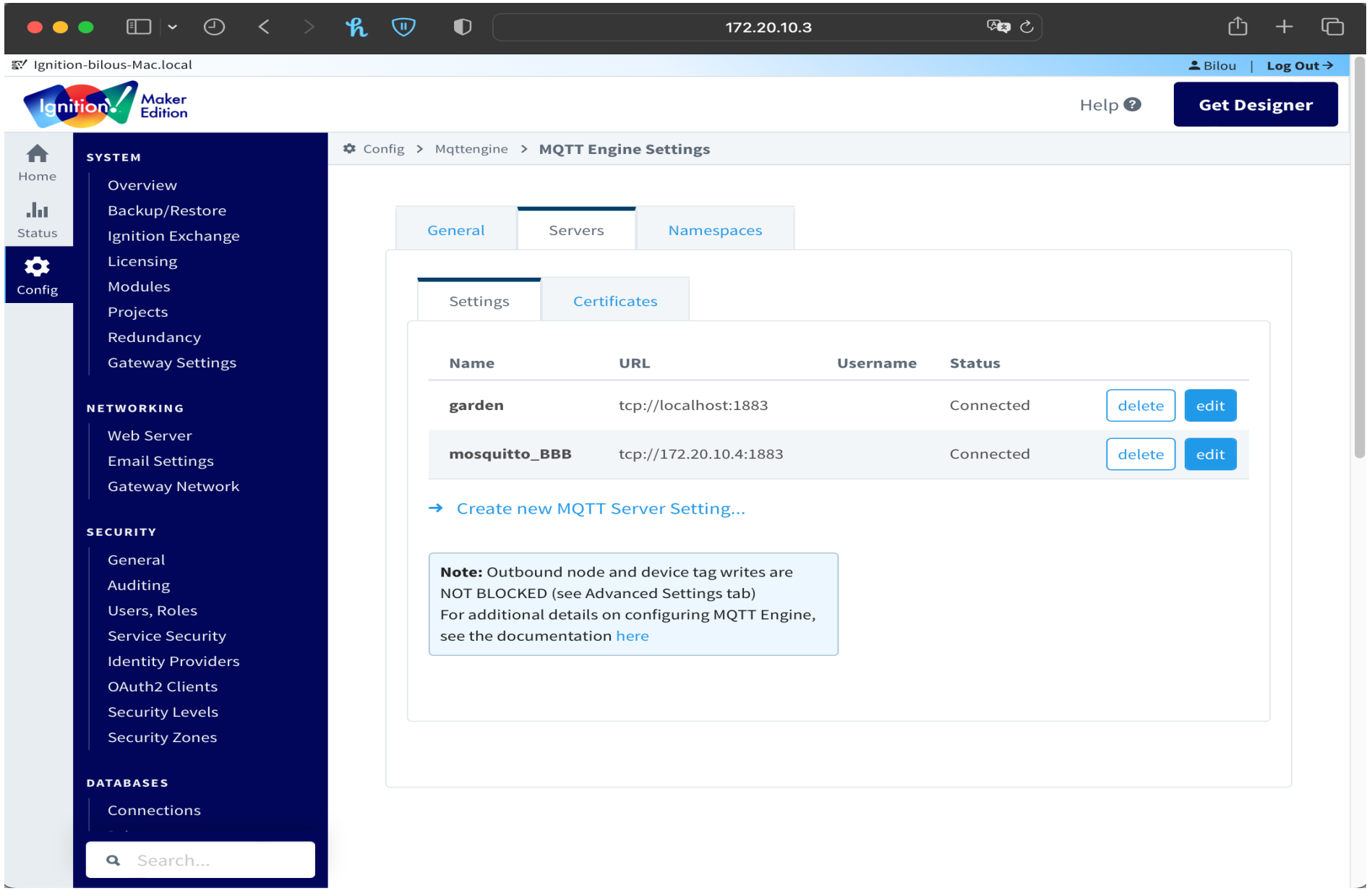
Running mosquitto on my beaglebone to listen for incoming ports: you can see how one of my ESPclients is connected.

```
[debian@beaglebone:~$ mosquitto_sub -h 172.20.10.4 -t esp/pump/state2
^C
[debian@beaglebone:~$ mosquitto_sub -h 172.20.10.4 -t waterpump/control2 -m 1
Error: Unknown option '-m'.

Use 'mosquitto_sub --help' to see usage.
[debian@beaglebone:~$ mosquitto_pub -h 172.20.10.4 -t waterpump/control2 -m 1
debian@beaglebone:~$
```

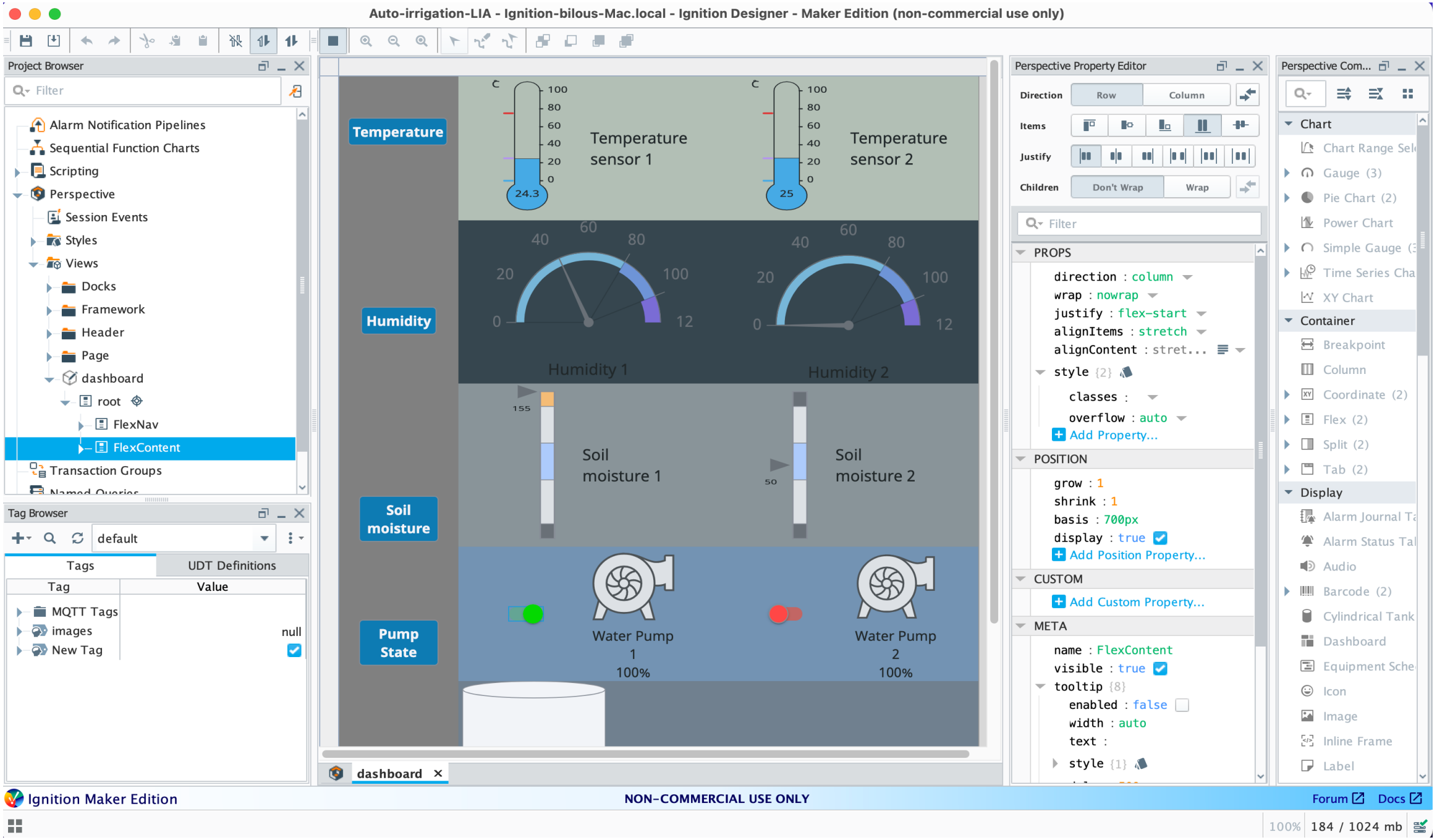
```
[debian@beaglebone:~$ mosquitto_sub -h 172.20.10.4 -t '#'
Temperature: 25.2, Humidity: 42.0
42.0
25.2
Soil Moisture: 155%
155
Water Level: 0%
0
```

Here is an example of how you can subscribe and publish from and to using the terminal:

Using ignition Maker to install and run MQTT modules:

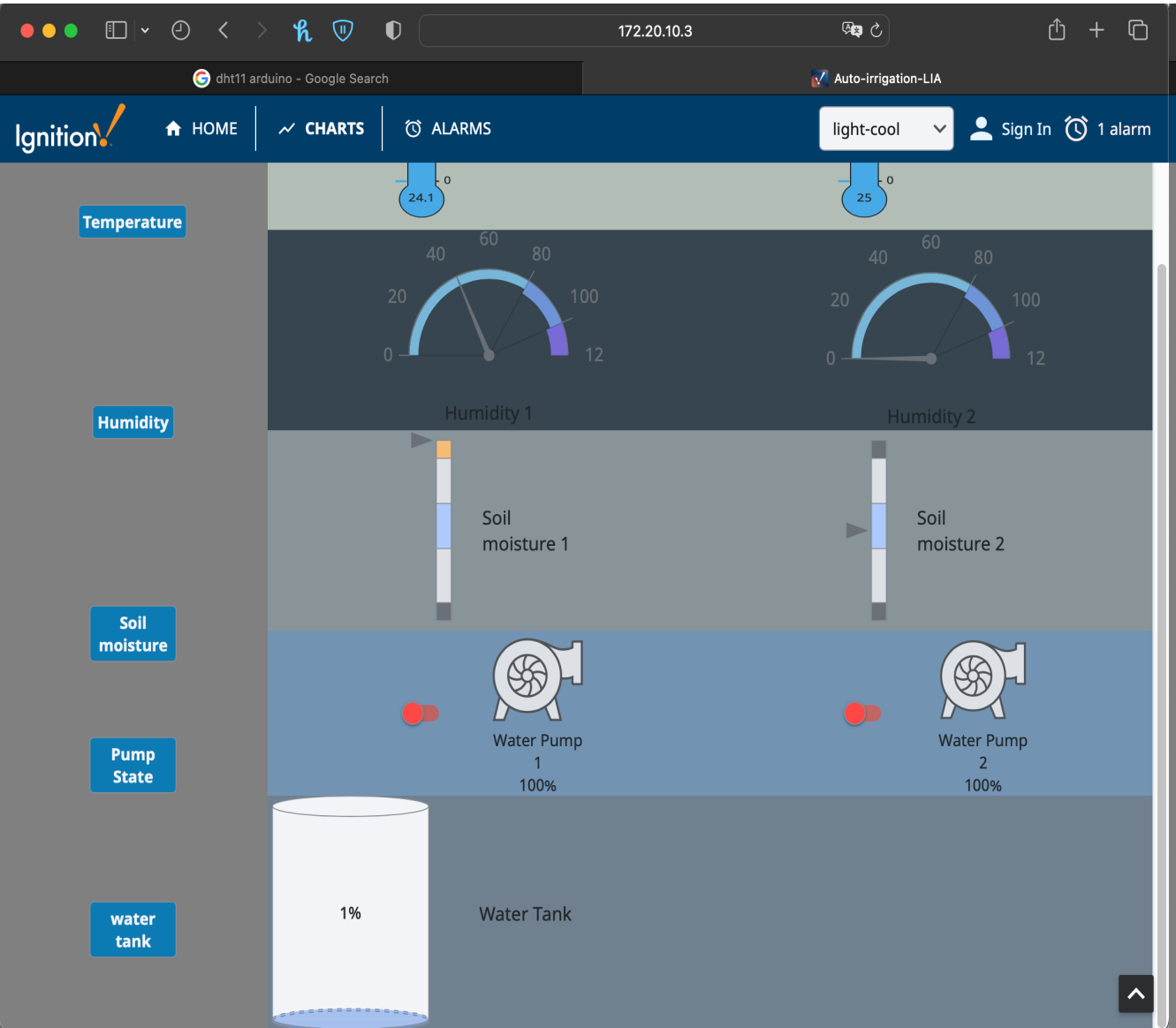


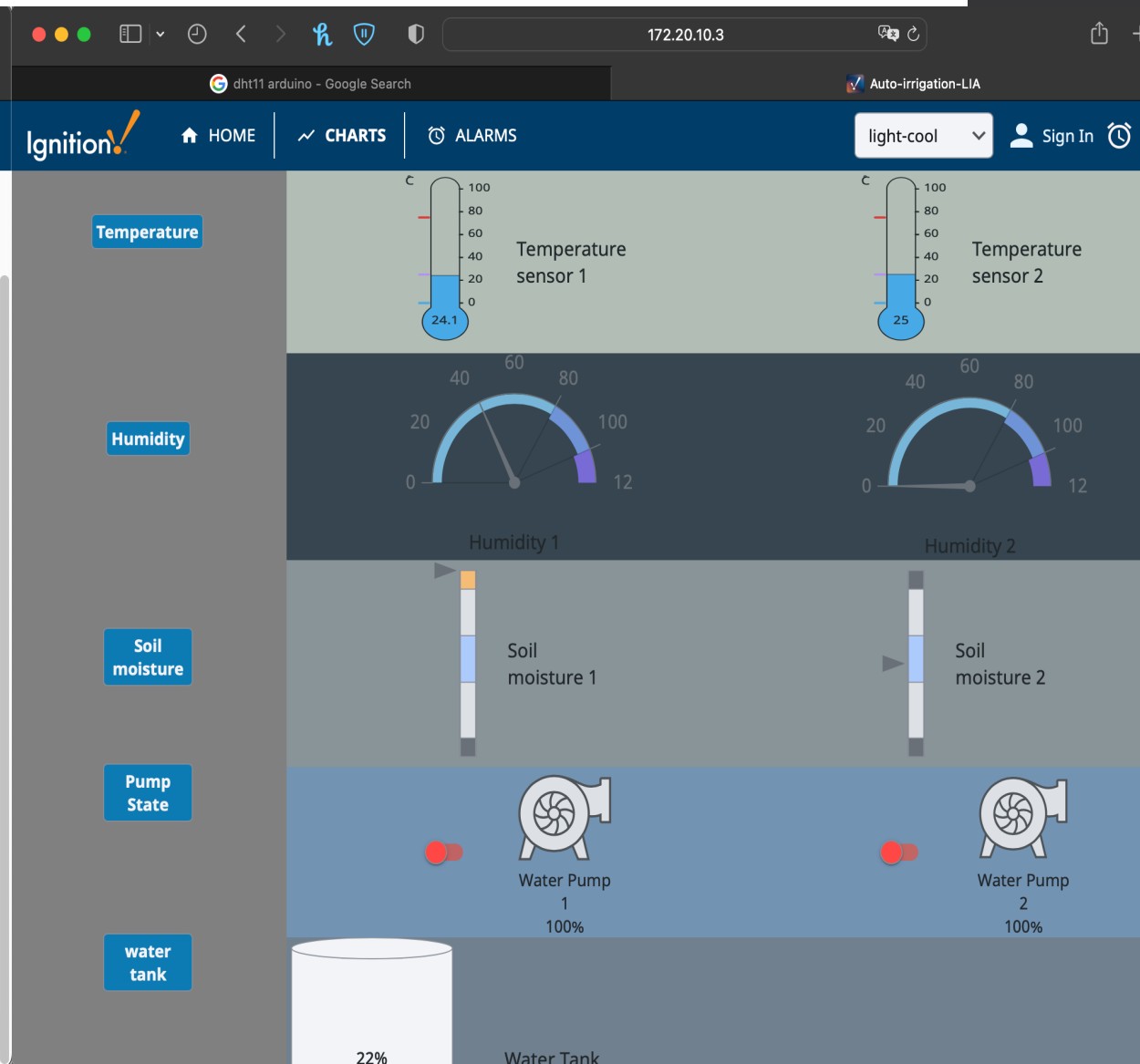
Using Ignition Designer to build an interactive HMI:



This HMI allows you to bind values from topics sent from your client into Tags,

-then you can display the Tags in creative ways.
-Reading live data (temperature, humidity, soil moisture and tank water level)
-In addition to controlling two water pumps!

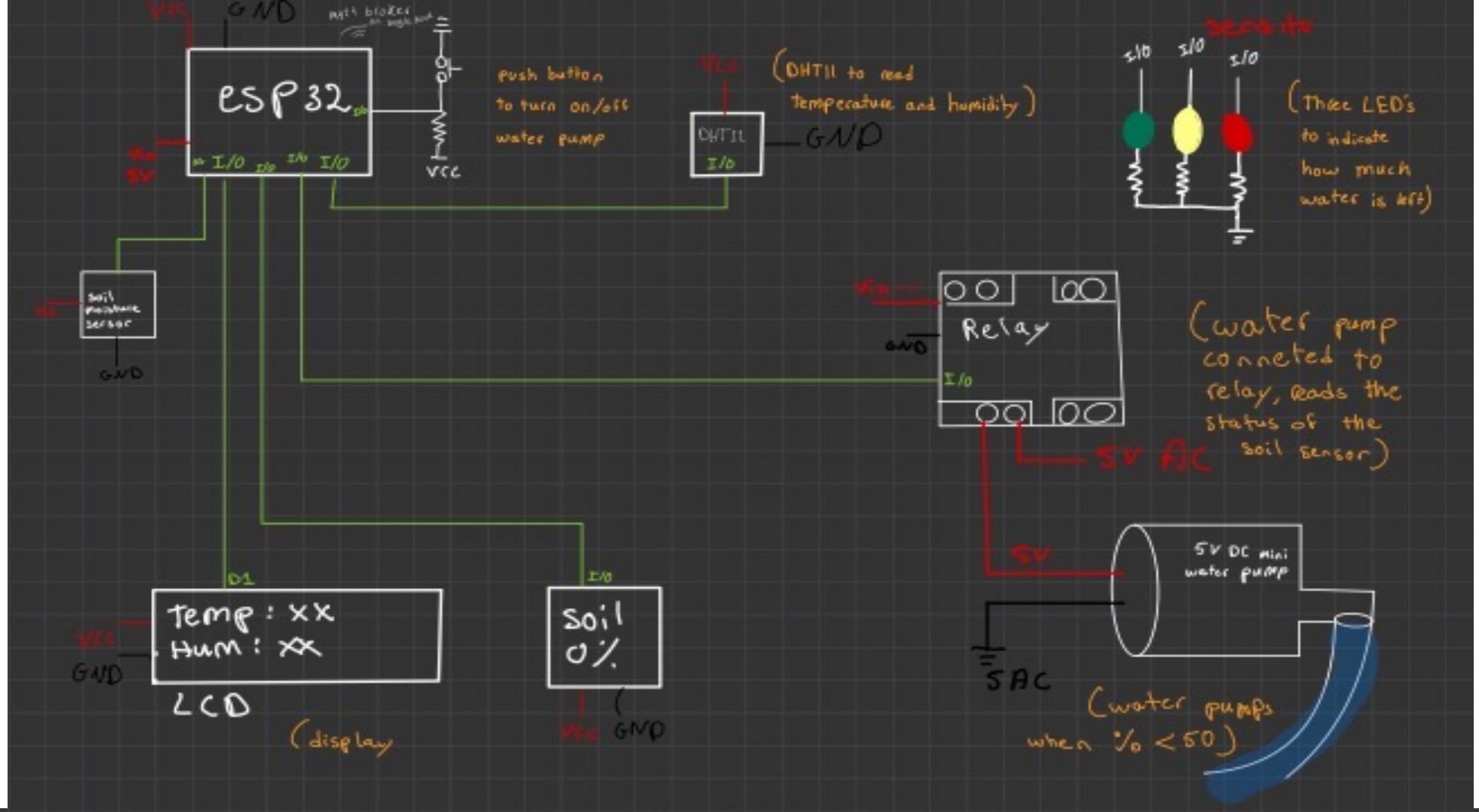




```
void readAndPublishSensorData() {
  // Read temperature and humidity from DHT11
  float h = dht.readHumidity();
  float t = dht.readTemperature();

  // Read soil moisture
  int soilMoistureValue = analogRead(SOIL_MOISTURE_PIN);
  int soilMoisturePercent = map(soilMoistureValue, 2544, 910, 0, 100);

  // Read water level
  int waterLevelValue = analogRead(WATER_LEVEL_PIN);
  int waterLevelPercent = map(waterLevelValue, EMPTY_VALUE, FULL_VALUE, 0, 100);


  // Publish sensor data
  snprintf(msg, 50, "Temperature: %.1f, Humidity: %.1f", t, h);
  client.publish("esp/dht2", msg);
  snprintf(msg, 50, "%.1f", h);
  client.publish("esp/hum", msg);
  snprintf(msg, 50, "%.1f", t);
  client.publish("esp/temp", msg);

  snprintf(msg, 50, "Soil Moisture: %d%%", soilMoisturePercent);
  client.publish("esp/soil2", msg);
```

```
void callback(char* topic, byte* payload, unsigned int length) {
  String messageTemp;
  for (int i = 0; i < length; i++) {
    messageTemp += (char)payload[i];
  }

  if (String(topic) == "esp/waterpump/control2") {
    if (messageTemp == "enable") {
      pinMode(RELAY_PIN, OUTPUT);
    } else if (messageTemp == "1") {
      digitalWrite(RELAY_PIN, LOW);
      client.publish("esp/pump/state2", "on");
    } else if (messageTemp == "0") {
      digitalWrite(RELAY_PIN, HIGH);
      client.publish("esp/pump/state2", "off");
    }
  }
```

Snippets of the code showing how to publish and subscribe using the ardiuno IDE:



Sketch of how the system works:

Thank you!